Virtual Infrastructure for Wireless Ad Hoc Networks

by

Seth Gilbert

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2007

Author .....

Department of Electrical Engineering and Computer Science
August 23, 2007

Certified by .....
Nancy Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Virtual Infrastructure for Wireless Ad Hoc Networks

by

Seth Gilbert

Submitted to the
Department of Electrical Engineering and Computer Science
on August 23, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

One of the most significant challenges introduced by ad hoc networks is coping with the unpredictable deployment, uncertain reliability, and erratic communication exhibited by emerging wireless networks and devices. The goal of this thesis is to develop a set of algorithms that address these challenges and simplify the design of algorithms for ad hoc networks.

In the first part of this thesis, I introduce the idea of *virtual infrastructure*, an abstraction that provides reliable and predictable components in an unreliable and unpredictable environment. This part assumes reliable communication, focusing primarily on the problems created by unpredictable motion and fault-prone devices. I introduce several types of virtual infrastructure, and present new algorithms based on the replicated-state-machine paradigm to implement these infrastructural components.

In the second part of this thesis, I focus on the problem of developing virtual infrastructure for more realistic networks, in particular coping with the problem of unreliable communication. I introduce a new framework for modeling wireless networks based on the ability to detect collisions. I then present a new algorithm for implementing replicated state machines in wireless networks, and show how to use replicated state machines to implement virtual infrastructure even in an environment with unreliable communication.

Thesis Supervisor: Nancy Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

In many ways, the best part of my years at MIT has been the amazing people that I have had the opportunity to meet, to collaborate with, and—in many cases—to get to know personally. Without their ideas, their enthusiasm, and their support, this thesis would never have happened.

The Thesis Committee

First and foremost, I would like to thank **Nancy Lynch**. Even more than her indispensable advice and technical skill, it has been her enthusiasm and positive attitude that has made this thesis possible. From the very first time we met, she was saying, "I think there's some exciting stuff to work on here!"

I would also like to thank the other members of my thesis committee, **Erik Demaine**, **Sam Madden**, and **Jennifer Welch**. They have provided valuable feedback and interesting perspectives on the ideas contained in this thesis.

The Virtual Infrastructure Project

I would like to call particular attention to the role played by **Jennifer Welch** and **Shlomi Dolev**, both early collaborators on this project. Many of the ideas presented in this thesis have grown out of conversations and e-mails with Jennifer and Shlomi, who posed the original question, "What sort of useful things can we do with swarms of mobile nodes?" It was this question that led to the "GeoQuorums approach" to developing algorithms for mobile ad hoc networks [32], which then evolved into the idea of virtual infrastructure. I am ever grateful for their friendly and welcoming attitude, and for their help on this project.

Alex Shvartsman was also an important contributor to the initial development of virtual infrastructure. Aside from his role in this thesis, he was also a source of helpful advice in the transition from industry to graduate school, and was a vital collaborator on my master's thesis.

Tina Nolte has been another key contributor to the virtual infrastructure project. Her work on timing-related issues (e.g., [29, 30]), self-stabilization (e.g., [35]), and applications (e.g., [83]) has played an important role in further developing the ideas in this thesis. Of particular interest is the idea of using virtual infrastructure for the purpose of coordinating the motion of mobile entities, first developed in a paper by **Sayan Mitra**, Tina Nolte, and Nancy Lynch [69]. Others who have been a part of the virtual infrastructure project include **Elad Schiller** and **Limor Lahiani**. I would also like to mention Matthew Brown and Michael Spindel who have pioneered the implementation of virtual infrastructure.

Many of the ideas developed in the second part of this thesis have grown out of conversations with **Gregory Chockler**, **Calvin Newport**, and **Murat Demirbas**. Together, we developed a model for wireless networks that accounts for contention and collisions on the communication channel; together we developed new algorithms for consensus in wireless networks that led directly to the emulation algorithm described in Chapter 10. More recently, Calvin Newport has been developing new ideas that

extend the virtual infrastructure notion presented in this thesis, considering problems caused by un-cooperative (perhaps, malicious) devices, and studying the uses of virtual infrastructure in the context of routing.

Collaborators and Colleagues

I have been fortunate to work with many other excellent collaborators and colleagues during my years at MIT. Let me begin by thanking **Charles Leiserson**, who has been an invaluable source of advice and inspiration. I still recall his first piece of advice when I arrived at MIT: "Always try to find the surprising fact; that's what makes for good research." This has been a guiding idea in my research program every since.

I would also like to thank **Michael Bender**, who has been a good colleague and friend. He has been a continual source of exciting (and sometimes offbeat) problems, interesting ideas, and the occasional *bon mot*. His unrelenting optimism and obvious enthusiasm has made working with him a pleasure.

Other collaborators of note include **Jeremy Fineman**, **Bradley Kuszmaul**, **Vincent Gramoli**, **Peter Musial**, **Matthew Lepinski**, **David Liben-Nowell**, **April Rasala Lehman**, and **Jacob Beal**.

Lastly, the TDS group has been a good home for me during my years at MIT, and I would like to thank all of is members during those years, including **Ling Cheung**, **Gregory Chockler**, **Murat Demirbas**, **Rui Fan**, **Roger Khazan**, **Dilsun Kirli**, **Carl Lividas**, **Sayan Mitra**, **Calvin Newport**, **Tine Nolte**, **Joshua Tauber**, and **Shinya Umeno**.

Family

Finally, let me conclude by thanking my family for their help and support over the last thirty years. And I would like to thank Valerie for everything; without her, none of this would have been possible.

Contents

Introductory Material

Concluding Material

List of Figures

Chapter 1

Introduction

There are several significant challenges associated with developing algorithms for ad hoc networks. First, the devices are unreliable: the physical hardware is often small and fragile; the batteries are easily exhausted; users may interrupt ongoing protocols by turning devices off. Second, the devices are often mobile, moving in an unpredictable manner, leaving the algorithm designer uncertain as to which devices may be participating in which protocols. Third, communication is unreliable: the devices communicate using wireless radios that make use of a shared spectrum that is subject to message collisions and other forms of interference. All of these challenges can be lessened by the deployment of a fixed infrastructure that is more reliable, less mobile, and supports more efficient communication. Unfortunately, it is often impractical—due to logistical or cost-related concerns—to deploy such a fixed infrastructure.

In this thesis we develop a set of algorithms that address the unpredictable and unreliable behavior exhibited by wireless ad hoc networks. These new algorithms, together with the new techniques and abstractions introduced in this thesis, simplify the design of algorithms for wireless ad hoc networks. Collectively, we refer to these abstractions and the algorithms that implement them as *virtual infrastructure*. Virtual infrastructure (VI) replaces the fixed infrastructure found in more traditional networks, thus compensating for unpredictable and unreliable behavior.

1.1 Ad Hoc Networks and Mobile Devices

Throughout this thesis, we focus on developing algorithms for ad hoc networks consisting of a large number of small, often mobile, devices that are deployed in an entirely ad hoc manner over some geographic region. These devices may be sensors, such as the Berkeley motes (see e.g., [87]), which are primarily static devices designed to collect and process data. Alternatively, these devices may be more mobile, such as handheld PDAS or cell phones.

All of these devices, however, share certain attributes. First, the devices are small and battery powered, and tend to be fault prone. Second, they tend to be deployed in an ad hoc manner: sensor networks are often deployed through random distribution, as from an airplane, rather than through careful placement; handheld devices tend

to travel with their owners, rather than as designated by an algorithm designer.

Third, these devices communicate primarily using wireless radio, rather than through a fixed network. Two devices can communicate only if they are geographically near to each other; long-distance communication requires multi-hop communication. As a result, it can be quite difficult in a network of mobile devices to ensure reliable point-to-point communication between devices.

Moreover, wireless communication is notoriously unreliable: messages may be lost due to collisions, electromagnetic interference, or other network anomalies. Moreover, the collisions may be non-uniform: some devices may receive a message, while others may not. The devices, however, have the ability to detect collisions: when a receiver misses a message it should have received, it can detect the disruption and report a collision. (Note that the transmitter, of course, cannot tell whether its messages was received or not, i.e., receives no notification of a collision.)

A final key assumption is that the devices have reliable access to a synchronized time source and a localization service that provides each device with its location.

1.2 Background

There are a wide variety of scenarios in which it seems more practical to equip a set of devices with radios, forming an ad hoc network, than to build a fixed network infrastructure. For example, firefighters in a burning building or soldiers on an enemy battlefield may need to coordinate their operations; cars on a highway may coordinate to provide a cooperative driving experience. In these (oft-cited) cases, it may be prohibitively expensive to deploy a fixed infrastructure, especially when it must be deployed over a large area under potentially hostile conditions. By contrast, radio transceivers are relatively cheap and easy to integrate into current devices. Unfortunately, it has long been recognized that a major barrier to deploying ad hoc networks is the complexity of developing algorithms in such an unpredictable environment. In this section, we discuss a variety of problems and protocols that arise in the context of wireless ad hoc networks, and some of the difficulties that arise.

1.2.1 Point-to-Point Routing

In wired networks, the basis of almost every high-level protocol is an efficient and reliable point-to-point communication protocol. The TCP/IP protocol stack [13, 88, 89], along with BGP routing [92], enables devices on the internet to communicate reliably with each other, despite congestion, message loss, and occasional failures.

Naturally, then, the first attempts at developing algorithms for ad hoc networks focused on providing reliable point-to-point communication, despite the constraints of wireless communication and the requirement of geographic proximity (see, e.g., [42, 84,85,93]). Under this paradigm, applications are oblivious to the underlying wireless network; they simply send and receive messages just as in the wired internet setting, and the routing layer compensates for the unpredictable and changing network.

There are two fundamental problems with this approach. The first problem is pragmatic: it can be quite expensive to implement a routing layer that entirely masks the underlying ad hoc behavior. Recall that long-distance communication can be quite expensive; if the application is entirely oblivious to the underlying network and the underlying topology, then the resulting protocol may spend significant time and energy sending messages needlessly across the network. If the devices themselves are mobile, the difficulty increases substantially: maintaining a routing network despite constant change can be expensive, if not impossible: the routing layer must continually monitor the changing network and exchange information to update its routing tables. In the worst case, where dynamic change is frequent, most protocols resort to flooding every message to every node in the network. It remains an open question whether existing algorithms can actually scale, in practice, to large mobile networks.

The second problem with relying on reliable point-to-point communication as a basis for ad hoc network protocols is philosophical: for many of the scenarios in which ad hoc networking appears useful, point-to-point routing is not the most useful or desirable form of communication. In particular, many of the applications for ad hoc networks are deeply rooted in geography, while point-to-point routing is oblivious to real geography. For example, when coordinating rescue workers, it is often important to coordinate reliably with nearby devices; moreover, accurately and efficiently locating individuals may be critical; and certain standing orders or danger warnings may be associated with a given location. None of these problems can be solved simply with a location-oblivious routing protocol. By contrast, wireless communication hardware is well-suited for this style of multicast communication. Protocols that focus on point-to-point communication often mask this useful capacity.

1.2.2 Wireless-Aware and Location-Aware

In response to these problems, many researchers began to focus instead on *wireless-aware* and *location-aware* algorithms. Instead of designing a protocol that uses point-to-point routing as a building block, a wireless-aware protocol is built directly on a wireless MAC layer and can take advantage of the underlying network topology and wireless capabilities. There have been a variety of approaches to building wireless-aware applications, mostly varying in the properties of the underlying MAC layer. For example, Kuhn et al. describe a protocol for clustering in a static network based on minimal MAC layer assumptions [55]. By contrast, Malpani et al. and Walter et al. present protocols for leader election [74] and mutual exclusion [102] (respectively) in a model with reliable communication, but changing topologies. The main difficulty with the wireless-aware approach is the complexity of the resulting algorithms. In many cases, it is nearly impossible to analyze the algorithms in a truly dynamic mobile environment. Even for static networks, analysis may be quite difficult: for example, many depend on complicated backoff protocols (to reduce contention on the wireless channels); each protocol requires a new (and often complicated) analysis. Finally, the close dependence on MAC layer attributes may lead to difficulties and inefficiencies as MAC layers and radio technologies evolve. For example, many protocols assume that message loss is relatively uniform (see, e.g., [10, 21, 22]), but that detecting collisions

is impossible. More recent research has indicated instead that message loss is not uniform, but that current MAC layers can in fact detect collision.

An alternative approach is to develop primitive building blocks that are wireless-aware, in contrast to point-to-point routing, which is location-oblivious. These primitives can then be themselves used to construct higher-level applications. Many of these building-block protocols take advantage of location information to simplify problems of coordination and communication. One of the most common wireless-aware and location-aware primitives is *geocast*, the geographic analogue to point-to-point routing: a geocast service delivers a message to a location, rather than to a specific device; any device near to the target location receives the message. A variety of protocols for geocast have been developed, analyzed, and simulated (see, e.g., [17,50,56,78]).

A second class of wireless-aware primitives provides location-based storage (see, e.g., [65,91]). These protocols use locations as a repository for data, allowing nodes to share data at known locations. In contrast to the algorithms presented in this thesis, however, these algorithms make only ad hoc attempts at guaranteeing reliability when devices fail or move. The *PersistentNode* abstraction by Beal [11, 12] also builds a data-storage primitive. A PersistentNode is a virtual entity that travels around a static (rather than mobile) sensor network. It can carry with it some state, but does not provide any consistency guarantees with respect to the data being stored.

Luo et al. attempt a more general approach to building an abstraction layer of wireless-aware services [68]. In NASCENT, they develop a suite of middleware services, including token circulation, leader election, and reliable broadcast. These services are then made available to build higher level applications. Others, such as Greenstein et al. [39], have also attempted to develop a general toolkit for developing algorithms for ad hoc networks.

1.3 Virtual Infrastructure

In this thesis, we introduce a new approach to the problem of building wireless-aware and location-aware algorithms: instead of developing a specific service or application, we suggest the use of *virtual infrastructure* as a general abstraction. Virtual infrastructure provides many of the advantages of a fixed infrastructure, in terms of simplicity and algorithm development, while simultaneously tolerating an ad hoc, potentially hostile, environment in which fixed infrastructure may be overly costly and impractical. In this thesis we describe two types of virtual infrastructure: virtual objects and virtual nodes.

1.3.1 Virtual Objects

The most basic type of virtual infrastructure introduced in this thesis consists of *virtual objects*. A virtual object is akin to a reliable storage unit deposited at some known location in the network. Each real node, i.e., *client*, can store and retrieve information from the virtual object. The object can implement any "variable type," and supports atomic invoke/response semantics (see, e.g., [70], Chapter 9).

A virtual object is (relatively) reliable, even though the set of real nodes that reside near—and implement—the object may be continuously changing. As long as some real nodes reside near the virtual object, it can continue to operate[1]. Virtual objects generalize some of the previously mentioned approaches for location-aware data storage [11, 12, 65, 91], and were first introduced in [32, 33].

1.3.2 Virtual Nodes

A virtual node is a natural extension of a virtual object. Instead of simply storing data and passively responding to requests, the virtual node can process data, send messages, and initiate actions. A virtual node resembles a reliable server, that is, a piece of computing infrastructure residing at some well-known location. Clients can send and receive messages to and from the virtual node, just as they would interact with a real device. If they are near to the virtual node, they may communicate with it via "local broadcast" communication; if the virtual node is farther away, they may communicate using a GeoCast service. Similarly, the virtual nodes may communicate with each other. Each virtual node is an arbitrary I/O automaton [70] (without tasks or fairness), and thus can execute any arbitrary (untimed) program.

Much like virtual objects, virtual nodes are designed to be more reliable than the individual mobile nodes in the underlying network. As long as some real nodes reside near the virtual node, it can continue to operate. If a virtual node fails (due to regional depopulation, say), it can recover if mobile nodes again return to the region near to the virtual node.

Moreover, a virtual node may be mobile, traveling on a predictable path through the network. The basic idea of executing algorithms on virtual *mobile* nodes (in contrast to static virtual infrastructure) was inspired by the development of *compulsory protocols* [20, 40, 66], and was first introduced in [31]. The basic observation in compulsory protocols is that if mobile nodes moved in a programmable way, algorithms could take advantage of the motion, providing elegant and simple solutions. This idea is illustrated by Hatzis et al. [40], who introduce the notion of a *compulsory* protocol, one that requires a subset of the mobile nodes to move in a pre-specified manner. They present an efficient compulsory protocol for leader election. The routing protocols of Chatzigiannakis et al. [20] and Li et al. [66] provide further evidence that compulsory protocols are simple and efficient. Using virtual mobile nodes, it is possible to take advantage of compulsory protocols even in networks where the underlying mobile nodes in fact do *not* behave in the desired manner.

1.4 Implementing Virtual Infrastructure

In this thesis we present three different algorithms for implementing virtual infrastructure. The first two of these algorithms are designed for a network that guarantees

[1] As presented in this thesis, there is no mechanism for virtual objects to recover; the same techniques that are employed to support the recover of virtual nodes, however, can be adopted for virtual objects.

reliable communication: the first implements virtual objects, and the second virtual nodes. The third algorithm is designed for a network in which communication is unreliable, that is, susceptible to collisions and lost messages.

All the algorithms in this thesis for implementing virtual infrastructure are, fundamentally, based on replicated state machine techniques. Each virtual infrastructure component is implemented by a set of participants (*replicas*) that maintain the state of the virtual infrastructure component. Whenever the state is updated, as a result of either a message received or a spontaneous action, all the replicas perform a consistent update, maintaining the replicated state.

The main challenges lie in dynamically determining an appropriate set of replicas as the underlying system changes, ensuring consistency even as the set of replicas is continuously changing, and ensuring that the resulting emulation is efficient.

Each virtual infrastructure component is emulated by a set of replicas that reside near the actual location of the virtual object or node. As real devices move toward and away from the virtual component, and as real devices join and leave the system, the set of replicas changes continuously. In particular, a device is able to examine its current location and determine whether to join or leave the emulation; the use of hysteresis may well improve efficiency.

In order to maintain consistency, each replica must apply updates in the same order. One way to achieve this is to use a (local) totally ordered broadcast service that ensures that messages sent within a specific region—the area of emulation of the virtual component—are delivered in the same order to all active participants. When the network guarantees reliable and timely communication, it is possible to build a totally ordered communication service using timestamp-based techniques originally developed by Lamport [58] in the context of fixed-infrastructure replicated state machines. When a node joins the emulation, it first requests a copy of the replicated state, and at the same time begins participation in the totally ordered broadcast service. On receiving a copy of the replicated state, the new node can update the state (adjusting for changes that may have occurred while the replica message was in transit) and begin participating in the emulation. In Part I of this thesis, we present protocols to implement virtual infrastructure based on these ideas.

Unfortunately, collisions and message loss may disrupt communication, making it difficult to implement totally ordered broadcast. In particular, the timestamp-based techniques for totally ordering messages do not adapt well to such an environment since it is difficult to determine when everyone has received a particular message. Most other prior techniques for implementing replicated-state machines, such as Paxos [60], also do not adapt well to a wireless environment. For example, many require all the participants to communicate. In a wireless setting, however, increased communication results in increased rates of collision; in such a setting, Paxos might never terminate.

Thus, in Part II of this thesis, we focus on the problem of implementing virtual infrastructure in collision-prone wireless networks. The protocol is based on a two-round consensus protocol in which all the replicas agree on each update to the replicated state. However, if consensus is used directly as a building block, the resulting replicated-state machine protocol is not suitable for implementing virtual infrastructure. In particular, the resulting emulation is not efficient: it results in mes-

sages that may be arbitrarily large, and there may be long delays in implementing even a single step of the virtual infrastructure component.

We therefore introduce a special type of replicated-state machine that is collision-aware and suitable for implementing virtual infrastructure. Instead of simply repeating the consensus algorithm for each step of the state machine, it adapts the basic consensus algorithm ideas to develop a more efficient protocol that guarantees efficient virtual nodes and efficient communication.

1.4.1 Performance

Virtual infrastructure is of course only useful when the virtual components perform efficiently and with low latency. For each algorithm implementing virtual infrastructure, this thesis analyzes the conditional performance.

The first two algorithms for implementing virtual infrastructure guarantee good performance under the assumption that the underlying physical network is well-behaved, e.g., it delivers messages within a bounded time. By good performance, we mean that each action taken by a virtual object or virtual node (e.g., sending or receiving a message) is emulated in only a small constant number of message delays.

The third algorithm for implementing virtual infrastructure (presented in Part II of this thesis) guarantees good performance if the underlying network is *eventually* well-behaved. In contrast to Part I, the second part of this thesis assumes a synchronous but unreliable broadcast service. If, eventually, messages are delivered reliably (and other stabilization criteria hold) in the underlying network, then the virtual infrastructure guarantees that eventually the *virtual* network is also well-behaved: messages are delivered reliably and other good criteria hold. Moreover, the implementation itself is efficient, requiring only a constant number of rounds of communication and constant-sized message-overhead to implement each virtual round.

1.5 Overview of the Thesis

The thesis is divided into two main parts. The first part of the thesis focuses on introducing the concept of virtual infrastructure in the context of reliable communication, while deferring many of the details related to collisions and message loss. The second part focuses on the more practical problem of designing and developing virtual infrastructure in real, collision-prone, ad hoc networks.

Part I

In the first part of the thesis, we assume that the mobile nodes can communicate reliably. Thus, Part I focuses on the other two major problems of ad hoc networks: fault-prone devices and unpredictable motion. The third problem, unreliable communication, is postponed to the second part of the thesis. By avoiding the problems of unreliable communication, Part I allows for an introduction to the basic ideas of

virtual infrastructure, and allows a simple presentation of algorithms that contain many of the major ideas used in Part II of the thesis.

Chapter 2. We begin by introducing a model of mobile ad hoc networks, a variation on the commonly used "unit-disc graph" model for wireless networks. Communication is either local (via a local broadcast service) or long-distance, via a GeoCast service. We discuss in this chapter some of the existing GeoCast protocols that may be used to implement this service. An alternate model that does not include long-distance GeoCast is discussed in Chapter 7.

Chapter 3. We then define the two main types of virtual infrastructure: the Virtual Object Model and the Virtual Node Model.

Chapter 4. In this chapter, we present a protocol that implements the Virtual Object Model in a mobile ad hoc network. This protocol, the Virtual Object Emulator, takes advantage of "focal points" where mobile nodes congregate to implement virtual objects. We argue that the protocol correctly emulates the Virtual Object Model, and we present a brief analysis of the efficiency of the Virtual Object Emulator.

Chapter 5. Next, we consider the Virtual Node Model. I extend the Virtual Object Emulator, developing a protocol that implements the Virtual Node Model. The Virtual Node Emulator is quite similar conceptually to the Virtual Object Emulator. The analysis, however, is more involved as the virtual entities being emulated are more powerful: virtual nodes can initiate actions on their own, and they can communicate directly with each other. We argue that the protocol correctly emulates the Virtual Node Model, and we present a brief analysis of the efficiency of the Virtual Node Emulator.

Chapter 6. In this chapter, we describe an application that can be implemented using the Virtual Object Model: a reliable, reconfigurable read/write shared memory. This application is built using virtual objects, which store the data and ensure the reliability and availability of the shared memory. This chapter serves as an example of how virtual infrastructure can be used to simplify the development of otherwise complicated applications. In particular, it is relatively simple to understand the behavior of the protocol, which would otherwise be complicated by analyses of mobility and failure rates.

Chapter 7. We conclude Part I with a discussion of how to modify the protocols described in the preceding chapters to use only local communication. The basic model introduced in Chapter 2 allowed for both some local communication ("fpcast") and long-distance communication (GeoCast). In practice, wireless networks typically contain only simple local broadcast. (In fact, one of the potentially compelling applications of virtual infrastructure is to build reliable long-distance communication.) Thus, in this section we describe a virtual node-based infrastructure that supports

only local communication, and describe how the protocol described in Chapter 5 can be modified to execute in an environment with only local communication.

Part II

In the second part of the thesis, we focus on the problem of implementing virtual infrastructure in wireless networks with unreliable communication. In general, this part is based on many of the ideas originally introduced in Part I; however, unreliable communication, collisions, and lost messages introduce a variety of complications that require some new techniques and new approaches.

Chapter 8. We begin the second part by presenting a more realistic model for wireless networks. The new model captures many attributes of real networks, such as (i) non-uniform patterns of message loss where different nodes receive different sets of messages; (ii) collision detection, which allows devices to detect anomalies in receiving messages; and (iii) reliable time and synchronous communication, which allows nodes to execute algorithms in synchronous rounds. We assume that the collision detectors are *complete*, meaning that they detect all collisions, and *eventually accurate*, meaning that, eventually, they report no false positives. The new model also provides devices with contention managers, which use backoff protocols to reduce the contention on wireless channels. Unlike in Part I, all communication is local.

Chapter 9. Next, we introduce a virtual node-based infrastructure that is compatible with a collision-prone network. In particular, the virtual infrastructure model introduced in this chapter reproduces many of the properties of the underlying wireless network: communication is local, and occurs in synchronous rounds, clients and virtual nodes can detect collisions, and both clients and virtual nodes have access to contention managers which can help to reduce the number of collisions.

Chapter 10. In this chapter, we present an algorithm that implements the virtual infrastructure model described in Chapter 9. Much like the emulator protocols in Part I, this algorithm is based on a replicated-state-machine approach. However, unlike Part I, this replicated state machine is collision aware, in that it can tolerate collisions and cooperate with a contention manager to reduce collisions.

At the heart of the emulator is a three-phase "agreement protocol" in which the replicas for each virtual node try to agree on an execution of that virtual node. This "agreement protocol" is derived from an earlier consensus protocol for wireless networks [23, 79], and is reminiscent in style of a three-phase-commit protocol. Unlike these earlier protocols, however, the "agreement protocol" does not always result in the replicas coming to single decision; disagreement is (inevitably) a possible outcome. It does guarantee, however, that when the network is well-behaved, the replicas agree; moreover, it ensures that the disagreement is limited such that the replicas can converge eventually on a single virtual node execution.

Chapter 11. We then proceed to argue that the protocol presented in Chapter 10 is a correct implementation of the virtual infrastructure layer. The main goal of the chapter is to transform an arbitrary execution α of the underlying (physical) system into an execution γ of the virtual infrastructure such that the clients cannot distinguish between α and γ. This indistinguishability property implies that, from the clients' perspective, the emulator is successfully implementing the virtual infrastructure. The proof proceeds by first constructing an execution of each of the clients and virtual nodes, along with the other components of the virtual system, and then pasting the executions together to produce a unified execution γ. The main difficulty is in constructing the individual virtual node executions in such a way that they satisfy the requisite properties.

1.6 Related Work

In this section, we discuss some of the other work related to the ideas presented in this thesis. We begin in Section 1.6.1 with a review of the progress that has been in the area of virtual infrastructure. Next, in Section 1.6.2, we review other abstractions for ad hoc networks. In Section 1.6.3, we describe prior research on replicated state machines, the key algorithmic tool in implementing virtual infrastructure. Finally, in Section 1.6.4, we present some background on radio networks and wireless communication.

1.6.1 Virtual Infrastructure

We first introduced the idea of virtual infrastructure in [32] in the context of virtual objects that are used to implement a reconfigurable, atomic memory. This paper was then extended in [33] to support general virtual objects. The protocol for virtual objects in Chapter 4 is derived from this paper, as is the example application in Chapter 6.

We extended the virtual infrastructure paradigm to include virtual nodes in [31]. The protocol in [31] extends the ideas in [32], and also introduces the idea of a *mobile* virtual entity. The protocol in Chapter 5 is derived from [31], as is the material in Chapter 7 on implementing virtual infrastructure in the context of local-only communication. This paper also introduced the idea that a virtual node might recover, even if it had failed (due to an insufficient number of replicas).

In [34], we introduce the idea of *autonomous* virtual nodes, that is, virtual nodes that determine their motion in an on-line fashion in response to ongoing sensing and computation. For example, an autonomous virtual node may choose to move itself toward a more populated region of the network, rather than languishing in a deserted corner of the world. Unlike other virtual node models, the autonomous virtual node moves discretely, where each move follows from a discrete programmatic output. They present an overview of a self-stabilizing protocol to implement this new virtual node layer, focusing on the difficulties in ensuring consistent membership in the context of an unpredictable virtual location. They also introduce new techniques for restoring

a failed virtual node. In particular, a key difficulty is determining when the virtual node has failed and needs to be recreated. They present three different techniques, one based on a network of static virtual nodes and two based on random-walk based techniques.

In [29, 30], we introduce a *timed* virtual infrastructure in which the virtual nodes represent timed I/O automata (TIOA). They present a *self-stabilizing* algorithm for implementing this virtual infrastructure model. They also describe how to implement a tracking service and a GeoCast routing service, which together instantiate a point-to-point routing service. More details on tracking can be found in [83], and more details on routing can be found in [35].

Tulone [99] extends [31], studying the restrictions on mobility necessary to ensure that a sufficient number of virtual nodes remain active, and developing robust quorum systems that allow for virtual nodes to restore their previous state after a failure.

Lynch et al. [69] develop a particularly interesting application of virtual infrastructure. They use virtual nodes to coordinate the motion of the clients in the system. The resulting protocol enables the mobile nodes to easily and efficiently form any particular geometric pattern in the plane, despite unreliable failures. Brown [15] has continued this direction, using a virtual infrastructure to construct an air-traffic control system in which virtual nodes act to coordinate the motion of airplanes. Brown focuses on the problem of free flight, where the individual airplanes can choose their own flight paths independently.

Brown et al. [14] have implemented a prototype virtual infrastructure on a set of handheld Compaq iPAQ devices. The protocol used in [14] is a simplified—and optimized—variant of the algorithm described in Chapter 5 and [31], including the modifications described in Chapter 7 (and first introduced in [31]). Using this prototype virtual infrastructure, we built a few simple applications, including a virtual traffic light that can be used to coordinate traffic at a busy intersection.

1.6.2 Other Abstractions for Ad Hoc Networks

There has been much work of late (and, in some cases, concurrently with the work in this thesis) on high-level programming languages and region-based abstractions for ad hoc networks, particularly sensor networks, e.g., [64, 80, 81, 103, 104]. Much of this work is complementary to the work on virtual infrastructures presented in this thesis: better programming languages are essential to simplifying the task of developing software for ad hoc networks; providing reliable virtual infrastructure with strong consistency guarantees can simplify the programming paradigm, regardless of the language employed.

The work of Nath and Niculescu [77] takes advantage of precalculated paths to forward messages in dense networks. Messages are routed along trajectories, where nodes on the path forward the messages. Similarly, prior GeoCast work (for example, [17, 78]) attempts to route data geographically. Sun et al. [98] use cars traveling on a highway to regionally broadcast alert information. In many ways, these strategies are ad hoc attempts to emulate some kind of traveling node. We provide a more general framework to take advantage of predictably dense areas of the network to per-

form arbitrary computation. A significant focus of these prior papers is *determining* good trajectories, a problem that we do not address.

1.6.3 Replicated State Machines

Many of the key techniques in implementing virtual infrastructure are derived from the literature on replicated state machines. The concept of implementing fault-tolerant services using replicated state machines (RSMs) was first introduced by Lamport in [58] and extended and popularized by Schneider in [94]. Lamport [57] also presents a technique for implementing a fault-tolerant state machine in a message-passing system with fail-stop replicas (i.e., with reliable failure detection). For stronger fault models, the Paxos protocol and its variants [61–63] provide an RSM implementation that is resilient to fewer than $n/2$ replica crashes, where n is the number of replicas, and that tolerates arbitrarily long periods of network instability. Castro and Liskov present a Byzantine resilient version of the Paxos protocol [19]. A separate line of research was dedicated to implementing RSMs on top of view-oriented group communication systems (e.g., [5, 36, 49]).

These protocols, however, were designed for traditional "wired" networks and therefore do not address several important concerns arising in wireless ad hoc environments. In particular, they lack the ability to dynamically adapt their patterns of communication to the number of participating nodes, and rely on *a priori* knowledge of the number of participants and their identities. An algorithm for atomic broadcast that tolerates dynamic networks with an unknown set of participants is presented by Bar-Joseph et al. [7]. Although their protocol can be used to implement a reliable state machine, it is nevertheless inapplicable in our setting as it assumes reliable, collision-free communication.

There has been a substantial body of prior work on developing TDMA schedules for wireless ad hoc networks where the number of participants is *a priori* unknown (e.g., [16, 41]). A TDMA schedule can be used to avoid collisions, which can simplify the problem of implementing an RSM. Note, however, that these techniques do not trivialize the implementation of an RSM as they result in initial periods of instability during which collisions may occur. Moreover, many of these algorithms are unsuitable for highly dynamic networks.

1.6.4 Wireless Communication and Radio Networks

Starting with a seminal paper by Bar Yehuda et al. [10], and followed by many others (e.g., [9, 21, 53]), reliable broadcast was studied in synchronous radio networks where a node is guaranteed to deliver a message in a given time slot if and only if exactly one of its neighbors is transmitting a message in this slot. In contrast to this model, in this thesis we develop a communication model that allows for unpredictable collision patterns which in particular, might result in non-uniform message loss. Such non-deterministic behavior is frequently observed in real networks [51, 106, 108], and in fact arises in simulations [23]. We also do not assume any advance knowledge of a node's neighbors and therefore, cannot attribute lost messages to specific nodes in

the networks. A variety of other variants to the reliable broadcast problem in a model similar to that of [10] have been considered in [4, 22, 25, 76].

Current practical research in wireless devices and networks motivates the basic model of a wireless network developed in Part II of this thesis. First, it is well known that wireless broadcast networks are inherently unreliable. Several recent experimental studies [37, 51, 105, 108] suggest that even with sophisticated collision avoidance mechanisms (e.g., 802.11 [1], B-MAC [86], S-MAC [107], and T-MAC [101]), and even assuming low traffic loads, the fraction of messages being lost can be as high as $20 - 50\%$.

The algorithms in Part II of this thesis rely on collision detectors to overcome uncertainties in message loss. The importance and practicality of having collision information available to applications was argued in [106]. Several existing MAC layers, such as B-MAC [86], already support some collision detection capability. Moreover, the recent study by Deng et al. [27] suggests that there is no technical obstacle to adding collision detection support to the current 802.11 protocol.

Part I

An Introduction to Virtual Infrastructure

Introduction

In the first part of this thesis, we introduce the idea of virtual infrastructure, and develop protocols to implement virtual infrastructure in wireless networks that support reliable communication. Part I focuses on two of the major problems of ad hoc networks: fault-prone devices and unpredictable motion. The third problem, unreliable communication, is postponed to the second part of the thesis.

We begin in Chapter 2 by introducing a model of mobile ad hoc networks. In this model, mobile nodes are fault-prone devices that move unpredictably in a two-dimensional plane. They communicate via various broadcast services: either local (via an fpcast service), or long-distance, via a GeoCast service (as per, say, [17, 50]).

Next, in Chapter 3, we formally introduce the idea of virtual infrastructure. This chapter first introduces the idea of virtual objects and virtual nodes, perhaps the key innovation in Part I of this thesis. We begin by defining the Virtual Object Model, a virtual infrastructure layer that resembles a typical distributed shared object model. In the virtual object model, clients interact with virtual objects, invoking operations and receiving responses. We then proceed to introduce the Virtual Node Model. Unlike the Virtual Object Model, in which the virtual entities are restricted to objects, the Virtual Node Model allows for the virtual entities to perform arbitrary computation. Communication between the "clients" and "virtual nodes" proceeds via a virtual GeoCast service.

In Chapters 4 and 5, we present protocols to implement the Virtual Node Model and the Virtual Object Model, respectively. Both of these protocols rely on "focal points," well-populated regions of the network, to implement the virtual entities: each focal point is associated with a virtual object or virtual node, and each mobile node in a focal point acts as a replica for the associated object. As mobile nodes arrive, leave, and fail, the set of replicas adapts to the current set of available mobile nodes. These protocols extend standard replicated-state-machine techniques to this new environment. We show that the two protocols are correct, and provide a brief analysis of their efficiency.

In Chapter 6, we present an example of an application that is built on the Virtual Object Model. Specifically, we show how to implement a reliable, reconfigurable read/write shared memory. The virtual objects are used to replicate data throughout the entire world, and a reconfigurable set of quorums are used to maintain consistency among the virtual nodes. The protocol is based on a simplified version of the RAMBOprotocols (see, e.g., [38]). We argue that the the resulting shared memory guarantees atomic read and write operations, and provide some analysis as to its

efficiency.

Finally, in Chapter 7, we consider networks in which all communication is local. That is, unlike in the previous chapters in Part I, there is no long-distance GeoCast service available, and no focal-point-oriented fpcast service available. We provide a brief description of a virtual node infrastructure (based on the Virtual Node Model introduced in Chapter 3) in which all communication is local (in contrast to the earlier model in which communication occurs via a virtual GeoCast service). We then describe how to modify the protocols described in Chapter 5 to implement this local virtual node layer.

Chapter 2

Wireless Ad Hoc Network Model

In this chapter, we describe the underlying model for a mobile ad hoc network. Figure 2-1 defines some of the notation used in Part I of the thesis. Figure 2-2 provides an overview of the system model.

The basic model consists of a bounded region of a two-dimensional plane, populated by mobile nodes. The mobile nodes communicate using an atomic local broadcast service, which we call fpcast, and a long-distance communication service, called GeoCast. Each mobile node receives discrete updates from an external "Geosensor" that notifies the mobile nodes of the real time and their location in the real world.

While we make no assumptions about the motion of the mobile nodes, we do assume that certain regions are usually "populated" by mobile nodes. We assume that there exists a finite collection of such regions in the plane, called *focal points*, such that (i) at any point during an execution, "enough" focal points remain "populated," ensuring that sufficiently many focal points remain available, and (ii) within each focal point, there is a reliable, atomic broadcast service available for the mobile nodes to communicate reliably with each other. A focal point can be static, meaning that it consists of a fixed geographic region, or a focal point can be mobile, changing throughout an execution.

We begin in Section 2.1 to describe the basic geometry of the world, focusing on the definition of a focal point. The definitions from this section are used later in other contexts (for example, the description of virtual objects and virtual nodes in Chapter 3). In Section 2.2, we describe the mobile nodes. In Section 2.3 we describe the two broadcast service: the fpcast service and the GeoCast service. In Section 2.4, we describe the liveness properties of the basic model.

2.1 Geometric Basics

The world consists of a bounded region of a two-dimensional plane[1]. We assume that there exists a finite collection of regions in the plane, called *focal points*. For each

[1]In fact, all the results described in this thesis carry over naturally to an arbitrary metric space. It is necessary only that the metric space supports appropriate broadcast services. In particular, it would be natural to consider three-dimensional space.

focal point, we identify a "focal point center," that is, a designated location within the focal point, and a "focal point region," that is, a portion of the plane covered by the focal point. A focal point can be static, meaning that it consists of a fixed geographic region, or a focal point can be mobile, changing throughout an execution[2]. We assume that the set of focal points is determined *a priori* and is common knowledge.

More formally, a focal point is defined by:

1. a unique identifier h chosen from the set O,
2. a function $\mathsf{fp\text{-}region}_h : \mathbb{R}^{\geq 0} \to \mathcal{P}(L)$ that maps each time t to a contiguous geographic region in the plane[3], the **focal point region**, and
3. a function $\mathsf{fp\text{-}center}_h : \mathbb{R}^{\geq 0} \to L$ that maps each time t to a point within that geographic region, the **focal point center**.

Notice that a focal point is **dynamic**, since its center and region can change with time. If the center and region of the focal point do not change with time, that is, if the two functions are constant, we say that the focal point is **static** and its maximum velocity is 0.

We define the radius of a focal point at some time t in the natural way: a focal point h has **radius** r at time t if r is the infimum value such that $\mathsf{fp\text{-}region}_h(t)$ is contained in a circle of radius r centered at $\mathsf{fp\text{-}center}_h(t)$.

Since the focal point center and region can change with time, we need to define the velocity of a focal point. If the focal point region is "rigid" and does not rotate, then we can define the velocity simply with respect to the focal point center. More generally, we can define the velocity in terms of how rapidly a point in the plane is "left behind" by the focal point region. For a dynamic focal point $h \in O$ whose geographic region is defined by the function $\mathsf{fp\text{-}region}_h(t)$, we define the maximum velocity as follows:

- Define $\overline{\mathsf{fp\text{-}region}_h(t)}$ to be the complement of $\mathsf{fp\text{-}region}_h(t)$, that is, all the points in the plane L not part of the focal point region for h at time t.
- For a point $\ell \in \mathsf{fp\text{-}region}_h(t)$, define $depth_h(\ell, t)$ to be the infimum distance between the point ℓ and the closest point in L that is not part of the focal point region for h at time t. It specified how close a location ℓ is to the edge of the focal point at time t.
- Define $v_{\mathsf{sup}}(h)$, a real number, to be the supremum velocity of focal point h as follows:

$$v_{\mathsf{sup}}(h) = \sup_{\substack{t,t' \in \mathbb{R}^{\geq 0},\ t \leq t' \\ \ell \in \mathsf{fp\text{-}region}_h(t) \\ \ell \notin \mathsf{fp\text{-}region}_h(t')}} \frac{depth(\ell, t)}{t' - t}$$

[2]Dynamic focal points, i.e., those that chance location during an execution, are used in implementing virtual *mobile* nodes. The mobility of virtual nodes can be useful in the context of moving data and computation through a network.

[3]By $\mathcal{P}(S)$ we refer to the power set of S, i.e., $\{s : s \subseteq S\}$.

Thus, the supremum velocity of a focal point h determines the minimum (in fact, infimum) length of time that it takes for some fixed point ℓ in a focal point region to cease to be in that focal point region. For example, if a point is at a distance d from the edge of a focal point region that has a supremum velocity of $v_{\mathsf{sup}}(h)$, then that point in the plane will remain within the focal point region for at least $d/v_{\mathsf{sup}}(h)$ time.

As a specific example, consider a focal point region that is defined by a circle of some fixed radius around the focal point center. Notice that in this case, the maximum velocity of the focal point is simply the maximum velocity of the focal point center. Define $v_{\mathsf{fp\text{-}max}}$ to be the maximum velocity $v_{\mathsf{sup}}(h)$ for any $h \in O$.

2.2 Mobile Nodes

The basic model is populated by mobile nodes, each assigned a unique identifier from the set I. We model the computation at each mobile node as an *asynchronous* I/O automaton; an asynchronous I/O automaton is a special case of a timed I/O automaton in which all the state variables are discrete, and in which the set of trajectories consists of all constant-valued mappings from all possible time intervals. (See Appendix A for more details on timed I/O automata.) As a result, it is reasonable to compose asynchronous automata with timed I/O automata.

The mobile nodes may join and leave the system, and may fail at any time. (We treat nodes leaving the system as having failed.) The mobile nodes can move on any continuous path in the plane, with speed bounded by a constant v_{max}. We assume there exists at least one node $i_0 \in I$.

In the following two subsections, we discuss the mobile nodes' interactions with their environment (Section 2.2.1), and their interactions with the regions of the world identified as focal points (Section 2.2.2).

2.2.1 The Real World

We model the environment, which interacts with the mobile nodes, using a timed *RealWorld* automaton (see [45, 71, 72] for a formal presentation of timed automata; see Appendix A for a brief review). The RealWorld automaton maintains in its state the current location of every mobile node, as well as the current real time. It also maintains the failure status of each mobile node. Finally, the RealWorld implements the various broadcast services that the mobile nodes use to communicate. We occasionally refer to these different functionalities of the RealWorld as "sub-components." (Notice that sub-components are not distinct automata, that is, they are *not* automata that can be composed to form the RealWorld.)

The Status Sub-Component. In order to model nodes joining and leaving the system, the RealWorld automaton maintains in its state an indication for each mobile node as to its status, i.e., whether it is asleep, awake or failed. Formally, when the execution begins, the RealWorld automaton is initialized with a set $A \subseteq I$ of mobile

nodes that begin the executions *awake*. Each mobile node is similarly initialized with an indicator as to whether it begins the execution awake or asleep. A new node i is woken up when the RealWorld automaton sends a wakeup_i signal to node i, and adds i to A. A node i fails when the RealWorld automaton sends a fail_i signal to node i and removes i from A. Throughout the first part of this thesis, for simplicity, we omit the formal details relating to a node waking up when presenting pseudocode: each mobile node automaton described in this part of the thesis can be trivially transformed to include a wakeup_i and a fail_i transition, and perform operations only when it is awake and not failed.

The GeoSensor Sub-Component. The RealWorld automaton also contains as a sub-component a *Geosensor*; the Geosensor updates each mobile node with its current location, and with the current real time, a nonnegative real number. It performs these updates via a geo-update action. The mobile node locations and the real time are the only two non-discrete state variables of the RealWorld state, and the only two non-discrete state variables in the system. We assume that each mobile node receives a geo-update from the Geosensor (i.e., the RealWorld) at least every time t_{upd}.

In practice, there are a number of ways to provide mobile nodes with location and time services. GPS is perhaps the most common means, but others, like Cricket [90], have being developed to remedy the weaknesses in GPS, such as the inability to operate indoors. Our algorithms can tolerate small errors in the time or location, though we do not discuss this here.

The Broadcast Service Sub-Components. The RealWorld automaton contains two further sub-components: an atomic local broadcast service, called fpcast due to its relationship with focal points, and a long-distance broadcast service called GeoCast. These are discussed in detail in Section 2.3.

2.2.2 Focal Points

Throughout the execution, nodes enter and exit the focal point regions; when a mobile node's location is in the focal point region at some point in time, we say that the mobile node is **inside** the focal point region. In the context of a focal point h, when we say that a mobile node i is inside h, we mean that it is inside the focal point region for h.

In order to compensate for the inexact location information available to the mobile nodes, we identify an "inner region" of the focal point region; we say that a mobile node is *well inside* the focal point region when it is in the inner region. The size of the buffer zone depends on two factors: the maximum speed of a mobile node, and the rate of geo-update events. The buffer zone is chosen to be large enough that a mobile node exiting a focal point has time to learn that it has left the inner region prior to leaving the focal point region altogether.

Informally, we say that a mobile node is **well inside** a focal point region when it has moved far enough into the focal point region: a node at location ℓ is **well inside**

a focal point at time t if:

$$depth(\ell, t) > t_{\mathsf{upd}} \cdot (v_{\mathsf{max}} + v_{\mathsf{fp\text{-}max}}) \ . \tag{2.1}$$

We refer to the locations ℓ that satisfy Equation 2.1 at time t to be the **inner region** of h at time t. In the context of a focal point h, when we say that a mobile node i is well inside h, we mean that it is well inside the focal point region for h.

Notice that according to this definition, if a node is *well inside* a focal point region at some time t, then during the most recent geo-update prior to time t the node learns that it is, in fact, *inside* the focal point region. Conversely, if a node is not *inside* a focal point region at time t, we can be certain that during the most recent geo-update prior to time t, the node learns that it is, in fact, not *well inside* the focal point region. (Note the contrast between *inside* and *well inside* in the previous claim.) We thus conclude the following:

Lemma 2.2.1. *Let α be any finite execution, let $i \in I$ be a mobile node, and let $h \in O$ be a focal point.*

- *If i is well inside h at the end of α, and if* $\mathsf{geo\text{-}update}(\ell, t)_i$ *is the last* $\mathsf{geo\text{-}update}_i$ *event in α, then i is inside the focal point region of h at time t.*
- *If* $\mathsf{geo\text{-}update}(\ell, t)_i$ *is the last* $\mathsf{geo\text{-}update}_i$ *event in α, and if i is well inside h at time t, then mobile node i is inside the focal point region of h at the end of α.*

Proof. Consider the first claim. Since the geo-update occurs at most time t_{upd} prior to the end of α, the relative distance travelled by the mobile node with respect to the focal point region during this interval of time is at most $t_{\mathsf{upd}} \cdot (v_{\mathsf{max}} + v_{\mathsf{fp\text{-}max}})$ (by the definition of $v_{\mathsf{fp\text{-}max}}$). Since i is well inside h, the depth in the focal point region is larger than this distance, and hence i is inside the focal point region at time t.

Next, consider the second claim. Again, since the geo-update occurs at most time t_{upd} prior to the end of α, the relative distance travelled by the mobile node with respect to the focal point region during this interval of time is at most $t_{\mathsf{upd}} \cdot (v_{\mathsf{max}} + v_{\mathsf{fp\text{-}max}})$ (by the definition of $v_{\mathsf{fp\text{-}max}}$). Since i is well inside h at time t, the depth in the focal point region is larger than this distance, and hence i is inside the focal point region at the end of α. □

In Chapters 4 and 5, we present protocols that take specific actions when a mobile node is well inside a focal point, according to its most recent geo-update. Lemma 2.2.1 allows us to conclude that whenever a mobile node takes such an action, it is in fact inside a focal point.

2.2.3 Motion-Controlled Devices

Throughout this section, we have assumed that the mobile nodes have no control over their own motion. In some practical situations, however, there may be an algorithm running on the mobile node that controls the physical motion of the device. For example, a mobile robot can direct its own motion in a suitable fashion; a vehicle

may choose its own velocity; machines in a factory may have some autonomous control over their actions.

Moreover, some of the most compelling uses for the virtual infrastructure paradigm presented in this thesis are those in which devices may have some control over their motion. For example, consider a battlefield scenario where a set of tanks wish to coordinate their motion according to some predetermined formation; in this case, the tanks might use virtual nodes to coordinate their motion. (See [69] for an example of using virtual infrastructure to solve a basic coordination problem among mobile robots.)

It is straightforward to augment the model presented here to include motion control. Specifically, each mobile node is assumed to produce (as an output) some motion control signal, for example, move(...). This signal may be in the form of a velocity vector, a target waypoint, or any other desired form of motion control signal. In practice, the most common such signal is acceleration. The RealWorld automaton, as a model of the environment, receives such signals as input and uses them to update the location of the mobile nodes.

2.3 Broadcast Services

In the first part of this thesis, we assume the availability of reliable communication services. (We relax the assumption of reliability in the second part of this thesis.) There are two different types of broadcast services that the mobile nodes use to communicate:

- fpcast_h, for each $h \in O$: a focal point broadcast service that supports reliable, totally-ordered communication among nodes in focal point h; each message broadcast is delivered eventually to every mobile node in the focal point region for h, and each node receives the messages in the same order.

- geocast: a long-range broadcast service that delivers messages to specified locations in the network; each message broadcast is eventually received by every node that is within some radius R of the designated destination location.

The first broadcast service, fpcast, is "local" in that, conceptually, it operates over a small region of the network, delivering messages to nearby nodes[4]. The second service, GeoCast, handles multi-hop message delivery: it can deliver a message from any location in the network to any other location in the network. Unlike a more traditional routing service, however, it delivers messages to a given *location* in the network, not to a given *mobile node*. In Chapter 7, we discuss how to implement virtual infrastructure when only local broadcast is available.

[4]While a focal point region could theoretically be quite large, we tend to think of a focal point region as a single-hop region of the network. Implementing the fpcast service over a multi-hop focal point region is presumably more difficult, leading to the conclusion that focal point regions are likely to be small.

Formally, the broadcast services are sub-components of the RealWorld automaton that models the external environment. For each broadcast service, we specify both the safety and liveness properties guaranteed by traces of the RealWorld automaton.

2.3.1 The fpcast Service

For each focal point, we assume that there is a broadcast service that guarantees reliable, totally-ordered local communication between mobile nodes within that focal point. The broadcast services guarantees the following properties: (1) *integrity*, meaning that every message delivered was previous broadcast; (2) *reliable delivery*, meaning that if two correct nodes reside in some focal point, and one sends a message, then it will eventually be delivered to the other; (3) *total order*, meaning that messages are delivered to all of the mobile nodes in the same order; and (4) *consistent delivery*, meaning that if a mobile node remains in a focal point for some period of time, there will be no "gaps" (with respect to the ordering) in the sequence of messages received by the mobile node. The service is defined here as "asynchronous," meaning that there is no bound on the delivery time for messages. Formally, this means that for every message broadcast, there is some ϵ that bounds the latency of the message. Later in Section 2.4, when making further assumptions as to the performance of the system, we assume that this message delay is bounded.

Formal Definition. The broadcast service for focal point $h \in O$, fpcast_h, supports the following interface for each mobile node i:

- Input $\mathsf{fpcast}(m)_{h,i}$
- Output $\mathsf{fpcast\text{-}rcv}(m)_{h,i}$

where m is an arbitrary message to be sent. Notice that the first index of the action indicates the focal point; the second index indicates the mobile node at which the event takes place. For mobile nodes that are in focal point h, the fpcast_h service satisfies the following properties:

***Integrity*:** For any message m and mobile node i, if an $\mathsf{fpcast\text{-}rcv}(m)_{h,i}$ event occurs, then an $\mathsf{fpcast}(m)_{h,j}$ event precedes it, for some mobile node j.

***Reliable Delivery*:** For each $\mathsf{fpcast}(m)_{h,i}$ event there exists some ϵ such that: assume that i is inside the focal point region for h when the event occurs, and assume that mobile node $j \in I$ (potentially $j = i$) is a mobile node satisfying the following:

- mobile node j is in the focal point region for h when the event occurs;
- mobile node j remains in focal point region until time ϵ after the event; and
- mobile node j does not fail until at least time ϵ after the event;

then an fpcast-rcv$(m)_{h,j}$ event occurs within time ϵ after the event, delivering the message m to node j.

***Total Order*:** For every execution, if every message sent by the fpcast$_h$ is unique[5], then there exists a total ordering $m_1, \ldots, m_k, \ldots$ of all messages sent by the fpcast$_h$ service during an execution; if some mobile node i receives messages m_r and m_t, then i receives m_r prior to m_t if and only if $r < t$.[6]

***Consistent Delivery*:** Assume there are three messages $m_1 < m_2 < m_3$, ordered according to the total order posited above. If node i receives messages m_1 and m_3, and remains inside the focal point region for the entire interval between receiving m_1 and m_3, then node i receives message m_2.

Notice that the *reliable delivery* property allows each message to have a different ϵ governing its delivery time. Thus, in an infinite execution, this definition allows for unbounded message delivery times. When discussion the performance of algorithms, we will make further assumptions, including placing an upper bound on ϵ. (See Section 2.4.)

The *consistent delivery* property is the only property not found in typical totally-ordered ("atomic") broadcast services (see, for example [8, 24, 48]). The *consistent delivery* property ensures that if a mobile node is within the focal point region for some interval of time, then it receives a sequence of messages in the total order. That is, if a mobile node receives some message m_s in the totally-ordered sequence of messages, and remains in the focal point region until it receives some later message m_t in the totally-ordered sequence of messages, then it also receives every message ordered after m_s and prior to m_t.

Notice that the *consistent delivery* property does not *necessarily* follow from the *reliable delivery* and *total order* properties. Consider the case where some node i receives messages m_1, m_2, and m_3; another node j that leaves the focal point region after receiving message m_3 may receive only messages m_1 and m_3: "Reliable Delivery" ensures that there exists some ϵ which bounds how long after time t' j has to remain in the focal point region in order to receive m_2; however, this ϵ may be quite large, and thus m_2 may not be delivered by the time j leaves the focal point after m_3 is delivered.

Discussion of Practical Issues. In terms of implementing the fpcast service in a real wireless network, the most notable property is that it guarantees totally-ordered

[5]Notice that if messages are not unique, then it becomes more difficult to define such a total ordering property.

[6]There is no formal requirement that this total ordering relate in any way to the real time order in which the messages were sent, though in practice it is likely that messages will be delivered in an order quite similar to that in which they are sent. In fact, in Chapter 7 where we present an algorithm that implements this totally-ordered message delivery, the messages are ordered with respect to the time on the local clocks of the broadcasting node; since the clocks may be out-of-date by some time t_{upd}, the messages are not necessarily delivered in the order in which they are sent. However, any two messages sent greater than time t_{upd} apart *are* in fact delivered in the order that they are sent.

message delivery, a property that is not typically provided by most physical MAC layers. In a radio network, however, each message is typically received at almost the exact instant that it is broadcast, as radio waves propagate quite quickly. Therefore, under some circumstances, it may be reasonable to assume that each node in a wireless network receives messages in the same order. In a prototype implementation of a virtual infrastructure [14], the only messages that typically arrived out of order at some node i were those very messages sent by i: the loopback mechanism resulted in i receiving its own messages earlier than other nodes.

Unfortunately most MAC layer protocols—the low-level implementation of wireless broadcast—do not simply broadcast a message once and assume that it is delivered successfully. In order to avoid collisions and compensate for lost messages, a single message broadcast can involve multiple wireless transmissions. Thus the very techniques that are typically used to ensure reliability—that is, backoff protocols—disrupt the ordering of messages and potentially result in different nodes receiving messages in different orders.

One solution is to avoid backoff-related techniques for ensuring reliability, instead depending on an alternative strategy for avoiding collisions. For example, the broadcast service might use two different (and non-conflicting) radio channels, instead of only one. The first channel, the *reservation channel*, might be used when joining the focal point. The second channel, the *communication channel*, might operate using a time-division/multiple access (TDMA) protocol that allocates each node a time slot in which to broadcast. When joining the focal point on the reservation channel, a mobile node might compete for a time slot in the TDMA schedule. This would eliminate collisions on the main communication channel, while ensuring that messages are delivered to each node in the exact order specified by the TDMA schedule.

In Chapter 7, as part of discussing how to implement virtual infrastructure using only basic local communication, we (briefly) present an alternative solution for implementing the fpcast service. We first introduce an alternative model for wireless broadcast, the lbcast service, that more closely resembles the typical behavior of wireless radio broadcast: each message is eventually delivered in a timely fashion to every node within some radius R. In practice, such a service could be implemented using simple backoff protocols, with a small number of retries to avoid collisions. We then show how to use the lbcast service to implement fpcast. This construction uses techniques originally introduced by Lamport in [59] to ensure that messages are delivered in the same order to each node.

2.3.2 GeoCast Service

The mobile nodes also have access to a global message delivery service, GeoCast. The GeoCast service delivers a message to every node within a certain radius R of some specified destination in the plane. Formally, the GeoCast service is parameterized by some constant R which determines the size of the destination region.

For the first part of this thesis, we fix a particular R_{geo}. The constant R_{geo} is chosen to be larger than the radius of the largest focal point. As a result, a GeoCast message directed at the center of a focal point will be delivered to every mobile node

in that focal point. When it is clear from context, we refer to the R_{geo}-GeoCast service as simply the GeoCast service.

The GeoCast service satisfies three properties: (1) *integrity*, meaning that every message delivered was previously broadcast, (2) *reliable delivery*, meaning that if a message is geocast to a specific region, and if some node resides in that destination region for sufficiently long, then it will receive the message; and (3) *minimum delivery time*, meaning that every message is delivered at least time t_{upd} after it is sent.

The *reliable delivery* property posits that for every geocast event, there exists some delivery time t' such that every correct mobile node that is in the delivery region at time t' and remains with the delivery region for sufficiently long after t' receives the message. This remains an asynchronous property, however, as there is no bound on how long the delivery may take.

The *minimum delivery time* ensures that messages are not delivered, for example, in zero time. This ensure that some every time two mobile nodes participate in a round-trip message exchange, some time passes. In particular, enough time passes that a response is received at a time strictly greater than when the initial message was sent, according to the sender's local clock. In particular, this implies that the local clock of a mobile node has sufficient resolution.

Formal Definition. For mobile node i, for $R \in \mathbb{R}$, the R-GeoCast service supports the following two actions:

- Input $\mathsf{geocast}(m, d)_i$
- Output $\mathsf{geocast\text{-}rcv}(m, d)_i$

where m is an arbitrary message to be sent and $d \in L$ is the destination location. The GeoCast service has the following properties:

***Integrity*:** For any GeoCast message m, location d, and node i, if a $\mathsf{geocast\text{-}rcv}(m, d)_i$ event occurs, then a $\mathsf{geocast}(m, d)_j$ event precedes it, for some node j.

***Reliable Delivery*:** For every $\mathsf{geocast}(m, d)_i$ event that occurs at time t where $i \in I$, there exists some $t' > t + t_{\mathsf{upd}}$ and $\epsilon \geq 0$ such that: assume that $j \in I$ is a mobile node satisfying the following:

- mobile node j is within distance R of location d at time t';
- mobile node j remains within distance R of location d until time $t' + \epsilon$; and
- mobile node j does not fail prior to time $t' + \epsilon$;

then a $\mathsf{geocast\text{-}rcv}(m, d)_j$ event occurs at some point in the interval $[t', t' + \epsilon]$, delivering the message to node j.

***Minimum Delivery Time*:** Each message takes some time $> t_{\mathsf{upd}}$ to be delivered.

Notice that the *reliable delivery* property allows each message to have a different ϵ governing its delivery time. Thus, in an infinite execution, this definition allows for unbounded message delivery times. When discussion the performance of algorithms, we will make further assumptions, including placing an upper bound on ϵ. (See Section 2.4.)

Discussion of Practical Issues. There has been a significant amount of research on implementing GeoCast services in wireless ad hoc networks, and GeoCast is a common communication service in mobile networks: a number of algorithms have been developed to solve this problem, originally for the internet protocol [78] and later for ad hoc networks (e.g., [17, 50]). GeoCast services have been used in routing algorithms (e.g., [44, 54]), tracking algorithms (e.g., [2]), and data storage algorithms (e.g., [65,91]). GeoCast can be implemented inefficiently via simple flooding protocols; more efficient implementations reduce the message traffic by attempting to find direct (geographic) routes from the source to the destination.

2.4 Liveness, Performance and Synchrony

For the purpose of proving liveness properties and analyzing performance, we will consider a restricted set of executions that guarantee certain synchrony assumptions. Specifically, we assume the following:

- Any enabled action at a mobile node is executed immediately with no real time elapsing.
- For every fpcast event, the ϵ specified by the "Reliable Delivery" property is bounded by d_{fp}. This implies that every message broadcast using the fpcast service is delivered within time d_{fp}.
- For every geocast event, the ϵ specified by the "Reliable Delivery" property is bounded by ϵ_{geo}.
- Every message broadcast using the geocast service is delivered within time d_{geo}.[7]

These assumptions are discussed in more detail in Sections 4.3 and 5.3.

[7]Notice that this bound on delivery time is not implied by the bound on ϵ for the GeoCast service, as the reliable delivery property states that every message is delivered with ϵ after some time t'; the delivery time is thus at most $(t' - t) + \epsilon_{\mathsf{geo}}$.

Notation:

- I, totally-ordered set of *node identifiers*
- $i_0 \in I$, a distinguished node identifier in I that is smaller than all other identifiers in I
- S, set of *port identifiers*, defined as $\mathbb{N}^{>0} \times OP \times I$, where OP is a set of operation identifiers.
- O, the totally-ordered, finite set of *focal point identifiers*
- T, set of *tags*, defined as $\mathbb{R}^{\geq 0} \times I$
- U, set of *operation identifiers*, defined as $\mathbb{R}^{\geq 0} \times S$
- X, set of *memory locations*
- For each $x \in X$:
  - V_x, the set of *values* for x
  - $v_{0,x} \in V_x$, the initial value of x
- M, a totally-ordered set of *configuration names*
- $c_0 \in M$, a distinguished configuration in M that is smaller than all other names in M
- C, totally-ordered set of *configuration identifiers*, defined as $\mathbb{R}^{\geq 0} \times I \times M$
- L, set of locations in the plane , defined as $\mathbb{R} \times \mathbb{R}$
- $t_{\mathsf{upd}} \in \mathbb{R}$, the frequency with which each mobile node receive time and location updates.
- $t_{\mathsf{join}} \in \mathbb{R}$, the time that a new mobile node needs to enter a focal point prior to the previous mobile node leaving in order for the focal point to remain populated.
- v_{max}, the maximum velocity of any mobile node.
- R_{FP}, the radius of the largest focal point.
- R_{geo}, the radius of the GeoCast service. $R_{\mathsf{geo}} > R_{\mathsf{FP}}$.
- R_{Vgeo}, the radius of the virtual GeoCast service. $R_{\mathsf{Vgeo}} = R_{\mathsf{geo}} - R_{\mathsf{FP}}$.
- R_{lb}, the radius of the Local Broadcast service.

Figure 2-1: Notation used throughout Part I of this thesis.

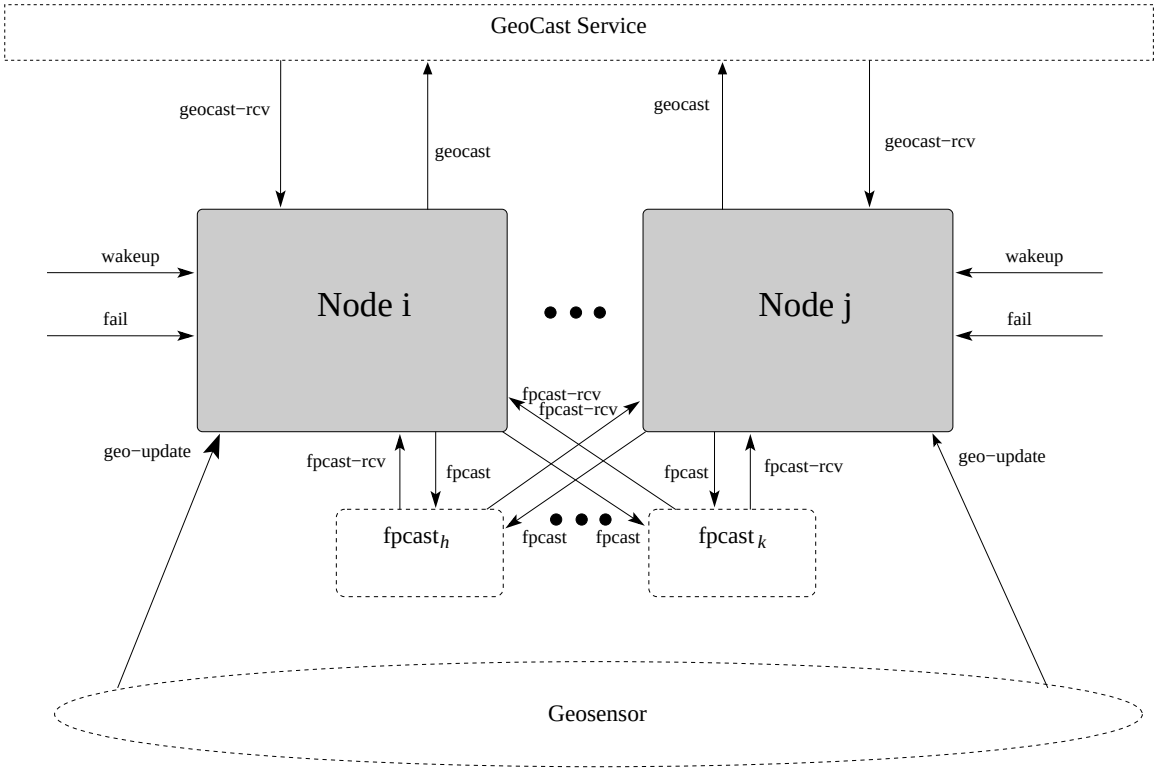

Figure 2-2: Architecture of the theoretical system model. The shaded boxes represent two mobile nodes i and j from the set I. Each mobile node is an asynchronous I/O automaton, which is a special case of a timed I/O automaton. Everything inside the shaded area represents programs running on the mobile nodes. Everything outside the shaded area is part of the RealWorld automaton, which represents the physical world in which the other automata operation; this includes the GeoCast service, multiple fpcast services (one per focal point), and the Geosensor. (Dashed lines indicate subcomponents of the RealWorld automaton.) The focal point identifiers h and k are from the set O. The wakeup and fail events are initiated by the RealWorld automaton.

Chapter 3

Virtual Infrastructure

In this chapter, we introduce two types of virtual infrastructure: an infrastructure based on *virtual objects* and an infrastructure based on *virtual nodes*.

The first of these abstractions is the **Virtual Object Layer**.[1] The Virtual Object Layer is a shared object model: clients (i.e., the mobile nodes) can perform operations on virtual objects that reside at pre-determined, potentially changing, locations in the network. From the perspective of a client, the virtual objects behave exactly as objects in a typical shared object model.

The second of these abstractions, the **Virtual Node Layer**, generalizes the Virtual Object Layer. Each virtual entity is similar to a mobile node in that it is an automaton that can broadcast and receive messages. Unlike a virtual object, a virtual node can initiate activity (i.e., messages and computation) directly, rather than simply responding to client requests. Also, virtual nodes can communicate with each other via a virtual GeoCast service. For example, virtual nodes can transport data from one end of a network to the other by transmitting it along a chain of virtual nodes. Clients and virtual nodes communicate with each other via a virtual GeoCast service.

We begin in Section 3.1 by introducing the Virtual Object Layer. We continue in Section 3.2 to introduce the Virtual Node Layer. Finally, in Section 3.3, we briefly discuss a few examples of applications that can be built using the Virtual Node Layer. A more detailed and extensive example using the Virtual Object Layer is described in Chapter 6.

3.1 Virtual Object Layer

The **Virtual Object Layer** is a simple shared-object model that hides the underlying dynamics of a mobile ad hoc networks. The Virtual Object Layer consists of three types of entities: **clients**, **virtual objects**, and a RealWorld automaton modelling the environment:

[1] This model was previously referred to as the Focal Point Object Model in [32, 33].

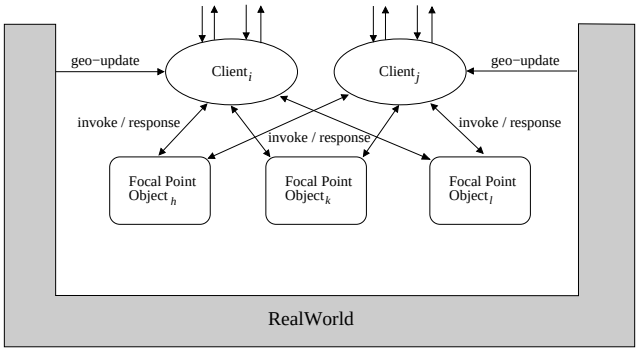

Figure 3-1: Virtual Object Layer. The clients interact with the virtual objects via invocations and responses.

- Each client is modelled as an asynchronous I/O automaton. The clients interact with the virtual objects.

- Each virtual object is specified by a variable type specification, and is also modelled as an asynchronous I/O automaton, specifically the *canonical automaton* that implements the specified variable type. Recall (see Appendix A for more details) that a variable type specification consists of (1) a set of values V, (2) some initial value $v_0 \in V$, (3) a set of invocations I, (4) a set of responses R, and (5) a deterministic transition function $f : I \times V \to R \times V$, that is, mapping an invocation and a state to a response and a new state. Given such a type specification, the associated canonical automaton is a trivial atomic implementation of the shared object with the semantics specified by the type specification. (See Appendix A for more details.) Virtual objects receive invocations from the clients and return responses generated by the transition function based on the invocation and their current state.

- The RealWorld automaton models the (virtual) environment and is a timed I/O automaton. The RealWorld maintains the location of each client, along with the real time, and delivers geo-updates to the clients. Updates occur at least every time t_{upd}. Since an asynchronous I/O automaton is a special case of a timed I/O automaton, and hence it is possible to composed the untimed clients with the timed RealWorld.

The Virtual Object Layer is presented schematically in Figure 3-1, where a set of clients interacts with a set of virtual objects.

Notice that the clients and virtual objects are modelled as asynchronous automata, although they receive updates from the RealWorld that contain timing information. In Section 3.1.2 we introduce further timing assumptions that are used when analyzing the liveness and performance of the Virtual Object Layer.

Client / Virtual Object Interactions. The clients communicate only through their interactions with the shared objects; there is no message-passing network. Each client has a (countably) infinite number of invocation/response ports onto each shared object, allowing it to invoke concurrent operations on an object. That is, the port set Q of each object is $S = \mathbb{N}^{>0} \times OP \times I$, where OP is a set of operations (for example, read, write, enqueue, dequeue, etc.). As is usual in shared object models, the clients can invoke only one operation at a time on each port of an object.

Virtual Objects and Focal Points. Each virtual object is associated with a focal point in the set O. At any given time, a virtual object resides at the center of its associated focal point. Specifically, if a virtual object is associated with focal point h, then at time t, the virtual object resides at the location $\mathsf{fp\text{-}center}_h(t)$.

If the function $\mathsf{fp\text{-}center}_h$ is constant over time, then the virtual object associated with h remains in the same location throughout an execution. If both functions $\mathsf{fp\text{-}center}_h$ and $\mathsf{fp\text{-}region}_h$ are constant over time, we say that virtual object h is **static**.

If every virtual object $h \in O$ is static, then we refer to this case as the **Static Virtual Object Layer**. Otherwise, we refer to the general case as the **Dynamic Virtual Object Layer**[2].

3.1.1 Virtual Object Failures

The virtual objects are fault prone. That is, a virtual object can fail during an execution, at which point it ceases to respond to invocations. The failure or correctness of a virtual object depends only on the client population of its associated focal point. If a focal point region is populated throughout an execution, then its associated virtual object is correct. If, on the other hand, a focal point region becomes depopulated, then the virtual object associated with that focal point may fail. Note that it does not matter how a focal point region becomes depopulated, be it as a result of clients failing, leaving the area, etc. Any depopulation results in the virtual object failing. We do not consider the possibility of a virtual object recovering; the same recovery definitions and techniques discussed in the context of a virtual node (see Section 3.2.3) can be applied to a virtual object.

A focal point is **populated throughout** a finite execution fragment if for some sufficiently large constant t_{join} there exists a sequence, $j_0, \ldots, j_k$ of clients such that:

- Client j_0 is well inside the focal point region when the execution fragment begins, and remains well inside the focal point region until at least time t_{join}. That is, the location of node j_0 at the beginning of the execution fragment through time t_{join} is within the inner region specified for the focal point, i.e., its location satisfies Equation 2.1. We refer to the time at which j_0 first leaves the inner region as the **departure time** for j_0.

- For every $\ell < k$, client $j_{\ell+1}$ is well inside the focal point region at least t_{join} prior to the departure time for j_ℓ, and remains well inside the focal point region and non-failed until at least time t_{join} after the departure time for j_ℓ. We refer to the first time after the departure time of j_ℓ at which $j_{\ell+1}$ leaves the inner region as the **departure time** of $j_{\ell+1}$. The constant t_{join} will be specified later in Sections 4.3 and 5.3 when analyzing the performance of the Virtual Object Layer[3].

[2]Notice that the location of a virtual object is, in many ways, irrelevant to the clients that are using these objects: the objects behave in the same manner regardless of their location. However, the location of a virtual object—as determined by its associated focal point—affects its correctness, since it remains correct only if its associated focal point region is populated. Thus it may be desirable for a virtual object to inhabit regions that are more populated. The location of a virtual object may also affect the cost of accessing it, if one were to account for the cost of a long-distance geocast; we do not examine this latter metric, however.

[3]Notice that this definition requires a mobile node to be well inside a focal point region at least time t_{join} prior to the earlier node leaving. As we have not yet made any liveness assumptions about the underlying physical system, we cannot at this point quantify t_{join}; intuitively, this time interval must be long enough for the focal point join protocol to complete and for any messages in the process of being delivered to arrive. In Sections 4.3 and 5.3, where we analyze the performance of one of our protocols, we quantify more precisely the size of t_{join}.

- Client j_k is well inside the focal point region at the end of the execution fragment.

A focal point is *populated throughout* an infinite execution fragment if it is populated for all finite prefixes of the execution fragment. Thus a virtual object is **correct** in some execution if its associated focal point is populated throughout that execution.

3.1.2 Timing and Liveness

In this thesis, we consider primarily an asynchronous variant of this model, that is, there is no upper bound on how long an operation at a virtual object might take, and there is no bound on the rate at which a client might take steps.

The Virtual Object Layer does, however, guarantee a lower bound on the duration of an operation at a virtual object: each operation takes some finite time $> t_{\mathsf{upd}}$. This technical assumption proves useful in building applications using the Virtual Object Layer, as it implies that the updates provided by the RealWorld automaton are sufficiently frequent to distinguish when an operation begins from when an operation ends.

For the sake of analyzing performance, however, we will consider a synchronous variant of the Virtual Object Layer in which:

- *Client responsiveness:* Any enabled action at a client is executed immediately.
- *Virtual object responsiveness:* If a correct client i at some location ℓ invokes an operation on a correct focal point h and remains forever within distance R_{geo} of location ℓ, then eventually i receives a response from h. In particular, there is an upper bound on how long i waits to receive a response.

This describes a special case of a timed I/O automaton (similar to the model presented in [75]) in which certain tasks have upper bounds associated with them.

Simple Example

As a very simple toy example of an algorithm that uses the Virtual Object Layer, consider the problem of implementing an unreliable atomic read/write memory in the Virtual Object Layer. In Chapter 6, we provide a more complete and involved example, involving the implementation of a *reliable* atomic read/write memory. In this simple example, we use only a single virtual object, $h \in O$, with operations $OP = \{\mathsf{read}, \mathsf{write}\}$. The virtual object implements the read/write memory variable type (presented, for reference, in Figure A-1). In order to perform read and write operations, the client simply invokes the appropriate operation on the virtual object. For example, when client i wants to satisfy a read request, it simply invokes the read operation on the virtual object. It does this by performing the following transition: $\mathsf{invoke}(\langle \mathsf{read} \rangle)_p$ where port $p = \langle 1, \mathsf{read}, i \rangle$, a port for client i on object h. Eventually, the virtual object responds as follows: $\mathsf{respond}(\langle \mathsf{read\text{-}ack}, val \rangle)_p$. At this point, the client i can return the value. A write operation proceeds similarly, invoking the write operation on an appropriate port of the virtual object.

This simple algorithm solves the problem of implementing *unreliable* read/write atomic memory in the Virtual Object Layer. This algorithm is, effectively, a centralized solution that depends on a single virtual object. If the virtual object h fails (because, say, the focal point itself becomes depopulated), then the read/write memory itself fails. In Chapter 6, we describe a fault-tolerant algorithm for read/write atomic memory in the Virtual Object Layer.

3.2 Virtual Node Layer

The **Virtual Node Layer** is a natural generalization of the Virtual Object Layer. It is quite similar to the Virtual Object Layer with the exception that (1) each virtual entity is a virtual node, rather than a virtual object, and (2) the entities—virtual and real—communicate using a (virtual) GeoCast service, rather than through an invocation-and-response mechanism. In addition, virtual nodes may recover, should they fail.

The Virtual Node Layer consists of three types of entities: **clients**, **virtual nodes**, and a RealWorld automaton modelling the environment:

- Each client is modelled as an asynchronous I/O automaton.
- Each virtual node is modelled a deterministic, asynchronous I/O automaton.
- The RealWorld automaton models the (virtual) environment and is a timed I/O automaton. The RealWorld maintains the location of each client, along with the real time, and delivers geo-updates to the clients. (Notice that it does *not* deliver such updates to the virtual nodes.[4]) Updates occur at least every time t_{upd}. The RealWorld automaton also supports a virtual GeoCast broadcast service, which allows clients and virtual nodes to geocast messages to each other. This service is discussed further in Section 3.2.4.

Notice that the clients and virtual nodes are modelled as asynchronous automata, although they receive updates from the RealWorld that contain timing information. In Section 3.2.5 we introduce further liveness and timing assumptions that are used when analyzing the liveness and performance of the Virtual Node Layer.

3.2.1 Virtual Nodes and Focal Points

Each virtual node is associated with a focal point in the set O. At any given time, a virtual node resides at the center of its associated focal point. Specifically, if a

[4]In practice, it is often useful for the virtual nodes to be aware of time and location information. Modelling the virtual nodes as *timed* automata introduces significant complications into both the virtual layer definition and into emulation algorithms. See [82] for an example of how to accomplish this. In order to achieve some of the benefits of a timed model, a simple application-level technique can be used: nearby clients can be programmed to send time and location information to the virtual nodes. This technique is a simple way of providing the virtual nodes with near-up-to-date information, and was first introduced in [69].

virtual node is associated with focal point h, then at time t, the virtual node resides at the location $\mathsf{fp\text{-}center}_h(t)$. If the function $\mathsf{fp\text{-}center}_h$ is constant over time, then the virtual node associated with h remains in the same location throughout an execution. If both functions $\mathsf{fp\text{-}center}_h$ and $\mathsf{fp\text{-}region}_h$ are constant over time, we say that virtual node h is **static**. If every virtual node $h \in O$ is static, then we refer to this case as the **Static Virtual Node Layer**. Otherwise, we refer to the general case as the **Dynamic Virtual Node Layer**.

3.2.2 Motion-Controlled Clients

Throughout this section, we have assumed that the clients have no control over their own motion. In some cases, however, we may be interested in considering a Virtual Node Layer in which the clients can direct their own motion. As in the case of the mobile nodes (see Section 2.2.3), it is a simple modification to support clients that execute an algorithm to control the clients motion. Specifically, each client is assumed to produce (as an output) some motion control signal, for example, $\mathsf{move}(\ldots)$. This signal may be in the form of a velocity vector, a target waypoint, or any other desired form of motion control signal. In practice, the most common such signal is acceleration. The RealWorld automaton, as a model of the environment, receives such signals as input and uses them to update the location of the clients.

3.2.3 Virtual Node Failures

The virtual nodes are fault prone, but may recover. The failure and recovery of a virtual node depends only on the client population of its associated focal point. If a focal point region is populated throughout an execution fragment, then its associated virtual node is correct—after some initial interval of time. If, on the other hand, a focal point region becomes depopulated, then the virtual node associated with that focal point may fail. Note that it does not matter how a focal point region becomes depopulated, be it as a result of clients failing, leaving the area, etc. Any depopulation results in the virtual node failing. When the focal point region is re-populated, then after some sufficient interval of time, the virtual node recovers.

A focal point is **populated throughout** a finite execution fragment if for some sufficiently large constant t_{join} there exists a sequence, $j_0, \ldots, j_k$ of clients such that:

- Client j_0 is well inside the focal point region when the execution fragment begins, and remains well inside the focal point region until at least time t_{join}. That is, the location of node j_0 at the beginning of the execution fragment through time $3t_{\mathsf{join}}$ is within the inner region specified for the focal point, i.e., its location satisfies Equation 2.1. We refer to the time at which j_0 first leaves the inner region as the **departure time** for j_0.

- For every $\ell < k$, client $j_{\ell+1}$ is well inside the focal point region at least t_{join} prior to the departure time for j_ℓ, and remains well inside the focal point region and non-failed until at least time t_{join} after the departure time for j_ℓ. We refer to the first time after the departure time of j_ℓ at which $j_{\ell+1}$ leaves the inner

region as the **departure time** of $j_{\ell+1}$. The constant t_{join} will be specified later in Sections 4.3 and 5.3 when analyzing the performance of the Virtual Node Layer[5].

- Client j_k is well inside the focal point region at the end of the execution fragment.

A focal point is *populated throughout* an infinite execution fragment if it is populated for all finite prefixes of the execution fragment.

We say that a virtual node is *correct* in some execution fragment if its associated focal point is populated throughout that fragment, and either (1) the fragment in question is the very beginning of the execution, or (2) there is sufficient time prior to the fragment for the virtual node to recover during which the focal point is populated. Formally, for every fragment α' of a timed execution α of the Virtual Node Layer, we say that a virtual node $h \in O$ is **correct** during α' if focal point h is populated throughout α' and either:

- fragment α' is a prefix of α; or
- there exists a fragment α'' of temporal length at least $2t_{\mathsf{join}}$ that precedes α' in α (i.e., $\alpha''.\alpha'$ is a fragment of α) and h is populated throughout $\alpha''.\alpha'$.

3.2.4 The RealWorld and Virtual GeoCast

The clients and virtual nodes communicate using a GeoCast service, which is formally part of the virtual RealWorld automaton that models the virtual environment. The virtual RealWorld automaton is similar to that of the underlying system, except that (1) it accounts for the failure status of clients and virtual nodes, instead of mobile nodes, (2) it accounts for the location of clients, instead of mobile nodes, and (3) it contains a virtual GeoCast service as a sub-component, but no fpcast services.

Each client and virtual node can use the virtual GeoCast service to send message to all the clients and virtual nodes within some radius R of a chosen destination location. To distinguish this broadcast service from the GeoCast service in the underlying physical network, we refer to it as the **virtual GeoCast** service; when a node sends a message, we say that it performs a virtual-geocast(m, d), sending message m to destination d; when a node receives a messages, we say that it performs a virtual-geocast-rcv(m, d).

For technical reasons, it is useful in this case to specify an automaton for the RealWorld that captures the safety properties of the environment; this will be useful when showing that a protocol satisfies the safety properties of the Virtual Node Layer.

[5]Notice that this definition requires a mobile node to join a focal point at least time t_{join} prior to the earlier node leaving. As we have not yet made any liveness assumptions about the underlying physical system, we cannot at this point quantify t_{join}; intuitively, this time interval must be long enough for the focal point join protocol to complete and for any messages in the process of being delivered to arrive. In Sections 4.3 and 5.3, where we analyze the performance of one of our protocols, we quantify more precisely the size of t_{join}.

(See Section 5.2.5.) In the case of the Virtual Object Layer, this was unnecessary as the virtual objects had no interaction with the RealWorld automaton. In this case, however, the clients and virtual nodes communicate via the virtual GeoCast service, which is part of the RealWorld. Hence when showing that an algorithm correctly emulates the Virtual Node Layer, it is useful to be able to discuss more precisely the behavior of the RealWorld. We describe the RealWorld automaton by specifying the safety properties that is exactly satisfies, and arguing that they are easily captured in an automaton.

The virtual RealWorld automaton maintains in its state the following components:

- the current real time (a non-negative real);
- the location of each client (a mapping from I to L);
- the time at which the last geo-update occurred for each client (a mapping from I to non-negative reals); this allows the RealWorld to ensure that updates are delivered with sufficient frequency;
- a set consisting of every message m broadcast by the virtual GeoCast service;
- a set $A \subseteq I \cup O$ consisting of all non-failed clients and virtual nodes.

Along with the virtual GeoCast interface—virtual-geocast and virtual-geocast-rcv—it has an output fail action which causes a client or virtual node to fail. The automaton is defined so that its trace set consists of exactly those traces γ satisfying the following properties:

1. In trace γ, for each non-failed client i, a geo-update event occurs at least every t_{upd} time.
2. Trace γ satisfies the integrity property (as defined in Chapter 2 in the context of the GeoCast service):

Integrity: For any virtual GeoCast message m, location d, and client or virtual node $i \in I \cup O$, if a virtual-geocast-rcv$(m, d)_i$ event occurs in an execution, then a virtual-geocast$(m, d)_j$ event precedes it, for some client or virtual node $j \in I \cup O$.

Notice that it easy to define transitions that enforce exactly these two properties: the only condition on the virtual-geocast-rcv is that the message to be delivered exists in the set of previously broadcast messages; the only restriction on trajectories is that each client receives geo-updates with sufficient frequency.

The virtual GeoCast service also satisfies a liveness property: *reliable delivery* (as is defined also in Chapter 2 in the context of the GeoCast service):

> *Reliable Delivery*: For every virtual-geocast$(m, d)_i$ event that occurs at time t where $i \in I \cup O$, there exists some $t' > t + t_{\mathsf{upd}}$ and $\epsilon \geq 0$ such that: let $j \in I \cup O$ be a client or virtual node satisfying the following:

– j is within distance R of location d at time t';

– j remains within distance R of location d until time $t' + \epsilon$; and

– j is correct for the interval $[t', t' + \epsilon]$;

then a $\mathsf{geocast\text{-}rcv}(m, d)_j$ event occurs at some point in the interval $[t', t' + \epsilon]$, delivering the message to j.

This property is treated as a restriction on the executions of interest in the virtual model. In particular, we will show that the Virtual Node Layer, when implemented using the algorithm in Chapter 5, guarantees reliable delivery (along with some specific performance metrics) when the underlying physical model guarantees additional liveness and timing properties.

Recall that the underlying GeoCast service is parameterized by some delivery radius R_{geo}, meaning that messages are delivered to every mobile node within distance R_{geo} of the specified destination. We now specify the delivery radius for the *virtual* GeoCast service. Let R_{FP} be the maximum radius of any focal point. Recall (from Section 2.3.2) that the radius R_{geo} of the GeoCast service in the underlying mobile network is $> R_{\mathsf{FP}}$. Fix $R_{\mathsf{Vgeo}} = R_{\mathsf{geo}} - R_{\mathsf{FP}}$.

This choice of R_{Vgeo} allows us to relate the behavior of the underlying GeoCast service in the mobile network to the virtual GeoCast service in the Virtual Node Layer. For example, assume that some mobile node i GeoCasts a message m to some location d. We know, by assumption, that every mobile node within distance R_{geo} of d receives message m. In the Virtual Node Layer, we also want every virtual node within distance R_{Vgeo} of location d to receive message m. In fact, if some focal point center ℓ is within distance R_{Vgeo} of location d, then every mobile node in the focal point region receives message m. That is, we claim the following:

Lemma 3.2.1. *For every* $\mathsf{geocast}(m, d)_i$ *event that occurs at time* t *where* $i \in I$, *there exists some* $t' > t + t_{\mathsf{upd}}$ *and* $\epsilon \geq 0$ *such that: assume that* $h \in O$ *is a focal point and* $j \in I$ *is a mobile node that satisfy the following:*

- *the focal point center of* h *is within distance* R_{Vgeo} *of* d *at time* t',
- *the focal point center of* h *remains within distance* R_{Vgeo} *of* d *through time* $t' + \epsilon$,
- *mobile node* j *is inside the focal point region of* h *at time* t',
- *mobile node* j *remains inside the focal point region of* h *through time* $t' + \epsilon$,
- *mobile node* j *does not fail prior to time* $t' + \epsilon$,

then a $\mathsf{geocast\text{-}rcv}(m, d)_j$ *event occurs at some point in the interval* $[t', t' + \epsilon]$, *delivering the message to node* j.

Proof. We need to show the following three conditions for j to satisfy the *Reliable Delivery* property of the underlying GeoCast service, which implies the desired result:

- Mobile node j is within distance R_{geo} of location d at time t': Since j is inside focal point h at time t', we know that j is within distance R_{FP} of the focal point center for h at time t'. Moreover, the focal point center of h is within distance R_{Vgeo} of d at time t'. Thus, by the triangle inequality, j is within distance $R_{\mathsf{geo}} \leq R_{\mathsf{Vgeo}} + R_{\mathsf{FP}}$ of d at time t.
- Mobile node j remains within distance R_{geo} of location d until time $t' + \epsilon$: This conclusion follows by the same argument, since j remains inside the focal point region for h through time $t' + \epsilon$, and since the focal point center for h remains within distance R_{Vgeo} of d through time $t' + \epsilon$.
- Mobile node j does not fail prior to time $t' + \epsilon$, by assumption.

□

3.2.5 Timing and Liveness

In this thesis we consider primarily an asynchronous variant of the Virtual Node Layer. That is, there is no upper bound on the rate at which clients and virtual nodes take steps, and there is no upper bound on the delay of the virtual GeoCast communication service.

For the sake of analyzing performance, however, we will consider a synchronous variant of the Virtual Node Layer in which:

- Any enabled action at a client is executed immediately.
- An enabled action at a virtual node is executed within a bounded time. That is, if at time t some action a is enabled at virtual node v, then after some bounded amount of time, either action a is executed, or at some point during that interval of time, some other action occurs such that action a is no longer enabled.
- The delay of the virtual-geocast/virtual-geocast-rcv communication service is bounded.
- For every execution, there is a bound on the ϵ specified in the reliable delivery property of the virtual GeoCast service.

This describes a special case of a timed I/O automaton (similar to the model presented in [75]) in which certain tasks have upper bounds associated with them.

3.3 Example Applications

In this section we briefly discuss some scenarios in which virtual nodes facilitate the design of algorithms. The algorithms described (briefly) in this section depend on using virtual nodes to collect, disseminate, and process information. We focus in particular on applications that involve *dynamic* virtual nodes, that is, virtual nodes

that travel through the network. In Chapter 6, we provide a complete example of a protocol that uses *static* virtual objects to reliably store data in a changing network.

These examples are, loosely, related to the problem of maintaining dynamic data structures in a mobile ad hoc network. For example, the first example in Section 3.3.1 is related to the problem of routing, which in most implementations attempts to maintain portions of a shortest-path tree. Due to the constantly changing environment, these algorithms are notoriously difficult to implement correctly and even harder to analyze: it is rare for such a dynamic algorithm to have provably good performance in the worst case. We suggest that instead of maintaining these data structures explicitly, it is easier to take advantage of the spatial topology of the mobile nodes and use virtual nodes to collect and process the data.

3.3.1 Routing

We first consider the problem of point-to-point message routing that delivers messages to specifically identified mobile nodes. Notice that the GeoCast service does not deliver messages to specified nodes; instead, it sends messages to a particular region of the network. Most routing algorithms (e.g., [42,84,85,93]) either track the location of every mobile node in the system, or flood the entire network with messages to discover the location of each node. Both approaches can be quite expensive, and optimizations are difficult.

We suggest instead a routing scheme based on the compulsory protocol of Chatzigiannakis et al. [20]. In its simplest version, a single virtual node travels through the network, collecting and delivering messages. In order to send a message, a mobile node i waits until it nears the virtual node, and then uses the geocast service to send a message to the virtual node's (predicted) location. A mobile node can determine that it is near to a virtual node by calculating the mobile node's location at a given time and comparing it to its own location. Since the mobile node receives time and location updates at least every t_{upd} time, this calculation is never too far off. The first time that the client receives a time update placing it within broadcast range of the virtual node, it transmits its messages. Underlying this idea is the assumption that a client will be within broadcast range of the virtual node for some time $\gg t_{\mathsf{upd}}$; this depends on the broadcast radius, the velocity of the client, and the velocity of the virtual node in question.

The virtual node collects messages that it has received, and waits to deliver them to clients. A client that wants to receive messages waits until the virtual node is nearby, and then requests from it any messages it has to deliver to that specific client.

Recall that virtual nodes fail in empty regions of the network. However, wherever there are nodes that need to send and receive messages, there is presumably enough network density to ensure that the virtual node does not fail. Notice, though, that this algorithm only works well when the populated regions of the network are "connected:" if a node in one populated region tries to send a message to a node in another populated region, but all paths between these two regions are sparsely populated, the virtual node cannot deliver the message.

If the virtual node is correct, i.e., never enters a depopulated region of the network, then we can conclude under certain conditions that messages are delivered. Specifically, under the timing assumptions described in Section 3.2.5, along with the assumption that, every so often, a client is within broadcast range of the virtual node for sufficiently long to detect its presence and exchange information with it, then we can conclude that every message will eventually be delivered. The time to deliver the message can be calculated based on (1) how long it takes for the sending client to approach the virtual node, and (2) how long it takes for the virtual node to reach the receiving node. If the virtual node sweeps through the network, then we can determine a bound on this time based on the size of the geographic region and the velocity of the virtual node.

Using more virtual nodes can shorten the average message latency, while increasing the overall cost of the protocol. For example, a set of virtual nodes may traverse the network in a covering fashion; whenever two virtual nodes pass each other, they send each other their stores of messages. In this way, all the messages spread to all the virtual nodes. A more space-efficient algorithm might use the scheme developed in [20], where the virtual nodes form a snake, winding through the network in a pseudorandom path, thus regularly visiting every populated region of the network and delivering messages to the resident nodes.

Compared to typical routing schemes (such as DSR and AODV), these algorithms are easier to tune, in terms of space versus latency trade-offs: by increasing the number of virtual nodes, and thus the space, messages are more rapidly delivered. Unlike DSR and AODV, the cost (in terms of message latency and space usage) of these algorithms scales with the number of virtual nodes and the size of the region being covered, rather than the number of nodes. On the other hand, using a Virtual Node Layer has larger associated overhead costs, as it requires emulating a virtual layer. It remains an interesting experimental question whether it is feasible to implement routing on a Virtual Node Layer in such a way that it is competitive with existing routing protocols. (See [97] for ongoing progress in this direction.)

3.3.2 Data Collection

A common use of ad hoc networks is to monitor environmental sensors. For example, one might wish to evaluate the average temperature, the average remaining battery charge, or the number of nodes in various regions of the network. (The latter application might be used to track animals or cars, or to determine the density of mobile nodes for other uses.)

We propose a very simple algorithm for collecting data in mobile (rather than static) networks. As in the case of routing, we assign the main work to the virtual nodes, which systematically explore the region in question, collecting and aggregating data. The primary difference from the routing algorithm is that the data may be aggregated, both regionally and temporally, as it is being collected. A client can specify as part of a query what sort of data the sensors should produce, and how the "network" should aggregate that data. The virtual nodes return only the necessary data to the clients. The rate of data collection can be calculated, as in the case of

routing, by considering the velocity of the virtual node and the size of the geographic region, under a similar set of performance and liveness assumptions. As in the case of routing, using more virtual nodes improves the performance, at the cost of increased communication.

Chapter 4

The Virtual Object Emulator

In this chapter we present a protocol that implements the Virtual Object Layer. The main goal of this chapter, then, is to show how to reliably emulate virtual objects using unreliable mobile nodes. The protocol for emulating a virtual object is based on the replicated-state-machine approach, in which all the mobile nodes in a focal point region cooperate to emulate the associated virtual object. Clients, by contrast, are emulated directly by the mobile nodes. The replicas for each virtual object coordinate using the fpcast service associated with the virtual object's focal point; this broadcast service provides for reliable, totally ordered broadcast. The standard replicated-state-machine techniques are extended to tolerate nodes joining and leaving the emulation. The result is a virtual object that remains reliable as long as the associated focal point region remains "populated" with clients; if the focal point region becomes depopulated, then the associated virtual object fails.[1]

In Section 4.1 we present an overview of the Virtual Object Emulator, the algorithm which implements the Virtual Object Layer. In Section 4.2 we argue that the emulation is correct. In Section 4.3 we discuss the performance of the Virtual Object Emulator.

An extended abstract of this work appeared in the 17th International Symposium on Distributed Computing (DISC 2003) [32], and a full version appeared in Distributed Computing [33]. The version included here includes some minor edits and improvements from the version in [33]. The protocol has been modified to tolerate *dynamic* virtual objects, and new names have been chosen for various entities and protocols. Specifically, the objects implemented are now referred to as virtual objects (rather than "focal point objects"), and the main emulation protocol is now referred to as the Virtual Object Emulator (rather than the "focal point emulator").

[1]Note that it does not matter how a focal point region becomes depopulated, be it as a result of mobile nodes failing, leaving the area, going to sleep, etc. Any depopulation results in the virtual object failing.

4.1 Virtual Object Emulator

In this section we present the **Virtual Object Emulator** (VOE), an algorithm that implements the Virtual Object Layer described in Chapter 3. We focus in this section on describing how the emulator implements a single virtual object associated with a single focal point h. Fix $h \in O$ for the remainder of this section. The entire layer is emulated simply by executing one instance of this protocol for each $h \in O$.

The basic idea of the Virtual Object Emulator is to use mobile nodes in the focal point region as replicas. Throughout an execution, mobile nodes move in and out of the focal point region. The nodes that are inside the focal point region at any given time act as replicas, and collaborate to implement the virtual object. They take advantage of the fpcast_h service to implement a replicated state machine that tolerates nodes continually joining and leaving. This replicated state machine consistently maintains the state of the atomic object, ensuring that the invocations are performed in a consistent order at all mobile nodes.

The Virtual Object Emulator consists of two automata that run on each mobile node. The VOE is designed to emulate a virtual object; it is not involved in emulating the client. The emulator consists of two components: the **VOE-Client** and the **VOE-Server**, which communicate with each other using the GeoCast service. Notice that both components of the emulator, though executing on the same mobile node, interact only via the GeoCast service. For each virtual object $h' \in O$ we assume a distinct instantiation of both Virtual Object Emulator components; recall that we are describing here the implementation of a single virtual object h.

Figure 4-1 depicts the various components, along with the two broadcast services $\mathsf{GeoCast}$ and fpcast_h. The clients (not depicted) send invocations to the VOE-Clients (on the left), which attach a tag to every request and broadcast the request to a focal point using the GeoCast service. Within the focal point, the request is received by the VOE-Servers. The VOE-Servers then coordinate amongst themselves to determine an ordering of the requests. They accomplish this coordination using the fpcast_h service, which delivers messages to all of the VOE-Servers in the same order. This total ordering determines the order in which the VOE-Servers process the requests. Based on this (ordered) sequence of requests, the VOE-Servers consistently update their local replicas of the atomic object, and send a response back to the originating VOE-Client using the $\mathsf{GeoCast}$ service. The VOE-Client then removes duplicates, and delivers the response to the client.

The VOE-Client runs on every mobile node that wants to access atomic object h; the VOE-Server runs on every mobile node and is active when the mobile node is well inside[2] the focal point region corresponding to the atomic object. We now proceed to describe these two components in more detail.

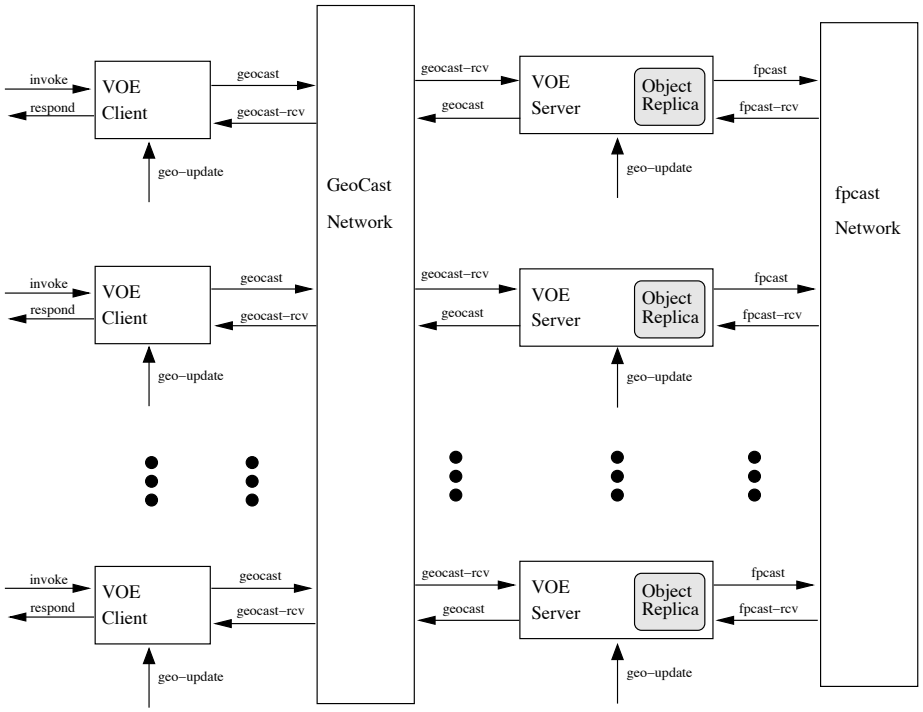

Figure 4-1: Diagram of the Virtual Object Emulator for emulating a single virtual object h. Each of the VOE-Clients and VOE-servers may execute on a different mobile node. When a VOE-Client (on the left) receives an invocation, it sends the request to all the VOE-Servers using the GeoCast service. At least one of the VOE-Servers broadcasts the request to all the other VOE-Servers in the focal point, using the fpcast_h service. Each VOE-Server then updates its object replica as a result of the request. At this point, at least one VOE Server sends a response to the originating VOE-Client, using the GeoCast service. The VOE-Client then filters out extraneous responses and delivers a response.

Figure 4-2: Automaton VOE-Client: Signature and State
for client i and object obj of variable type $\tau = \langle V, v_0, invocations, responses, \delta \rangle$

Input:
  $\mathsf{invoke}(inv)_p$, $inv \in invocations$, $p \in Q$
  $\mathsf{geocast\text{-}rcv}(\langle \mathsf{response}, resp, oid, loc \rangle, d)_i$, $resp \in responses$, $oid \in U$, $d \in L$
  $\mathsf{geo\text{-}update}(l, t)_i$, $l \in L$, $t \in R^{>0}$

Output:
  $\mathsf{geocast}(m, d)_i$, $m \in \mathsf{invoke} \times invocations \times U \times L \times L$, $d \in L$
  $\mathsf{respond}(resp)_p$, $resp \in responses$, $p \in Q$

Constants:
  $fp\text{-}center : \mathbb{R}^{\wedge}\{\geq 0\} \rightarrow L$, a function mapping time to the focal point center
  $fp\text{-}region : \mathbb{R}^{\wedge}\{\geq 0\} \rightarrow \mathcal{P}(L)$, a function mapping time to the focal point region

State:
  $clock \in R^{\geq 0}$, the current time, initially 0, updated by the geosensor
  $location \in L$, node i's location, initially i's initial location, updated by the geosensor
  $ready\text{-}responses \subseteq Q \times responses$, a set of an operation responses, initially $\emptyset$
  geocast-queue, a queue of messages to be geocast, initially $\emptyset$
  $ongoing\text{-}oids \subseteq U$, a set of operation identifiers, initially $\emptyset$

Figure 4-3: Automaton VOE-Client: Transitions
for client i and object obj of variable type $\tau = \langle V, v_0, invocations, responses, \delta \rangle$

```
1   Input invoke(inv)_p
2   Effect:
3       new-oid ← ⟨clock, p⟩
4       Enqueue(geocast-queue, ⟨⟨invoke, inv, new-oid, location⟩, fp-center(clock)⟩)
5       ongoing-oids ← ongoing-oids ∪ {new-oid}
6
7   Input geocast-rcv(⟨response, resp, oid, loc⟩, d)_i
8   Effect:
9       if (oid ∈ ongoing-oids) then
10          ⟨c, p⟩ ← oid
11          ready-responses ← ready-responses ∪ {⟨p, resp⟩}
12          ongoing-oids ← ongoing-oids − {oid}
13
14  Input geo-update(l, t)_i
15  Effect:
16      location ← l
17      clock ← t
18
19  Output geocast(m, d)_i
20  Precondition:
21      Peek(geocast-queue) = ⟨m, d⟩
22  Effect:
23      Dequeue(geocast-queue)
24
25  Output respond(resp)_p
26  Precondition:
27      ⟨p, resp⟩ ∈ ready-responses
28  Effect:
29      ready-responses ← ready-responses − {⟨p, resp⟩}
```

4.1.1 VOE-Client

The signature and state of the VOE-Client are presented in Figure 4-2, and the transitions for the VOE-Client are presented in Figure 4-3. The VOE-Client has three basic purposes. First, it ensures that each invocation receives at most one response by eliminating duplicates. It accomplishes this by attaching a unique tag to each request and delivering only one response for each tag. Second, it instantiates the invoke/response interface for the virtual object, delivering the requests via the GeoCast service to the actual replicas responsible for emulating the virtual object. Third, it provides each mobile node with multiple ports with which to access the virtual object.

4.1.2 VOE-Server

The VOE-Server performs the actual replicated emulation of the virtual object. The VOE-Server implements a replicated state machine. Thus, each mobile node, when it is active in the emulation, maintains a copy of the state of the virtual object. The mobile nodes use the total-ordering property of the $\mathsf{fpcast}_\mathsf{h}$ service to ensure that the replicated state is updated consistently. Figure 4-4 contains the signature and state of the VOE-Server. The remaining code for the VOE-Server is in Figures 4-5 and 4-5.

Emulator State. We begin by enumerating the elements of the VOE-Server's state:

- The *clock* and the *location* variables store the time and location information from the most recent geo-update.
- The *status* determines whether the emulator is currently participating in the emulation as a replica, joining the emulation, or idle.
- The value *val* stores the current state of the replica.
- The *join-id* is a unique identifier selected and used during the join protocol.
- The fpcast-queue and geocast-queue are two queues storing outgoing messages for the respective broadcast services.
- The *answered-join-reqs* is a set indicating which join requests have already been answered.
- The queue *pending-ops* stores messages waiting to be processed.
- The queue *completed-ops* stores messages that have already been processed.
- The counter *fpcast-cnt* is used to ensure that each fpcast message is unique.

[2]Recall that a mobile node is well inside a focal point region when it is sufficiently deep inside the region, as was defined in Section 2.2.2.

Figure 4-4: Automaton VOE-Server: Signature and State

Input:
- geocast-rcv($\langle$invoke, *inv*, *oid*, *loc*$\rangle$, *d*)$_i$, *inv* $\in$ *invocations*, *oid* $\in U$, *loc* $\in L$, *d* $\in L$
- fpcast-rcv($\langle$join-req, *jid*$\rangle$)$_{obj,i}$, *jid* $\in T$
- fpcast-rcv($\langle$join-ack, *jid*, *v*$\rangle$)$_{obj,i}$, *jid* $\in T$, *v* $\in V$
- fpcast-rcv($\langle$invoke, *inv*, *oid*, *loc*$\rangle$)$_{obj,i}$, *inv* $\in$ *invocations*, *oid* $\in U$, *loc* $\in L$
- geo-update(*l*, *t*)$_i$, *l* $\in L$, *t* $\in R^{>0}$

Output:
- geocast($\langle$response, *resp*, *oid*, *loc*$\rangle$, *d*)$_i$, *resp* $\in$ *responses*, *oid* $\in U$, *loc* $\in L$, *d* $\in L$
- fpcast($\langle$join-req, *jid*$\rangle$)$_{obj,i}$, *jid* $\in T$
- fpcast($\langle$join-ack, *jid*, *v*$\rangle$)$_{obj,i}$, *jid* $\in T$, *v* $\in V$
- fpcast($\langle$invoke, *inv*, *oid*, *loc*$\rangle$)$_{obj,i}$, *inv* $\in$ *invocations*, *oid* $\in U$, *loc* $\in L$

Internal:
- join()$_i$
- simulate-op(*inv*)$_i$, *inv* $\in$ *invocations*

Constants:
- *fp-center* : $\mathbb{R}^{\geq 0} \rightarrow L$, a function mapping time to the focal point center
- *fp-region* : $\mathbb{R}^{\geq 0} \rightarrow \mathcal{P}(L)$, a function mapping time to the focal point region

State:
- *clock* $\in R^{\geq 0}$, the current time, initially 0, updated by the geosensor
- *location* $\in L$, node i's location, initially i's initial location, updated by the geosensor
- *status* $\in$ {idle,joining,listening,active}, initially active if node is in the focal point region; idle otherwise
- *val* $\in V$, holds current value of the simulated atomic object, initially v_0
- *join-id* $\in T$, unique id for current join request, initially $\langle 0, i_0 \rangle$
- *fpcast-queue*, a queue of messages to be sent by fpcast, initially $\emptyset$
- *geocast-queue*, a queue of messages to be sent by the GeoCast, initially $\emptyset$
- *anwered-join-reqs*, set of ids of join requests that have already been answered, initially $\emptyset$
- *pending-ops*, queue of operations waiting to be simulated, initially $\emptyset$
- *completed-ops*, queue of operations that have been simulated, initially $\emptyset$
- *fpcast-count* $\in \mathbb{N}$, counter of fpcast operations, initially 1

Figure 4-5: Automaton VOE-Server: Input Transitions
for client i and object obj of variable type $\tau = \langle V, v_0, \mathit{invocations}, \mathit{responses}, \delta \rangle$

Input fpcast-rcv($\langle\langle$join-req, $jid\rangle$, cnt, $k\rangle$)$_{obj,i}$
Effect:
  $\mathit{completed\text{-}ops} \leftarrow \mathit{completed\text{-}ops} \cup \langle$join-req, $jid\rangle$
  if $((\mathit{status} =$ joining$) \wedge (\mathit{jid} = \mathit{join\text{-}id}))$ **then**
    $\mathit{status} \leftarrow$ listening
  if $((\mathit{status} =$ active$) \wedge (\mathit{jid} \notin \mathit{answered\text{-}join\text{-}reqs}))$ **then**
    Enqueue($\mathit{fpcast\text{-}queue}$, $\langle$join-ack, jid, val, $\mathit{completed\text{-}ops}\rangle$)

Input fpcast-rcv($\langle\langle$join-ack, jid, v, $\mathit{c\text{-}ops}\rangle$, cnt, $k\rangle$)$_{obj,i}$
Effect:
  $\mathit{answered\text{-}join\text{-}reqs} \leftarrow \mathit{answered\text{-}join\text{-}reqs} \cup \{\mathit{jid}\}$
  $\mathit{completed\text{-}ops} \leftarrow \mathit{completed\text{-}ops} \cup \langle$join-ack, jid, v, $\mathit{c\text{-}ops}\rangle$
  if $((\mathit{status} =$ listening$) \wedge (\mathit{jid} = \mathit{join\text{-}id}))$ **then**
    $\mathit{status} \leftarrow$ active
    $\mathit{completed\text{-}ops} \leftarrow \mathit{completed\text{-}ops} \cup \mathit{c\text{-}ops}$
    $\mathit{val} \leftarrow v$

Input fpcast-rcv($\langle\langle$invoke, inv, oid, $\mathit{loc}\rangle$, cnt, $k\rangle$)$_{obj,i}$
Effect:
  if $((\mathit{status} =$ listening $\vee$ active$)$
    $\wedge$
  $(\langle \mathit{inv}, \mathit{oid}, \mathit{loc}\rangle \notin \mathit{pending\text{-}ops} \cup \mathit{completed\text{-}ops}))$ **then**
    Enqueue($\mathit{pending\text{-}ops}$, $\langle$invoke, inv, oid, $\mathit{loc}\rangle$)

Input geocast-rcv($\langle$invoke, inv, oid, $\mathit{loc}\rangle$, d)$_i$
Effect:
  if $(\langle \mathit{inv}, \mathit{oid}, \mathit{loc}\rangle \notin \mathit{pending\text{-}ops} \cup \mathit{completed\text{-}ops})$ **then**
    Enqueue($\mathit{fpcast\text{-}queue}$, $\langle$invoke, inv, oid, $\mathit{loc}\rangle$)

Input geo-update(l, t)$_i$
Effect:
  $\mathit{clock} \leftarrow t$
  $\mathit{location} \leftarrow l$
  if $(\mathit{depth}_{obj}(\mathit{location}, \mathit{clock}) < t_{\mathsf{upd}} \cdot (v_{\mathsf{max}} + v_{\mathsf{fp\text{-}max}}))$ **then**
    $\mathit{status} \leftarrow \mathit{idle}$
    $\mathit{join\text{-}id} \leftarrow \langle 0, i_0\rangle$
    $\mathit{val} \leftarrow v_0$
    $\mathit{answered\text{-}join\text{-}reqs} \leftarrow \emptyset$
    $\mathit{pending\text{-}ops} \leftarrow \emptyset$
    $\mathit{fpcast\text{-}queue} \leftarrow \emptyset$
    $\mathit{geocast\text{-}queue} \leftarrow \emptyset$

Figure 4-6: Automaton VOE-Server: Internal and Output Transitions
for client i and object obj of variable type $\tau = \langle V, v_0, invocations, responses, \delta \rangle$

43 **Internal** simulate-op$(inv)_i$
44 **Precondition**:
45 $status$ = active
46 **Peek**$(pending\text{-}ops) = \langle$invoke, inv, oid, $loc\rangle$
47 **Effect**:
48 **if** ($\langle$invoke, inv, oid, $loc\rangle \notin completed\text{-}ops$) **then**
49 $(val, resp) \leftarrow \delta(inv, val)$
50 **Enqueue**$(geocast\text{-}queue, \langle\langle$response, $resp$, oid, $loc\rangle$, $loc\rangle)$
51 **Enqueue**$(completed\text{-}ops,$ **Dequeue**$(pending\text{-}ops))$
52
53 **Internal** join$()_i$
54 **Precondition**:
55 $depth_{obj}(location, clock) \geq t_{\text{upd}} \cdot (v_{\text{max}} + v_{\text{fp-max}})$
56 $status = idle$
57 **Effect**:
58 $join\text{-}id \leftarrow \langle clock, i\rangle$
59 $status \leftarrow$ joining
60 **Enqueue**$(fpcast\text{-}queue, \langle$join-req, $join\text{-}id\rangle)$
61
62 **Output** fpcast$(\langle m, fpcast\text{-}count, i\rangle)_{obj,i}$
63 **Precondition**:
64 **Peek**$(fpcast\text{-}queue) = m$
65 **Effect**:
66 $fpcast\text{-}count \leftarrow fpcast\text{-}count + 1$
67 **Dequeue**$(fpcast\text{-}queue)$
68
69 **Output** geocast$(m, d)_i$
70 **Precondition**:
71 **Peek**$(geocast\text{-}queue) = \langle m, d\rangle$
72 **Effect**:
73 **Dequeue**$(geocast\text{-}queue)$

Joining and Leaving the Emulation. The emulator transitions between three statuses: idle, joining, listening, and active. We first discuss how a mobile node joins the emulation. The join protocol is activated when the mobile node enters the inner region of focal point h, i.e., when it hears that it is well inside the focal point region. In this circumstance, the join transition is enabled (Figure 4-6, lines 53–60). This results in the mobile node broadcasting a *join-request* message using the fpcast_h service, at which point it sets its status to joining and awaits a response.

The other active nodes in the focal point receive the join request (Figure 4-5, lines 1–7) and respond by sending the current replicated state of the virtual object using the fpcast_h service. As an optimization to avoid unnecessary message traffic, if a node observes (by examining *answered-join-reqs*) that someone else has already responded to a join request, then it does not respond; only one node needs to send a join response. Notice (Figure 4-5, line 6) that only active nodes will send join responses. When the node that originated the join request receives its own join request (Figure 4-5, lines 4–5), it sets its status to listening.

When the joining node receives a response to its join request (Figure 4-5, lines 9–16), it starts participating in the emulation, setting its status to active. This completes the join protocol.

By contrast, ceasing to participate in the emulation does not require any coordination with other replicas. When a node hears that it has left the inner focal point region (either due its own motion, or the movement of the virtual object), it re-initializes its variables, and becomes idle (Figure 4-5, lines 30–41).

Responding to Invocations. We next discuss how active replicas respond to an invocation. When a node participating in focal point h receives a GeoCast message containing an invocation on object h (Figure 4-5, lines 25–28), it rebroadcasts the invocation with the $\mathsf{fpcast}_\mathsf{h}$ service, thus ordering the invocation with respect to the other messages broadcast with the $\mathsf{fpcast}_\mathsf{h}$ service (i.e., join-request messages, join responses, and other operation invocations). Since it is possible that a GeoCast invocation is received—and rebroadcast—by more than one node in the focal point, there is some bookkeeping to ensure that only one copy of the same invocation is actually processed by the nodes. We include an optimization that if a node observes via the *pending-ops* or *completed-ops* queues that an invocation has already been sent by the $\mathsf{fpcast}_\mathsf{h}$ service, then it does not do so again.

Active nodes keep track of invocations in the order in which they receive the invocation messages via the $\mathsf{fpcast}_\mathsf{h}$ service (Figure 4-5, lines 18–23). Duplicates are discarded using the unique operation identifiers, which are stored in the sets *pending-ops* and *completed-ops*. The operations are then performed (Figure 4-6, lines 43–51) on the simulated state in this order. After each operation is executed, a GeoCast is sent back to the invoking node with the operation's response. We can be sure that the invoking node receives a response when it remains in the same region from which it sent the invocation, allowing the GeoCast to find it.

There are some subtle points involved in deciding when during the process of joining the emulation a node should start recording invocation messages, when a node

should start executing operations, and when a node should start sending responses. During the process of joining, a node receives a snapshot of the object's state. However by the time the snapshot is received, it might be out of date, since there may have been some intervening messages delivered by the $\mathsf{fpcast}_\mathsf{h}$ service that have been received since the snapshot was sent. Therefore the joining node must record all the operation invocations that are broadcast after its join request was broadcast but before it receives the snapshot. This is accomplished by having the joining node enter a "listening" state once it receives its own join-request message; all invocations received when a node is in either the listening or the active state are recorded, and actual processing of the invocations can start once the node has received the snapshot and has the active status. The processing consists of first emptying the queue of waiting invocations, and then proceeding to handle newly arrived invocations.

A final point to notice is that every message sent by the fpcast_h service is unique: attached to each message is a counter (*fpcast-cnt*) and the process identifier; each processor increments its counter after each fpcast, ensuring that no message is duplicated. Notice that each message m that is fpcast is of the form $\langle P, cnt, k\rangle$, where cnt is a counter and k is a node identifier. We refer to P as the **payload** of m.

4.2 Analysis

In this section we show that the Virtual Object Emulator correctly implements the Virtual Object Layer. We first focus on a single focal point h, and show that the Virtual Object Emulator for h correctly emulates the canonical atomic object associated with h. The key tool in this proof is Theorem A.3.2 (see Appendix A.3 for more details). In order to use this theorem, we define a total order on all the operations simulated by the VOE, and show that this order satisfies certain properties. Once we have shown that each individual VOE emulates a single virtual object, we argue that the entire emulation of all the virtual objects results in an emulation of the Virtual Object Layer. Of note, throughout this proof, we depend only on the basic timing assumptions described in Chapter 2, i.e., the accurate time and location updates received sufficiently frequently from the (RealWorld) GeoSensor. In Section 4.3, we make additional timing assumptions regarding the behavior of the mobile nodes and the broadcast service, and analyze the performance of the emulator.

We begin by describing what it means for the Virtual Object Emulator to emulate the Virtual Object Layer. For each $h \in O$, let A_h be the canonical atomic object automaton specified by the type of virtual object h. (See Appendix A.3 for a definition of the canonical atomic object automaton.) Notice that A_h is an asynchronous automaton, formalized with TIOA, that is, with arbitrary time passage between discrete events. Let S be the composition of all the VOE-Clients, VOE-Servers, and the RealWorld, where all the actions that are not invoke, respond, or geo-update actions are hidden. For each $h \in O$, let S_h be automaton S with all actions except for invocations and responses for object h hidden. Thus, the external interface to S_h consists only of invoke and respond actions on object h. (Notice that all the hidden actions are output actions, since the only inputs to S are invoke actions.) The result of this

definition is that the external interface of S_h is identical to the external interface of A_h.

Our first goal is to show that every trace of S_h is also a trace of A_h. That is, for every possible pattern of invocations by the clients, the result trace of the emulator is also a trace of the atomic object A_h. The majority of the analysis lies in proving this claim. Notice that S_h is a timed automaton, while A_h is asynchronous in the sense that it allows for arbitrary time passage events between discrete events. Note that traces of S_h and A_h are both timed, even though A_h is asynchronous.

Once we have shown that every trace of S_h is also a trace of A_h, it remains to show that for the entire composition of the emulators and the RealWorld, i.e., for the emulation of all the objects simultaneously, every trace is also a trace of the Virtual Object Layer. We show this fact in Section 4.2.4 by pasting together the individual client and virtual object executions.

We now fix one $h \in O$ and focus on showing that S_h implements A_h. In particular, we need to show that every trace of S_h is also a trace of A_h. (Again, recall that traces are timed.) Formally, let U be any well-formed environment: a well-formed environment is an arbitrary timed automaton that preserves well-formedness. Our goal is to show that $traces(U \times S_h) \subseteq traces(U \times A_h)$.

The main body of the proof consists of determining a total ordering on all the operations of the object h, given an arbitrary execution in which all the operations complete. The total ordering is shown to have certain properties, allowing us to conclude that the algorithm correctly implements an atomic object. Specifically, the key properties are that the ordering is consistent with the real-time ordering of operations, and that the ordering is consistent with the responses sent during the execution. We then use Theorem A.3.2 (see Appendix A.3 for further details) to conclude the proof.

Fix an arbitrary execution α of the system $U \times S_h$ in which every operation completes, where U is an arbitrary environment for S_h that is compatible with S_h and preserves well-formedness. (Theorem A.3.2 indicates that we need only consider executions in which all the operations complete.) Let Π be the set of operations in α. Since every operation completes, we know that for every operation $\pi \in \Pi$, for some port $p \in Q$ (the set of ports), an $\mathsf{invoke}(\pi)_{h,p}$ event occurs in α, followed by a $\mathsf{respond}(\pi)_{h,p}$ event.

We begin in Section 4.2.1 to define some notation, and a total order on the operations in Π. These operations are ordered based on the order in which the associated fpcast messages are delivered. We then proceed in Section 4.2.2 to prove certain properties of this total order. Specifically, in Invariant 4.2.1 we prove the key property of the total order: at every point in the execution α, the replicated state of each VOE active in the replication is equal to the state after some prefix of the ordered operations. Notice that this claim alone is sufficient to show that the replicas agree on an an execution of the virtual object, and moreover that the agreed-upon execution is exactly defined by the total order on operations. In Section 4.2.3, we invoke Theorem A.3.2 to show that the VOE emulator S_h implements the virtual object A_h, and in Section 4.2.4 conclude that the VOE emulator as a whole implements the Virtual Object Layer.

4.2.1 Totally Ordered Operations

We begin by defining some preliminary notation. Recall that we have already fixed an execution α of $S_h \times U$. We define $id(\pi)$ to be the unique identifier of each operation π, and $id(m)$ to be the operation identifier of each message m; for each prefix β of α we define $IM(\beta)$ to be the set of messages sent by the fpcast_h service in β. Finally, we define a total order $\prec$ on the operations in Π based on the messages in $IM(\alpha)$. We proceed in more detail.

First, recall that every operation begins, by definition, with an invocation on some port, and notice that each operation $\pi \in \Pi$ is assigned a unique identifier $\langle clock, p\rangle$ by the VOE-Client before it is GeoCast to the VOE-Servers (see Figure 4-3, line 3). We refer to this identifier as $id(\pi)$.

Second, for any prefix β of execution α, let $IM(\beta)$ be the set of messages sent (and later delivered) by the fpcast_h service in β. That is:

$$IM(\beta) = \{m \mid \exists i \in I, \mathsf{fpcast}(m)_{h,i} \in \beta\}$$

The set $IM(\beta)$ contains one message m for every $\mathsf{fpcast}(m)$ event in β.

Third, since each message fpcast is unique, the properties of the fpcast_h service guarantee that there exists a total ordering of the messages in $IM(\alpha)$ that is consistent with the order in which each mobile node receives the messages. That is, if node i receives two messages m_r and m_t in that order, then $m_r \prec m_t$ in the total ordering. Fix $\prec$ to be this ordering.

Fourth, notice that a "message" m consists of the tuple: $\langle\langle op, inv, oid, loc\rangle, \cdot, \cdot\rangle$, the tuple sent by an fpcast. Each invoke message, that is, each message in which $op =$ invoke, includes an identifier $oid \neq \perp$. If m is an invoke message, then let $id(m)$ be the identifier associated with message m. This identifier is closely related to the identifier associated with operation, $id(\pi)$ that led to this message: for each invoke message, oid is set to the unique identifier of an operation associated with that message. (See Figure 4-5, line 28, and notice that the enqueued fpcast_h message uses the identifier from the received GeoCast request). This allows us to say that certain fpcast_h messages are associated with each operation. However, it also means that the oid identifiers in each message are not unique: two fpcast_h messages (sent by different mobile nodes) may have the same oid identifier as they may result from two different mobile nodes receiving the same geocast invoke message.

Finally, we define a total ordering on operations $\pi \in \Pi$ as follows, using the ordering $\prec$ induced by the fpcast_h service. Let π_i and π_j be two operations in Π. For operation π_i, let m_1 be the first message according to the ordering $\prec$ in $IM(\alpha)$ associated with operation π_i, that is, where $id(m_1) = id(\pi_i)$. The message m_1 is the first message delivered by fpcast_h in α that was generated by operation π_i.

Similarly, for operation π_j, let m_2 be the first message according to the ordering $\prec$ in $IM(\alpha)$ such that $id(m_2) = id(\pi_j)$. We say that $\pi_i \prec \pi_j$ if $m_1 \prec m_2$. Since operation identifiers are unique and an operation can only be associated in this way with a single message (i.e., the first), this defines a total ordering on operations.

4.2.2 Properties of the Total Order

Our goal now is to show that the total ordering on operations satisfies the criteria of Theorem A.3.2, thus implying that the emulation of atomic object A_h is correct. The most difficult property required by Theorem A.3.2 is Property 2: we need to show that the total ordering of operations is consistent with the responses sent by the VOE-Servers. In order to show this property, we focus on a key invariant that relates the total ordering to the state of each replica. Invariant 4.2.1 shows that at every point in the execution, the state of each replica precisely reflects a prefix of the ordered operations. From this we are able to conclude that each response sent by a replica is also consistent with the total order.

We begin by defining, for each prefix of α and for each replica i, a prefix of the ordered messages in $IM(\alpha)$; this prefix of the ordered messages represents the set of messages processed by i. It is easy to see that every message processed by i is in this set. We argue (in the proof to Invariant 4.2.1) that this set exactly captures the set of messages that have been processed to produce i's replicated state. That is, i's replicated state is exactly the value v resulting from executing each operation in this designated prefix of $IM(\alpha)$. We now proceed in more detail.

Let α' be any finite prefix of execution α, and let i be any node. Recall that each message $m \in IM(\alpha)$ consists of a payload P, along with a counter and a node identifier. Of all the fpcast_h messages ordered in $IM(\alpha')$, we are interested in a particular subset of these messages: those that are delivered in α' to i that have not had their payload added to *pending-ops*$_i$. This subset includes all the messages processed by i in α', since every message processed by i has been received and is no longer in the pending queue. More formally, define $\overline{IM(\alpha', i)}$ to be the set of messages $\bar{m}$ that satisfy the following three conditions:

1. $\mathsf{fpcast\text{-}rcv}(\bar{m})_{obj,i}$ occurs in α', and
2. $\bar{m} = \langle P, \cdot, \cdot \rangle$ and
3. $P \notin \ell state(\alpha').pending\text{-}ops_i$.

Here $\ell state(\alpha')$ denotes the last state in α'.

We now define $\gamma(\alpha', i)$ to be the state of the atomic object after processing all the operations in α' up to and including the last operation in α' processed by i, that is, the operations represented by all the messages in $IM(\alpha')$ up to and including the largest message in $\overline{IM(\alpha', i)}$. (An operation is processed when it is applied to the state of the atomic object.) We now define $\gamma(\alpha', i)$ more formally.

First, choose $\bar{m}$ to be the largest message (according to the total ordering $\prec$) in $\overline{IM(\alpha', i)}$. The message $\bar{m}$ is, in effect, the most recent message that has been processed by node i: all prior messages have been received, added to the set *pending-ops*$_i$, and then removed from the set *pending-ops*$_i$; all later messages are in the set of pending operations *pending-ops*$_i$, or have not yet been received. If no such $\bar{m}$ exists, i.e., if $\overline{IM(\alpha', i)} = \emptyset$, then set $\bar{m} = \perp$.

We now define $\gamma(\alpha', i)$ to be the state of the atomic object after processing all fpcast_h messages in $IM(\alpha)$ prior to (and including) $\bar{m}$. That is, $\gamma(\alpha', i)$ is the state after beginning in the initial state v_0 and processing all the invoke messages for node i in $IM(\alpha)$, stopping after message $\bar{m}$, while skipping the "duplicate" messages, referring to the same operation, that might occur in $IM(\alpha')$. If $\bar{m} = \perp$, then $\gamma(\alpha', i) = v_0$.

Thus $\gamma(\alpha', i)$ is the state after processing the operations, $\pi_1, \pi_2, \ldots, \pi_t$, where π_t is the most recent operation that i has processed in α'.

We show that for every prefix α' of execution α, if node i has completed the join protocol, then the state of the replica at node i is equal to $\gamma(\alpha', i)$. That is, the state of the replica at node i is consistent with all prior operations in $IM(\alpha')$ having occurred.

If node i has itself received all the messages in $IM(\alpha')$, this claim is immediate, as it executes each operation that it receives. In the case that node i has joined the focal point during the execution, however, node i may have received only a suffix of the sequence. As a result, the main difficulty in proving the following invariant is showing that the join protocol works, i.e. that after a node sets its status to active, it has correctly acquired a good snapshot of the state of the world.

Invariant 4.2.1. *Let α' be any finite prefix of execution α. If $\ell state(\alpha').status_i =$ active, then $\ell state(\alpha').val_i = \gamma(\alpha', i)$.*

Proof. We show this by induction on the length of α'. Throughout the induction, we also maintain a second invariant. This sub-invariant shows that the sets *pending-ops* and *completed-ops* together reflect all the operations that i is aware of. Specifically, every message m' that precedes the largest received message is in one of those two sets.

Invariant 4.2.2. *For every $i \in I$: if $\ell state(\alpha').status_i =$ active and message m is the largest message such that a $\mathsf{fpcast\text{-}rcv}(m)_{h,i}$ occurs in α', then for every message $m' \leq m, m' \in IM(\alpha')$ where $m' = \langle P, \cdot, \cdot \rangle$, the payload P is in $\ell state(\alpha').pending\text{-}ops_i \cup \ell state(\alpha').completed\text{-}ops_i$.*

For the base case (for both the main invariant, as well as Invariant 4.2.2), consider the initial state of the system, before any events occur in α. Let β' be this empty prefix of α. If i is not in the focal point region, then $status_i =$ idle, and both claims are trivial. If i is in the focal point, then the state of val_i is v_0. In this case $IM(\beta')$ is empty, and as a result $\gamma(\beta', i)$ also equals v_0, satisfying the main invariant. Invariant 4.2.2 follows from the fact that there are no messages received in the empty execution.

There are two types of inductive steps to consider for executions of timed automata: discrete events and trajectories. Notice that for trajectories, the invariants are trivially maintained since all the state elements mentioned in the invariants are discrete and unchanged under time passage. Consider, then, a discrete event.

Let s and s' be the states before and after the new event, respectively. Let β be the previous prefix of α, that is, $s = \ell state(\beta)$. Let β' be the new prefix of α, that is, $s' = \ell state(\beta')$.

We already know, inductively, that for any finite prefix α' of execution β, for any node $j \in I$, if $\ell state(\alpha').status_j = \mathsf{active}$, then: $\ell state(\alpha').val_j = \gamma(\alpha', j))$, and Invariant 4.2.2 holds. We need to show that $\ell state(\beta').val_i = \gamma(\beta', i)$, and that Invariant 4.2.2 continues to hold. We now consider the various actions relevant to this claim.

- $\mathsf{fpcast\text{-}rcv}(\langle\langle \mathsf{invoke}, inv, oid, loc\rangle, cnt, k\rangle)_{obj,i}$: (Figure 4-5, lines 18–23.)
  Recall that the set $\overline{IM(\beta', i)}$ includes only messages that have been sent in β', received by i, and whose payload is no longer in *pending-ops*$_i$. As a result of this event, the message payload $\langle \mathsf{invoke}, inv, oid, loc\rangle$ is added to *pending-ops*$_i$, and therefore the sequence $\overline{IM(\beta', i)}$ is equal to $\overline{IM(\beta, i)}$, and as a result, $\gamma(\beta', i) = \gamma(\beta, i)$. By induction, we already know that $\gamma(\beta, i) = \ell state(\beta).val_i$. The state of the replicated object is also unchanged, that is: $\ell state(\beta').val_i = \ell state(\beta).val_i$. The claim then follows.

  We next proceed to argue that Invariant 4.2.2 is maintained. Notice that the message $m = \langle\langle \mathsf{invoke}, inv, oid, loc\rangle, cnt, k\rangle$ is the largest message received by i from the fpcast_h service in β', since the total-ordering reflects the order in which messages are received. Let m' be the largest message received by i in β from the fpcast_h service. The induction hypothesis states that every message $m'' \leq m'$ in $IM(\beta)$ has its payload in either *pending-ops* or *completed-ops*. Since this event does not remove messages from either of these sets, and since this event adds payload $\langle\langle \mathsf{invoke}, inv, oid, loc\rangle, \cdot, \cdot\rangle$ to *pending-ops*, it remains to show that for every message $m' < m'' < m$ in $IM(\beta')$, they payload of m'' is either in *pending-ops* or *completed-ops*. Assume for the sake of contradiction that such a message m'' exists that its payload is not in either set. First, we argue that this implies that node i has exited the focal point region at some point after receiving message m' and prior to receiving message m: since the payload of every message received from the fpcast_h service is added to either *pending-ops* or *completed-ops*, and messages are removed only when i exits the focal point region, there are two possible cases:

  - Node i exits the focal point region at some point between receiving m and m', and resets *pending-ops* during the $\mathsf{geo\text{-}update}$), removing message m'' from *pending-ops*.
  - Node i does not receive m''. In this case, we conclude by the *consistent delivery* property of the fpcast_h service that i exits the focal point region at some point between receiving m and m'.

  Moreover, we know that message m' is the most recent message received by i prior to m (since it is the largest message in β). Thus between receiving message m and m', node i exits the focal point region, setting its status to idle, and node i does not receive a $\mathsf{join\text{-}ack}$ message, allowing it to reset its status to active, contradicting our assumption that i has a status of active.

- $\mathsf{simulate\text{-}op}(inv)_i$: (Figure 4-6, lines 43–51.)
  First, when this action occurs, $status_i = \mathsf{active}$, since that is a precondition

to this action (Figure 4-6, line 45). The invoke operation removes a message payload $P = \langle inv, \ldots \rangle$ from the set $pending\text{-}ops_i$, and in doing so adds message $m = \langle P, \cdot, \cdot \rangle$ to $\overline{IM(\beta, i)}$. As a result, $\overline{IM(\beta', i)} = \overline{IM(\beta, i)} \cup \{m\}$.

We next notice that the state $\gamma(\beta, i)$ includes all the messages preceding m in $IM(\beta')$. Otherwise, it would imply that the message immediately preceding m in $IM(\beta')$ was not received by i in β and hence that i had left the focal point region and set $status_i = \mathsf{idle}$ (during a geo-update), contradicting our assumption that i is active.

The total-ordering guarantee of the fpcast service ensures that m is the largest message in $\overline{IM(\beta', i)}$. There are two subcases to consider.

- Message m has the same payload as a message already received and processed. That is, there exists a message delivered earlier in the ordering with the same invocation and operation identifier. In this case, $\gamma(\beta', i) = \gamma(\beta, i)$. It remains to show that val_i is also unmodified as a result of the $\mathsf{fpcast\text{-}rcv}_h$ event. Inductively, we know from Invariant 4.2.2 that every message in $IM(\beta)$ has its payload in $pending\text{-}ops_i \cup completed\text{-}ops_i$. Thus, the earlier message with the same operation identifier also has its payload in $pending\text{-}ops_i \cup completed\text{-}ops_i$. We know that it is not in $pending\text{-}ops_i$, as the payload for message m is the oldest in *pending-ops* (as it is a queue); from this we conclude that the earlier message is in $completed\text{-}ops_i$, in which case the simulated-op event does not modify val_i, and the main invariant is maintained.
- Message m refers to a new operation, in which case $\gamma(\beta', i) = \delta(inv, \gamma(\beta, i))$. In this case, the state $val_i = \gamma(\beta, i)$ in state s, by induction. After the simulate-op action, val_i is set to $\delta(inv, \gamma(\beta, i))$, as this is the definition of how the object responds to invocations. This maintains the desired invariant.

Since this event neither results in a new message being received from the fpcast_h service, nor removes any message payloads from *pending-ops* $\cup$ *completed-ops* (only moving a message from one to the other), Invariant 4.2.2 is maintained.

- $\mathsf{fpcast\text{-}rcv}(\langle \langle \mathsf{join\text{-}ack}, jid, v, c\text{-}ops \rangle, \cdot, \cdot \rangle)_i$: (Figure 4-5, lines 9–16.)
In this event, node i sets *status* to active (Figure 4-5, line 14), if the message is a response to an outstanding join-req previously sent by i. (If this is not the case, then this action causes no change to $status_i$, $pending\text{-}ops_i$, or val_i, and the invariant is trivially maintained.)

  The fpcast_h service guarantees that if a message is received, an earlier fpcast occurred at some node j that sent the message (that is, it guarantees message integrity). In particular, some node j previously fpcast a join acknowledgment, i.e., performed a $\mathsf{fpcast}(\langle \mathsf{join\text{-}ack}, jid, v, c\text{-}ops \rangle)_j$.

  The only action that causes a join-ack to be sent is a prior join request being received . Therefore, node j previously performed an $\mathsf{fpcast\text{-}rcv}(\langle \mathsf{join\text{-}req}, jid \rangle)_i$. Let β'' be the prefix of β ending with the $\mathsf{fpcast\text{-}rcv}(\langle \mathsf{join\text{-}req}, jid \rangle)_j$ action.

Consider the state of node j at the end of β''. First, the status, $status_j$, must be active; otherwise node j would not send a response to the join request. Inductively, then, we know that $\ell state(\beta'').val_j$ is equal to $\gamma(\beta'', j)$. Moreover, we can conclude that v, the value send from j to i, is also equal to $\gamma(\beta'', j)$.

Since val_i is set to v when the join-ack message is received, i.e., during the transition in question, it remains only to show that:

$$\gamma(\beta', i) = \gamma(\beta'', j) \ , \tag{4.1}$$

and we can conclude that $val_i = \gamma(\beta', i)$, as desired.

Let m be the largest message in $IM(\beta')$ received by i that does not have its payload in $s'.pending\text{-}ops_i$, that is, m is the largest messages in $\overline{IM(\beta', i)}$. We claim that m must correspond to node i's join request, which implies Equation 4.1.

First, notice that i ignores all messages that it receives before its own join request. None of these messages have their payload added to $pending\text{-}ops_i$, and therefore m is no smaller than all such messages.

Second, notice that i adds the payload of every message it receives after its own join request to $pending\text{-}ops_i$ because when i receives its own join request, it sets $status_i$ to listening (Figure 4-5, line 5). No message is removed from the set $pending\text{-}ops_i$ because $status_i$ is not yet active (Figure 4-6, line 45). Therefore m cannot be equal to any message received by i after i's own join request.

We conclude, then, that m is exactly i's join request. The state $\gamma(\beta', i)$ is defined as the state reached after processing all messages prior to and including m, that is, prior to i's join request.

Notice, though, that i's join request is exactly the last message processed by j in β'' before sending a response to i. In particular, then, $\gamma(\beta'', j)$ is the state reached on processing every message in $IM(\beta''$ prior to i's join request.

Therefore, Equation 4.1 holds, and the invariant holds in state s'.

We next argue that Invariant 4.2.2 is also maintained. Notice that when node j sends the join-ack message, it also includes a copy of its set $completed\text{-}ops_j$, and that at this point in the execution, Invariant 4.2.2 holds with respect to j, meaning that every message preceding node i's join request has its payload in $pending\text{-}ops_j \cup completed\text{-}ops_j$. Since $pending\text{-}ops_j$ is a queue, and when responding to the join request, all the messages preceding it have been removed from $pending\text{-}ops_j$, we can conclude that every message preceding the join request has its payload in $completed\text{-}ops_j$. When node i receives the join-ack, it adds all the messages in the set $c\text{-}ops$ to its own set $completed\text{-}ops_i$. It remains only to show that every message sent after i's join request has its payload in $pending\text{-}ops_i$. This follows immediately from the fact that i has remained in the focal point region, and the *consistent delivery* property of the fpcast_h service.

The rest of the cases are straightforward, having no effect on the status of i, the elements of $pending\text{-}ops_i$, or the state of the replicated object. □

4.2.3 Atomicity

We can now show that the two properties required by Theorem A.3.2 hold for the total ordering we have defined, and as a result, that the Virtual Object Emulator correct emulates virtual object h. Specifically, the emulator S_h when composed with any well-formed environment U is a correct implementation of the object A_h composed with the same well-formed environment U.

Theorem 4.2.3. *traces*$(S_h \times U) \subseteq$ *traces*$(A_h \times U)$.

Proof. Fix some execution α of $S_h \times U$. We consider the properties from Theorem A.3.2 in order:

1. (Property 1) Assume π_i and π_j are two operations and π_i completes before π_j begins. We want to show that $\pi_i < \pi_j$ in the total ordering. Let message m_1 be the first message associated with operation π_i, and let message m_2 be the first message associated with operation π_j. Since π_i completes before π_j begins, the message m_1 must be received before the message m_2 is sent, and therefore m_1 precedes m_2 in the total ordering induced by the fpcast service. This then implies that π_i precedes π_j in the total ordering on operations, as claimed.

2. (Property 2) We need to show that the total ordering on operations is consistent with the responses output. This follows from Invariant 4.2.1.

   Fix some operation $\pi \in \Pi$. Let $inv_1, inv_2, \cdots, inv_k$ be the invocations of the operations preceding π in the total ordering, indexed according to the total ordering. Let $inv(\pi)$ be the invocation that initiates π, and $resp(\pi)$ be the response that concludes π.

   Let v be the value of the variable type that results from starting with the initial value, v_0, and processing the following invocations:

   $$inv_1, inv_2, \ldots, inv_k \ .$$

   We need to show that the response to operation π is consistent with the object being in state v. More specifically, we need to show that for some v':

   $$\langle resp(\pi), v' \rangle = \delta(inv(\pi), v) \ .$$

   A VOE-client automaton that delivers a response for π does so because it receives a GeoCast with a response for π for the first time from an VOE-server, say node i. Let α' be the prefix of α ending just before the simulate-op event that causes i to enqueue the GeoCast response. The value v determined by starting in state v_0 and handling all the invoke(π_r) operations prior to π is exactly $\gamma(\alpha', i)$: every preceding operation is associated with an earlier fpcast message (by the way in which the total ordering is defined); at the time when node i invokes operation π, it has removed all prior messages from the *pending-ops* queue, and therefore the message associated with operation π is exactly the largest message that i

has received in α' that is not in *pending-ops*$_i$. Invariant 4.2.1 shows that the state of val_i prior to operation π is equal to $\gamma(\alpha', i)$. Therefore the response to operation π is exactly that obtained by applying π to $\gamma(\alpha', i)$.

Therefore we conclude from Theorem A.3.2 that there exists an execution γ of $A_h \times U$ such that $trace(\alpha) = trace(\gamma)$, implying the desired result. □

4.2.4 Pasting

Finally, we consider the entire Virtual Object Layer. Let C be the composition of all the clients. Recall that S is the composition of all the VOE-Clients, VOE-Servers, and the RealWorld, with all the actions hiddent except invoke, respond and geo-update actions. Let A be the composition of all the virtual objects A_h for $h \in O$, and let VRW be the virtual RealWorld automaton. Our goal is to show that $traces(C \times S) \subseteq traces(C \times VRW \times A)$. (Recall that we are considering *timed* traces.) Notice that the only external actions in both the underlying system and the Virtual Object Layer are the invoke, respond, and geo-update actions.

In order to show this fact, we first fix an arbitrary execution α of $C \times S$. From this execution, we derive (timed) executions γ_i, for every client $i \in I$, and (timed) executions γ_h, for every $h \in O$, that correspond to executions of the clients and virtual objects in the Virtual Object Layer. We also derive an execution γ_{VRW} of the virtual RealWorld automaton. We then paste these executions together to form an execution γ of $C \times VRW \times A$ such that $trace(\alpha) = trace(\gamma)$, which implies the desired trace property.

We will need the following key lemma to paste the executions together. This lemma is an extension of Theorem 7.3 from [45, 46]:

Lemma 4.2.4. *Let A_1 and A_2 be compatible timed automata, and $A = A_1 \times A_2$. Let α_1 and α_2 be executions of A_1 and A_2, respectively.*

Let β be an $(E, \emptyset)$-sequence, where E is the set of external actions of A. Suppose that $\beta|(E_i, \emptyset) = trace(\alpha_i)$, $i \in \{1, 2\}$.

Then there exists an execution α of A such that $trace(\alpha) = \beta$, and $\alpha_i = \alpha|(A_i, X_i)$, $i \in \{1, 2\}$.

It follows that we can paste together a finite collection of components:

Corollary 4.2.5. *Let $A_1, A_2, \ldots, A_k$ be a finite collection of compatible timed automata, and let $A = A_1 \times A_2 \times \ldots \times A_k$. Let α_i be an execution of A_i.*

Let β be an $(E, \emptyset)$-sequence, where E is the set of external actions of A. Suppose that $\beta|(E_i, \emptyset) = trace(\alpha_i)$, $i \in \{1, 2, \ldots, k\}$.

Then there exists an execution α of A such that $trace(\alpha) = \beta$, and $\alpha_i = \alpha|(A_i, X_i)$, $i \in \{1, 2, \ldots, k\}$.

We now proceed to prove the main result of this chapter:

Theorem 4.2.6. *The Virtual Object Emulator implements the Virtual Object Layer.*

Proof. Recall that our goal is to show that $traces(C \times S) \subseteq traces(C \times VRW \times A)$. Begin by fixing an execution α of $C \times S$.

For each $i \in I$, we derive an execution γ_i of client i in the Virtual Object Layer that has the same trace as the client in α. Specifically, define $\gamma_i = \alpha|\langle E_i, X_i\rangle$, where E_i is the set of external actions for client i and X_i is the set of variables for client i. Notice that since α is a timed execution, γ_i is a timed execution.

Next, for each $h \in O$, we derive an execution γ_h of virtual object A_h in the Virtual Object Layer that has the same trace as S_h in α. Notice that the clients C are a well-formed environment, and hence given an execution $\alpha|\langle T, Y\rangle$ of $C \times S_h$ where T contains all the actions of C and S_h and Y contains all the variables of C and S_h, we know by Theorem 4.2.3 that there exists an execution γ' of $C \times A_h$ such that $trace(\gamma') = trace(\alpha)|\langle E_h, \emptyset\rangle$, where E_h is the set of external actions for A_h. Thus we define $\gamma_h = \gamma'|\langle E_h, X_h\rangle$, where X_h is the set of variables for A_h. Thus γ_h is an execution of A_h such that $trace(\alpha)|\langle E_h, \emptyset\rangle = trace(\gamma_h)$. Notice that γ_h is a timed execution.

Finally, we construct execution γ_{VRW} of the virtual RealWorld automaton. While the underlying RealWorld both outputs geo-updates and also implements the GeoCast service, the virtual RealWorld only outputs geo-updates. We construct the execution of the virtual RealWorld simply by restricting α to the geo-update events, which are the only external events by the virtual RealWorld automaton. More specifically, let E_{VRW} be the set of external actions for the virtual RealWorld automaton, i.e., the $\mathsf{geo\text{-}update}_i$ transitions for $i \in I$. Then we define $\gamma_{VRW} = \alpha|\langle E_{VRW}, X_{VRW}\rangle$, where X_{VRW} is the set of variables for the virtual RealWorld automaton.

We now need to construct a hybrid sequence β consisting of invoke event, respond events, and geo-update events, along with appropriate timing information. This sequence β is used to paste together the various execution components. We derive β from the invoke,respond, and geo-update events in α. More specifically, let $E_C = \bigcup_{i \in I}\{\mathsf{invoke}_i, \mathsf{respond}_i, \mathsf{geo\text{-}update}_i\}$. Define $\beta = \alpha|\langle E_C, \emptyset\rangle$.

It is immediately clear by construction that for each client $i \in I$, the trace of β is equal to the trace of γ_i; that is, $\beta|\langle E_i, \emptyset\rangle = \gamma_i|\langle E_i, \emptyset\rangle$, where E_i is the external actions of client i. This follows since both β and γ_i are constructed from α to include i's external events (and no other events of i).

Observe too that for each virtual object $h \in O$, the trace of β is equal to the trace of γ_h; that is, $\beta|\langle E_h, \emptyset\rangle = \gamma_h|\langle E_h, \emptyset\rangle$, where E_h is the external actions of virtual object h. This follows from the fact that $trace(\gamma_h) = trace(\alpha)|E_h$; since every external event in γ_h is in α, we can conclude that each such event also occurs in β.

Lastly, the trace of β is equal to the trace of γ_{VRW}; that is, $\beta|\langle E_{VRW}, \emptyset\rangle = \gamma_{VRW}|\langle E_{VRW}, \emptyset\rangle$, where E_{VRW} is the set of external actions for the virtual RealWorld. This follows from the fact that both β and γ_{VRW} are constructed from α to include the virtual RealWorld's external events (and no other events of the virtual RealWorld).

We thus conclude from Corollary 4.2.5 that there exists an execution γ of $C \times VRW \times A$ where $trace(\gamma) = trace(\alpha)$.

□

4.3 Performance Analysis

We now briefly discuss the performance of the Virtual Object Emulator, and determine the maximum latency of an operation. In particular, our goal is to prove the *client* and *virtual object* responsiveness properties. For the purpose of analyzing the performance of the focal point emulator, we make the following additional liveness assumptions:

- Any enabled action at a mobile node is executed immediately.
- For every fpcast event, the ϵ specified by the "Reliable Delivery" property is bounded by d_{fp}. This implies that every message broadcast using the fpcast service is delivered within time d_{fp}.
- For every geocast event, the ϵ specified by the "Reliable Delivery" property is bounded by ϵ_{geo}.
- Every message broadcast using the geocast service is delivered within time d_{geo}.

We also assume, in this case, that $t_{\mathsf{join}} > 2d_{\mathsf{fp}} + \epsilon_{\mathsf{geo}} + t_{\mathsf{upd}}$, which means that the join protocol has sufficient time to complete from the first point at which a mobile node is well inside a focal point region.

First, notice that the *client responsiveness* property follows immediately from the assumption that an enabled action at a mobile node is executed immediately. We focus for the remainder of this section on the *virtual object responsiveness* property.

For the purpose of this analysis, we consider a virtual object h that does not fail, and we assume that some correct mobile node i at location ℓ invokes an operation on object h; moreover, we assume that node i remains for sufficiently long within distance R_{geo} of location ℓ. We conclude that node i receives a response to its invocation within time $2d_{\mathsf{geo}} + d_{\mathsf{fp}}$:

Theorem 4.3.1. *If a mobile node i at location ℓ invokes an operation on a correct virtual object h, and if i remains within distance R_{geo} of location ℓ for time $2d_{\mathsf{geo}} + d_{\mathsf{fp}}$, then node i receives a response within time $2d_{\mathsf{geo}} + d_{\mathsf{fp}}$.*

Proof. Assume that i invokes an operation at time t, resulting in a geocast at time t by the VOE-Client associated with i; this geocast sends the invocation to the VOE-Servers for h. Let $t' > t$ be the time specified by the GeoCast *reliable delivery* property.

The first key claim is that at time t' there is some replica $j \in I$ that has joined h prior to time t' and does not fail until after time $t' + \epsilon_{\mathsf{geo}}$. We show that this follows from the definition that a virtual object is correct, i.e., that it is populated.

Specifically, the assumption that h is correct implies that there exists some $j_1, j_2, \ldots$ such that each $j_{\ell+1}$ enters the inner region of h at least time t_{join} prior to j_ℓ departing, and remains well inside h without failing until at least t_{join} after j_ℓ departs. Since the join protocol requires two broadcasts using the fpcast service, and since $j_{\ell+1}$ enters at least t_{join} prior to j_ℓ departing, it is easy to see (inductively) that each $j_{\ell+1}$ completes the join protocol and becomes active at least ϵ_v prior to j_ℓ departing.

We can thus conclude that for the interval of time $[t', t' + \epsilon_{\mathsf{geo}}$, either some replica j_ℓ resides well inside the focal point region throughout the specified interval, or j_ℓ leaves the focal point region during the specified interval. In the first case, we have identified a replica, as claimed; in the latter case, by assumption, there exists some $j_{\ell+1}$ that enters at least time t_{join} prior to j_ℓ departing, and has has joined at least time ϵ_{geo} prior to j_ℓ departing. In this latter case, j_ℓ remains well inside h until at least ϵ_{geo} after j_ℓ departs, which is at least ϵ_{geo} after t'.

The theorem then follows immediately by simply summing the message latencies involved in processing an invocation. The invoked operation is immediately sent by the GeoCast service to focal point h, at which point it is received time d_{geo} later by some replica for h, since h is correct. Immediately it is rebroadcast using the fpcast_h service, at which point it is received time d_{fp} later by some replica for h. Immediately, the replica sends a response to i, which receives the response time d_{geo} later. □

Chapter 5

Emulating Virtual Nodes

In this chapter we present an algorithm that implements the Virtual Node Layer in a wireless ad hoc network. We begin with a brief review of the Virtual Node Layer, and then present the Virtual Node Emulator (VNE), an algorithm for emulating the Virtual Node Layer, in Section 5.1. In Section 5.2, we analyze the Virtual Node Emulator and show that it is a safe emulation of the Virtual Node Layer. In Section 5.3, we examine the performance of the emulator and show that every execution of the emulation also satisfies the liveness properties of the Virtual Node Layer (specifically, the liveness properties of the virtual GeoCast service). The material in this chapter is based on a protocol published in the 18th International Symposium on Distributed Computing (DISC 2004) [31]. The main difference is that in [31], mobile nodes communicate only via local broadcast, and in the Virtual Node Layer, clients and virtual nodes communicate only via local broadcast. Here, mobile nodes communicate via a GeoCast service, and clients and virtual nodes communicate via a virtual GeoCast service. This choice ensures that the primary difference between Chapters 4 and 5 is the type of computation that occurs at the virtual entity, not the type of communication. Moreover, the virtual GeoCast simplifies the use of virtual nodes since clients can communicate with virtual nodes that are not nearby. In Chapter 7, we discuss modifications that support this local-only communication.

Recall from Chapter 3 that the Virtual Node Layer contains three types of entities:

- *clients*, each of which is associated with a mobile node;
- *virtual nodes*, each of which is associated with a focal point;
- *RealWorld*, containing a R_{Vgeo}-*GeoCast* service virtual-geocast, which performs communication between the clients and virtual nodes; the RealWorld tracks the location and failure status of each node (real or virtual), and provides clients with updates as to the real time and their current location.

In the Virtual Object Layer, clients interacts directly with (virtual) objects; by contrast, in the Virtual Node Layer, clients and virtual nodes interact with each other. In the former, clients *invoke* operations and (virtual) objects *respond*; in the latter, clients and virtual nodes use a virtual-geocast service to *geocast* messages to each

other. Notice that a virtual node is more powerful than a virtual object in that it can initiate actions on its own and it can coordinate directly with other virtual nodes.

The algorithm implementing the Virtual Node Layer is, in many ways, a natural extension of the protocol presented in Chapter 4. As before, each mobile node acts as a replica for nearby virtual entities: when a mobile node enters a focal point, it joins the emulation for the associated virtual node; when it leaves the focal point, it leaves the emulation. If a focal point becomes depopulated, then the associated virtual node fails. In contrast to the virtual objects in Chapter 4, if the focal point becomes populated again, then the associated virtual node recovers.

As in Chapter 4, the key to maintaining the consistency of the replicas is the totally-ordered broadcast service fpcast, which guarantees that within a focal point messages are delivered reliably and consistently in a well-defined order. The local broadcast service is used to implement a type of replicated state machine, one that tolerates joins and leaves of mobile nodes.

Despite the similarities between the material in this chapter and Chapter 4, the material in this chapter is complete and can be read independently of Chapter 4.

5.1 Virtual Node Emulator

In this section we present the Virtual Node Emulator (VNE), an algorithm that implements the Virtual Node Layer (described in Section 3.2). The basic idea of the algorithm is the same as the Virtual Object Emulator presented in Chapter 4: The Virtual Node Emulator uses the mobile nodes in each focal point region as replicas for a virtual node associated with that focal point. Throughout an execution, mobile nodes move in and out of the focal point regions. The nodes that are inside the focal point region at any given time act as replicas, and collaborate to implement the virtual node. They take advantage of the fpcast service associated with the focal point to implement a replicated state machine that tolerates nodes continually joining and leaving. This replicated state machine consistently maintains the state of the virtual node.

As a slight abuse of notation, throughout this section, we refer to the virtual node by the name of the focal point with which it is associated; thus we often refer to some focal point $v \in O$, and then later refer to the automaton for virtual node v, meaning the automaton for the virtual node associated with that particular focal point.

In Chapter 4, we presented an algorithm for implementing a single virtual object, and then noticed that by composition, this algorithm extended immediately to an emulator for many virtual objects. For the purpose of description, we continue in this manner, describing how to implement a single virtual node $v \in O$. Formally, however, the pseudocode for the algorithm is specified for the entire set of virtual nodes, one for each focal point in the set O. This is because the virtual nodes communicate with each other via the virtual GeoCast service, and hence we need to emulate this interaction.

The Virtual Node Emulator consists of two automata that run on each mobile node. The VNE is designed to emulate a virtual node; it is not involved in emulating

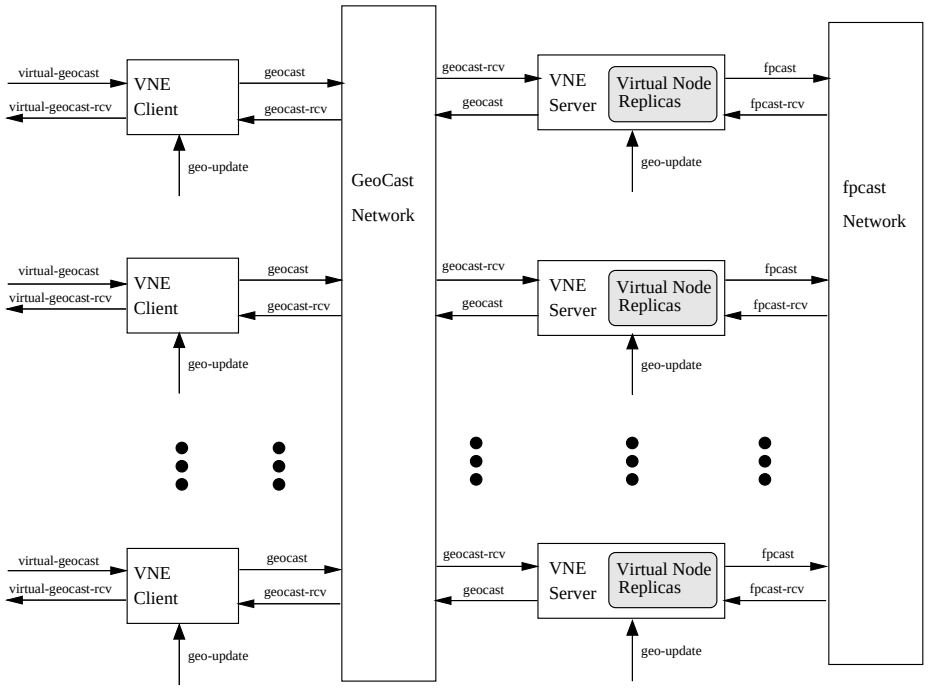

Figure 5-1: Diagram of the Virtual Node Emulator for emulating virtual nodes in O. Each of the VNE-Clients and VNE-servers may execute on a different mobile node. When a VNE-Client (on the left) receives a virtual-geocast(m, ℓ) from a client (not pictured), it sends the message m to all the VNE-Servers in the destination region near ℓ using the underlying GeoCast service. If there is a virtual node v in that region, i.e., near location ℓ, then at least one of the VNE-Servers in the focal point region for v broadcasts the request to all the other VNE-Servers in the focal point using the fpcast$_v$ service. Each VNE-Server then updates its replica of virtual node v's state, simulating the virtual node v receiving the message in question. The VNE-Servers can also propose other actions for a virtual node by broadcasting the proposal using the fpcast$_v$ service. Whenever a VNE-Server receives a proposal, it simulates the virtual node taking that action (if it is possible for the virtual node to take that action).

the client. The emulator consists of two components: the **VNE-Client** and the **VNE-Server**, which communicate with each other using the GeoCast service. Notice that both components of the emulator, though executing on the same mobile node, interact only via the GeoCast service. Each mobile node executes one VNE-Client, and one VNE-Server, both of which are together responsible for emulating all the virtual nodes.

The Virtual Node Emulator is presented schematically in Figure 5-1. The clients (not depicted) pass virtual GeoCast messages to the VNE-Clients, which attach a tag to every message and broadcast the message to a focal point using the GeoCast service. Within the focal point, the request is received by the VNE-Servers. The VNE-Servers then coordinate amongst themselves to implement the input, internal and output transitions, and determine an ordering of the virtual node's transitions. They accomplish this coordination using the fpcast service, which delivers messages to all of the VNE-Servers in the same order. This total ordering determines the order in which the VNE-Servers simulate the virtual node's transitions. Based on this (ordered) sequence of events, the VOE-Servers consistently update their local replicas of the virtual node. When a VNE-Server simulates the virtual node sending a GeoCast message, the VNE-Server sends the message on behalf of the virtual node. When the VNE-Client receives a GeoCast message, it delivers it to the client.

5.1.1 VNE-Client

The pseudocode for the VNE-Client automaton is presented in Figures 5-2 and 5-3. The VNE-Client acts as a filter between the client and the GeoCast service: Whenever the client performs a virtual-geocast, the VNE-Client translates this event into a geocast in the physical system, attaching a unique identifier. Whenever the VNE-client receives a geocast message from the physical system, it translates it into a virtual-geocast-rcv for the client. Unlike in the case of the Virtual Object Emulator, it does not need to remove duplicates, as there is no requirement that the virtual GeoCast service deliver messages only once.

5.1.2 VNE-Server

The VNE-Server performs the actual replicated emulation of the virtual nodes. Here we describe the behavior with respect to one particular virtual node $v \in O$. The VNE-Server implements a replicated state machine, enhanced to perform internal and output actions. Each mobile node, when it is active in the emulation, maintains a copy of the state of the virtual node. The mobile nodes use the total-ordering property of the $\mathsf{fpcast}_{\mathsf{v}}$ service to ensure that the replicated state is updated consistently. Figure 5-4 contains the signature and state of the VNE-Server automaton. The remaining pseudocode for the VNE-Server automaton for node i is in Figures 5-5 and 5-6. Much of the description that follows is identical to the VOE-Server description in Chapter 4; we call attention to the key differences.

Emulator State. We begin by enumerating the elements of the VNE-Server's state:

Figure 5-2: Automaton VNE-Client: Signature and State
for client i

1 **Input**:
2 virtual-geocast$(m, d)_i$, $m \in msgs$, $d \in L$
3 geocast-rcv$(\langle m, oid \rangle, d)_i$, $m \in msgs$, $oid \in U$, $d \in L$
4
5 **Output**:
6 virtual-geocast-rcv$(\langle m, oid \rangle, d)_i$, $m \in msgs$, $oid \in U$, $d \in L$
7 geocast$(m, d)_i$, $m \in msgs$, $d \in L$
8
9 **State**:
10 $count \in \mathbb{N}$, the current count of unique ids, initially 1
11 *ready-msgs*, a set of GeoCast messages, initially $\emptyset$
12 *geocast-queue*, a queue of messages to be geocast, initially $\emptyset$

Figure 5-3: Automaton VNE-Client: Transitions
for client i

```
1   Input virtual-geocast(m, d)_i
2   Effect:
3       new-oid ← ⟨count, i⟩
4       count ← count +1
5       Enqueue(geocast-queue, ⟨⟨geo, m, oid⟩, d⟩)
6
7   Input geocast-rcv(⟨geo, m, oid⟩, d)_i
8   Effect:
9       if (⟨m, oid⟩ ∉ delivered-oids ∪ ready-msgs) then
10          ready-msgs ← ready-msgs ∪ {⟨m, d⟩}
11
12  Output geocast(m, d)_i
13  Precondition:
14      Peek(geocast-queue) = ⟨m, d⟩
15  Effect:
16      Dequeue(geocast-queue)
17
18  Output virtual-geocast-rcv(m, d)_i
19  Precondition:
20      ⟨m, d⟩ ∈ ready-msgs
21  Effect:
22      ready-msgs ← ready-msgs − {⟨m, d⟩}
23      delivered-oids ← delivered-oids ∪ {⟨m,d⟩}
```

Figure 5-4: Automaton VNE-Server: Signature and State
for client i

Input:
- geocast-rcv($\langle m, oid\rangle, d)_i$, $m \in msgs$, $oid \in T$, $d \in L$
- fpcast-rcv($\langle$join-req, jid, $vnid\rangle)_{v,i}$, $jid \in T$, $vnid \in O$, $v \in O$
- fpcast-rcv($\langle$join-ack, jid, $vnid$, val, $r\text{-}id$, $c\text{-}ops\rangle)_{v,i}$, $jid \in T$, $vnid \in O$, $val \in V_v$, $r\text{-}id \in T$, $c\text{-}ops$ a set, $v \in O$
- fpcast-rcv($\langle act, vnid, oid\rangle)_{v,i}$, $act \in actions_{vnid}$, $vnid \in O$, $oid \in T$, $v \in O$
- geo-update($l, t)_i$, $l \in L$, $t \in R^{>0}$

Output:
- geocast($\langle m, oid\rangle, d)_i$, $m \in msgs$, $oid \in T$, $d \in L$
- fpcast($\langle$join-req, jid, $vnid\rangle)_{v,i}$, $jid \in T$, $vnid \in O$, $v \in O$
- fpcast($\langle$join-ack, jid, $vnid$, val, $r\text{-}id$, $c\text{-}ops\rangle)_{v,i}$, $jid \in T$, $vnid \in O$, $val \in V$, $r\text{-}id \in T$, $c\text{-}ops$ a set of ops , $v \in O$
- fpcast($\langle act, vnid, oid\rangle)_{v,i}$, $act \in actions_{vnid}$, $vnid \in O$, $oid \in T$, $v \in O$
- fpcast($\langle$reset, $vnid$, $oid\rangle)_{v,i}$, $vnid \in O$, $oid \in T$, $v \in O$

Internal:
- join()$_{v,i}$, $v \in O$
- initiate-op(act, $v)_i$
- initiate-reset($v)_i$, $v \in O$
- simulate-op(act, v, $oid)_i$, $act \in actions[v]$, $v \in O$, $oid \in T$
- process-join-req(jid, $vnid)_i$, $jid \in T$, $vnid \in O$
- process-join-ack($v)_i$, $v \in O$
- process-reset(v, $oid)_i$, $v \in O$, $oid \in T$

Constants:
- $fp\text{-}center_v : \mathbb{R}^{\geq 0} \to L$, a function mapping time to the focal point center for $v \in O$.
- $fp\text{-}region_v : \mathbb{R}^{\geq 0} \to \mathcal{P}(L)$, a function mapping time to the focal point region for $v \in O$.
- $actions_v$, $v \in O$, the set of actions for virtual node v
- $input_v$, $v \in O$, the set of input actions for virtual node v
- $output_v$, $v \in O$, the set of output actions for virtual node v
- $internal_v$, $v \in O$, the set of internal actions for virtual node v
- δ_v, $v \in O$, the transition function for virtual node v
- $v_{\mathsf{max}} \in \mathbb{R}^{\hat{}}\{\geq\ 0\}$, the maximum speed of a mobile node
- $v(v)_{\mathsf{fp\text{-}max}} \in \mathbb{R}^{\hat{}}\{\geq\ 0\}$, $v \in O$, the maximum speed of virtual node v
- $msgs$, a message alphabet containing all the messages sent by virtual nodes in the set O

State:
- $clock \in R^{\geq 0}$, the current time, initially 0, updated by the geosensor
- $location \in L$, node i's location, initially i's initial location, updated by the geosensor
- $status_v \in$ {idle,joining,listening,active}, $v \in O$, initially active if node is in the focal point region; idle otherwise
- $val_v \in V$, $v \in O$, holds current value of the simulated atomic object, initially v_0
- $join\text{-}id_v \in T$, $v \in O$, unique id for current join request, initially $\langle 0, i_0\rangle$
- $fpcast\text{-}queue_v$, $v \in O$, a queue of messages to be sent by fpcast, initially $\emptyset$
- $geocast\text{-}queue_v$, $v \in O$, a queue of messages to be sent by the GeoCast, initially $\emptyset$
- $anwered\text{-}join\text{-}reqs_v$, $v \in O$, set of ids of join requests that have already been answered, initially $\emptyset$
- $pending\text{-}ops_v$, $v \in O$, queue of operations waiting to be simulated, initially $\emptyset$
- $completed\text{-}ops_v$, $v \in O$, set of operations that have been simulated, initially $\emptyset$
- $fpcast\text{-}count \in \mathbb{N}$, counter of fpcast operations, initially 1
- $op\text{-}counter \in \mathbb{N}$, initially 0, a counter for operations initiates by i
- $reset\text{-}id_v \in T$, identifier of last reset, initially $\langle -1, i_0\rangle$

Figure 5-5: Automaton VNE-Server: Input Transitions for client i

```
1  Input fpcast-rcv(⟨⟨join-req, jid, vnid⟩, cnt, k⟩)_{v,i}
2  Effect:
3      if (v = vnid) then
4          if ((status_v = joining) ∧ (jid = join-id_v)) then
5              status_v ← listening
6          else if ((status_v = listening ∨ active) then
7              pending-ops_v ← pending-ops_v ∪ {⟨join-req, jid, vnid⟩}
8
9  Input fpcast-rcv(⟨⟨join-ack, jid, vnid, val, c-ops, r-id⟩, cnt, k⟩)_{v,i}
10 Effect:
11     if (v = vnid) then
12         answered-join-reqs_v ← answered-join-reqs_v ∪ {jid}
13         if ((status_v = listening) ∧ (jid = join-id_v)) then
14             status_v ← active
15             completed-ops ← completed-ops ∪ c-ops
16             val_v ← val
17             completed-ops ← completed-ops ∪ ⟨join-ack, jid, vnid, val, c-ops⟩
18             reset-id_v ← r-id
19         else
20             pending-ops_v ← pending-ops_v ∪ {⟨join-ack, jid, vnid, val, c-ops⟩}
21
22 Input fpcast-rcv(⟨⟨act, vnid, r-id, oid⟩, cnt, k⟩)_{v,i}
23 Effect:
24     if (v = vnid) then
25         if ((status_v = listening ∨ active)
26                 ∧
27         (⟨act, vnid, oid⟩ ∉ pending-ops_v ∪ completed-ops_v)) then
28                 Enqueue(pending-ops_v, ⟨act, vnid, r-id, oid⟩)
29
30 Input fpcast-rcv(⟨⟨reset, vnid, oid⟩, cnt, k)_{v,i}
31 Effect:
32     if (v = vnid) then
33         if ((status_v = listening ∨ active) then
34                 Enqueue(pending-ops_v, ⟨reset, v, oid⟩)
35
36 Input geocast-rcv(⟨geo, m, oid⟩, d)_i
37 Effect:
38     for every v ∈ O do
39         if (status_v ≠ idle) then
40             if (⟨virtual-geocast-rcv(m), v, oid⟩ ∉ pending-ops_v ∪ completed-ops_v) then
41                 Enqueue(fpcast-queue_v, ⟨virtual-geocast-rcv(m), v, reset-id, oid⟩)
42
43 Input geo-update(l, t)_i
44 Effect:
45     clock ← t
46     location ← l
47     for every v ∈ O do
48         if (depth_v(location, clock) < t_upd ·(v_max + v_fp-max)) then
49             status_v ← idle
50             join-id_v ← ⟨0, i_0⟩
51             val_v ← v_0
52             answered-join-reqs_v ← ∅
53             pending-ops_v ← ∅
54             fpcast-queue_v ← ∅
55             geocast-queue_v ← ∅
56
57 Internal join()_{v,i}
58 Precondition:
59     depth_v(location, clock) ≥ t_upd ·(v_max + v_fp-max)
60     status_v = idle
61 Effect:
62     join-id_v ← ⟨clock, i⟩
63     status_v ← joining
64     Enqueue(fpcast-queue_v, ⟨join-req, join-id_v⟩)
```

Figure 5-6: Automaton VNE-Server**: Internal Transitions** for client i

Internal initiate-op(act, v)$_i$
Precondition:
  $status_v$ = active
  $\delta(act, val_v)_v \neq \perp$
  $act \in internal_v \wedge act \in output_v$
Effect:
  $oid \leftarrow \langle op\text{-}counter, i\rangle$
  $op\text{-}counter \leftarrow op\text{-}counter + 1$
  Enqueue($fpcast\text{-}queue_v$, $\langle act, v, reset\text{-}id, oid\rangle$)

Internal simulate-op(act, v, $r\text{-}id$, oid)$_i$
Precondition:
  $status_v$ = active
  Peek($pending\text{-}ops_v$) = $\langle act, v, r\text{-}id, oid\rangle$
Effect:
  if ($\langle act, v, oid\rangle \notin completed\text{-}ops_v$) **then**
    if ($r\text{-}id = reset\text{-}id_v$) **then**
      if ($\delta(act, val_v) \neq \perp$) **then**
        $val_v \leftarrow \delta(act, val_v)$
        if (act = virtual-geocast(m, d)) **then**
          Enqueue($geocast\text{-}queue_v$, $\langle\langle m, oid\rangle, d\rangle$)
  Enqueue($completed\text{-}ops_v$, **Dequeue**($pending\text{-}ops_v$))

Internal process-join-req(jid, $vnid$)$_i$
Precondition:
  $status_v$ = active
  Peek($pending\text{-}ops_v$) = $\langle$join-req, jid, $vnid\rangle$
Effect:
  if ($jid \notin answered\text{-}join\text{-}reqs_{vnid}$) **then**
    Enqueue($fpcast\text{-}queue_{vnid}$, $\langle$join-ack, jid, $vnid$, val_{vnid}, $reset\text{-}id_{vnid}$, $completed\text{-}ops_{vnid}\rangle$)
  Dequeue($pending\text{-}ops_{vnid}$)

Internal process-join-ack(v)$_i$
Precondition:
  $status_v$ = active
  Peek($pending\text{-}ops_v$) = $\langle$join-ack, $\ldots\rangle$
Effect:
  Dequeue($pending\text{-}ops_v$)

Internal initiate-reset(v)$_i$
Precondition:
  $location \in fp\text{-}region(clock)_v$
  $status_v \neq$ active
Effect:
  $oid \leftarrow \langle op\text{-}counter, i\rangle$
  $op\text{-}counter \leftarrow op\text{-}counter + 1$
  Enqueue($fpcast\text{-}queue_v$, $\langle$reset, v, $oid\rangle$

Internal process-reset(v, oid)$_i$
Precondition:
  Peek($pending\text{-}ops_v$ = $\langle$reset, v, $oid\rangle$
Effect:
  Dequeue($pending\text{-}ops_v$)
  if ($location \in fp\text{-}region(clock)_v$ **then**
    $status \leftarrow$ active
    $val_v \leftarrow v_0$
    $fpcast\text{-}queue_v \leftarrow \emptyset$
    $geocast\text{-}queue_v \leftarrow \emptyset$
    $reset\text{-}id \leftarrow oid$

Figure 5-7: Automaton VNE-Server: Output Transitions for client i

126 **Output** fpcast($\langle m,\ \mathit{fpcast\text{-}count},\ i\rangle$)$_{v,i}$
127 **Precondition**:
128 **Peek**($\mathit{fpcast\text{-}queue}_v$) $= m$
129 **Effect**:
130 $\mathit{fpcast\text{-}count} \leftarrow \mathit{fpcast\text{-}count} + 1$
131 **Dequeue**($\mathit{fpcast\text{-}queue}_v$)
132
133 **Output** geocast($m,\ d$)$_i$
134 **Precondition**:
135 $v \in O$
136 **Peek**($\mathit{geocast\text{-}queue}_v$) $= \langle m,\ d\rangle$
137 **Effect**:
138 **Dequeue**($\mathit{geocast\text{-}queue}_v$)

- The *clock* and the *location* variables store the time and location information from the most recent geo-update.
- The *status* determines whether the emulator is currently participating in the emulation as a replica, joining the emulation, or idle.
- The value *val* stores the current state of the replica.
- The *join-id* is a unique identifier selected and used during the join protocol.
- The *fpcast-queue* and *geocast-queue* are two queues storing outgoing messages for the respective broadcast services.
- The *answered-join-reqs* is a set indicating which join requests have already been answered.
- The queue *pending-ops* stores messages waiting to be processed.
- The queue *completed-ops* stores messages that have already been processed.
- The counter *fpcast-cnt* is used to ensure that each fpcast message is unique[1].
- The counter *op-counter* is used when simulated internal and output events to assign a unique identifier to each event.
- The identifier *reset-id* is used to uniquely identify each time the virtual node is reset.

For many of these state elements, the emulator maintains one copy for each $v \in O$. For example, if v_1 and v_2 are two different virtual nodes, there exists state elements $\textit{pending-ops}_{v_1}$ and $\textit{pending-ops}_{v_2}$. Each element is used only in the emulation of the specified virtual node. Some, such as the *clock*, *location*, *fpcast-count*, and *op-counter* are used in the emulation of all the virtual nodes. This distinction is not critical; the elements are not duplicated when it is clear that this causes no additional complications.

Joining and Leaving the Emulation. The emulator transitions between three statuses: idle, joining, and active. We first discuss how a mobile node joins the emulation. The join protocol is activated when the mobile node enters the inner region of a focal point v, i.e., when it hears that it is well inside the focal point region. In this circumstance, the join transition is enabled (Figure 5-6, lines 57–64). This results in the mobile node broadcasting a *join-request* message using the fpcast_v service, at which point it sets its status to joining and awaits a response.

The other active nodes in the focal point receive the join request (Figure 5-5, lines 1–7) and respond by sending the current replicated state of the virtual object

[1]Up until this point, the emulator state has been equivalent to that of the VOE-Server. The remaining elements are specific to the VNE-Server

using the fpcast_v service (Figure 5-6, lines 89–96). Notice that this is slightly different than in the case of the Virtual Object Emulator, as the join request is processed in the order received using the *pending-ops* queue. This is due to the need in this case to order the join request with respect to reset messages that may reset the virtual node.

As an optimization to avoid unnecessary message traffic, if a node observes (by examining *answered-join-reqs*) that someone else has already responded to a join request, then it does not respond; only one node needs to send a join response. Notice (Figure 5-5, line 91) that only active nodes will send join responses. When the node that originated the join request receives its own join request (Figure 5-5, lines 4–5), it sets its status to listening.

When the joining node receives a response to its join request (Figure 5-5, lines 9–20), it starts participating in the emulation, setting its status to active. This completes the join protocol.

By contrast, ceasing to participate in the emulation does not require any coordination with other replicas. When a node hears that it has left the inner focal point region (either due its own motion, or the movement of the virtual node), it re-initializes its variables, and becomes idle (Figure 5-5, lines 43–55).

Simulating Input, Output, and Internal Actions. We next discuss how active replicas simulate the virtual node's transitions. When a node participating in emulating virtual node v receives a GeoCast message containing a message to deliver to virtual node v (Figure 5-5, lines 36–41), it broadcasts a proposal that the virtual node simulate a virtual-geocast-rcv of that message; it broadcasts this proposal using the $\mathsf{fpcast}_\mathsf{v}$ service, thus ordering the virtual-geocast-rcv event for that message with respect to the other messages broadcast with the $\mathsf{fpcast}_\mathsf{v}$ service (i.e., join-request messages, join responses, and other virtual node transitions). Notice that this proposal includes the identifier for the most recent reset, i.e., the *reset-id*, which allows nodes receiving the proposal to determine whether the virtual node was reset between when the transition was proposed and when it was received. (This is different from the Virtual Object Emulator, which does not include resets.)

Since it is possible that a GeoCast message is received—and hence rebroadcast—by more than one node in the focal point, there is some bookkeeping to ensure that only one copy of the same invocation is actually processed by the nodes. We include an optimization that if a node observes via the *pending-ops* or *completed-ops* queues that an invocation has already been sent by the $\mathsf{fpcast}_\mathsf{v}$ service, then it does not do so again.

Replicas also simulate other types of transitions that the virtual nodes may execute, that is, its locally controlled actions. (This is different from the Virtual Object Emulator, which only needs to process operation invocations.) At any time, a replica may examine its local state and determine that some particular locally-controlled action is enabled. In this case, it broadcasts a proposal using the fpcast_v service suggesting that this transition be executed (Figure 5-6, lines 66–74). Again, notice that the proposal includes the *reset-id*.

Active nodes keep track of messages received via the $\mathsf{fpcast}_\mathsf{v}$ service in the order in which they are received (Figure 5-5, lines 22–28); this orders the various input, output, and internal actions to be simulated for virtual node v. Duplicates are discarded using the unique operation identifiers, which are stored in the sets *pending-ops* and *completed-ops*. The transitions are then simulated (Figure 5-6, lines 76–87) on the replicated state in this order, as long as the *reset-id* included in the proposal matches the *reset-id* of the emulator. Specifically, for input actions (e.g., virtual-geocast-rcv transitions), the transition is always simulated, since input actions are always enabled. For internal and output actions, the transition is simulated if it is still enabled. Notice, though, that it is possible that intervening events might have disabled this transition after it was originally proposed, and hence it cannot be simulated. Whenever a replica simulates a virtual-geocast of some message m, the replica adds the message m to its outgoing geocast-queue, resulting in the GeoCast of the specified message (Figure 5-6, line 86).

There are some subtle points involved in deciding when during the process of joining the emulation a node should start recording invocation messages, when a node should start executing operations, and when a node should start sending responses. During the process of joining, a node receives a snapshot of the object's state. However by the time the snapshot is received, it might be out of date, since there may have been some intervening messages delivered by the $\mathsf{fpcast}_\mathsf{v}$ service that have been received since the snapshot was sent. Therefore the joining node must record all the operation invocations that are broadcast after its join request was broadcast but before it receives the snapshot. This is accomplished by having the joining node enter a "listening" state once it receives its own join-request message; all invocations received when a node is in either the listening or the active state are recorded, and actual processing of the invocations can start once the node has received the snapshot and has the active status. The processing consists of first emptying the queue of waiting invocations, and then proceeding to handle newly arrived invocations.

A final point to notice is that every message sent by the fpcast_v service is unique: attached to each message is a counter (*fpcast-cnt*) and the process identifier; each processor increments its counter after each fpcast, ensuring that no message is duplicated. Notice that each message m that is fpcast is of the form $\langle P, cnt, k\rangle$, where cnt is a counter and k is a node identifier. We refer to P as the **payload** of m.

Resetting the Virtual Node. Sometime, particularly when the virtual node has failed due to depopulation, it may be useful for the virtual node to reset itself. In particular, if a node does not receive a response during the join protocol, it can initiate a reset (Figure 5-6, lines 105–112). In this case, it broadcasts a reset message using the fpcast service. This reset message contains a unique identifier that is used to name each successful reset event. Each proposed transition for the virtual node includes the identifier of the most recent reset; if a more recent reset has occurred in the interim since the transition was proposed, then the proposal is ignored.

When a node receives a reset message, if it is well inside the focal point region, then it processes the reset message in order from the *pending-ops* queue, sets its status

to active, and resets its emulator state, including the replicated state *val* of the virtual node.

5.2 Analysis: Safety of the Emulation

In this section we show that the Virtual Node Emulator ensures a *safe* emulation of the Virtual Node Layer, meaning that every trace of the emulator, when restricted to the clients actions, is a trace of the Virtual Node Layer. The performance of the emulator is considered separately in Section 5.3. We begin be reviewing what it means for the protocol to ensure a safe emulation.

For each $i \in I$, let c_i be a client automaton, and let C be the composition of all the client automata. Let A_v be a virtual node automaton, specifically, the automaton designated to execute on virtual node v. When describing A_v, let δ_v be the transition relation associated with automaton A_v. Recall that automaton A_v is deterministic, and hence for every state s and every action a, there is at most one state s' such that $\langle s, a, s' \rangle \in \delta_v$.

Let S be the physical system consisting of the client automata c_i, each composed with (1) a VNE-Client automaton and (2) a VNE-Server automaton. Let U represent the environment, i.e., the RealWorld automaton containing the geocast and fpcast_v services. Thus the automaton $S \times U$ describes the underlying physical system executing a set of clients along with the Virtual Node Emulator.

Recall that the virtual node layer consists of three types of automata: clients, virtual nodes, and the RealWorld automaton, containing the virtual GeoCast service. Let S' be the subsystem of the Virtual Node Layer consisting of the set of clients c_i, and the set of virtual nodes O. Let U' be the virtual RealWorld automaton, which includes the virtual GeoCast service. Thus the automaton $S' \times U'$ describes the Virtual Node Layer.

We want to show that every trace of $S \times U$, when restricted to the external actions of the clients C, is a trace of $S' \times U'$, restricted to the external actions of the clients C. (Recall that a trace is a timed sequence of externally-visible events.) Formally, we first define a set H containing all the hidden actions in $S \times U$: let H be the entire set of externally-visible actions for $S \times U$, excluding the externally-visible actions for C. Similarly we define a set H' containing all the hidden actions in $S' \times U'$: let H' be the entire set of externally-visible actions in $S' \times U'$, excluding the externally-visible actions for C.

We say that the emulation is **safe** if for every admissible execution α of $S \times U$, there exists an execution γ of $S' \circ U'$ such that

$$trace(hide_{H'}(\gamma)) = trace(hide_{H}(\alpha)) \ .$$

In this case, the clients cannot distinguish whether they are executing in the virtual system or the real system.

For the purpose of this section, fix an arbitrary admissible execution α of system $S \times U$. The goal of this section is to demonstrate an execution γ of $S' \times U'$ satisfying

the requisite trace indistinguishability.

We first construct individual executions for the components of system S', that is, the clients (Section 5.2.1) and virtual nodes (Section 5.2.2). These individual executions are constructed independently, based on the execution α of the underlying physical system. We then construct a trace of the composed system S' that is consistent with each of the individual executions (Section 5.2.3). Finally, we invoke a pasting lemma to construct a single execution of the composed system S' (Section 5.2.4). The resulting execution defines the behavior of the clients and virtual nodes in the virtual system.

It remains to consider the virtual RealWorld automaton and the behavior of the virtual GeoCast service, i.e., U' (Section 5.2.5). Recall that the virtual RealWorld automaton is defined as an automaton that produces every trace satisfying *integrity*, while maintaining the real time and the location of each client, along with a restriction on traces that guarantees *reliable delivery*. To this point, we have constructed an execution $\gamma_{S'}$ of system S'; this execution implies a trace of the virtual RealWorld automaton (and the virtual GeoCast service) U'. We show that this trace satisfies the integrity property, and thus that there is an execution of U' consistent with this trace. A second invocation of the pasting lemma produces an execution γ of the entire virtual system $S' \times U'$, and completes the proof that the emulation is safe.

5.2.1 Client Executions

We begin by constructing an execution γ_i for each client $i \in I$. Specifically, we define γ_i to be the restriction of α to the actions of client i. Formally, let $actions_i$ be the set of actions for client c_i; then $\gamma_i = \alpha | actions_i$.

5.2.2 Virtual Node Executions

We now construct an execution γ_v for each virtual node $v \in O$. For the rest of Section 5.2.2, fix some virtual node $v \in O$.

First, we define a set of events $IM(\alpha)_v$ that occur in execution α. Second, we define a total order on these events. Third, we construct an execution of virtual node v derived from this total order.

We then show an important property of this execution: the state of each replica at each point in α reflects exactly the events described in $IM(\alpha)_v$ as ordered according to the total order that we have defined in this proof. This construction and this proof is analogous to the construction in Sections 4.2.1 and 4.2.2.

Defining the Set of Operations. For execution α, let $IM(\alpha)_v$ be the set of events for virtual node v simulated by the VNE-Server automata, along with the set of reset events. Specifically:

$$\begin{aligned} IM(\alpha)_v &= \{\langle \mathsf{act}, v, oid\rangle : \exists j \in I \text{ such that } \mathsf{simulate\text{-}op}(act, v, oid)_j \text{ occurs in } \alpha\ \} \\ &\cup\ \{\langle \mathsf{reset}, v, oid\rangle : \exists j \in I \text{ such that } \mathsf{process\text{-}reset}(v, oid)_j \text{ occurs in } \alpha\ \}\ . \end{aligned}$$

Thus, for each operation simulated in α and for each reset event, there is one element $\langle \mathsf{act}, v, oid \rangle$ in the set $IM(\alpha)_v$.

Defining a Total Order. Notice that each element in the set $IM(\alpha)_v$ is associated with at least one message $m = \langle \langle \mathsf{act}, v, oid \rangle, \cdot, j \rangle$ or $m = \langle \langle \mathsf{reset}, v, oid \rangle, \cdot, j \rangle$ where for some $j \in I$ there is an $\mathsf{fpcast}(m)_j$ event in α: the precondition of simulate-op ensures that each simulated event is drawn from the queue *pending-ops*$_v$; the precondition of process-reset also ensures that each reset event is drawn from the queue *pending-ops*$_v$. This queue is populated with messages received from the fpcast_v service. We say that an event $\langle \mathsf{act}, v, oid \rangle$ is associated with some message m if the event can be traced to the reception of message m, which added the event to the *pending-ops*$_v$ queue, and then was processed/simulated.

The total-ordering property of the $\mathsf{fpcast}_\mathsf{v}$ service guarantees that, since every fpcast message is unique, there exists a total ordering of the messages that is consistent with the order in which each mobile node receives the messages. That is, if node i receives two messages m_r and m_t in that order, then $m_r \prec m_t$ in the total ordering. Thus, we order the events in $IM(\alpha)_v$ according to the total ordering on their associated messages: fix $\prec_v$ to be this ordering. (We will drop the subscript v where it is clear by context that we are referring to messages from this particular fpcast_v service.)

Defining an Execution. We define execution γ_v based on the events in $IM(\alpha)$. We first construct a non-timed version of γ_v inductively as is described in Figure 5-8, and then we insert appropriate time-passage events. In this construction, we iterate over the messages in $IM(\alpha)_v$, considering the messages that represent either an action for v or a reset request. We apply these actions/resets in the specified total order, omitting those that are not enabled, and inserting fails before recoveries.

More formally, we proceed as follows. For each message proposing an action, if that action is enabled, we add a transition in δ_v to the end of the execution; that is, we apply the action to the state at the end of the execution. Since the virtual node is deterministic, we can uniquely identify the behavior of each action. For each message proposing a reset, we add a fail transition to the end of the execution; that is, we apply a fail action to the state at the end of the execution. We then add a reset action, which returns the virtual node to its initial state. Finally, if there are no more messages to consider, and if in execution α there are eventually no active emulators, then we add a final fail event to the end of the execution. Notice also that for each message proposing a virtual-geocast-rcv for some message m with an associated unique identifier u, we consider only the first such message.

To this point, there are no trajectories in execution γ_v. For each event in γ_v we fix a time at which that event occurs, and then add appropriate trajectories in between the discrete events. Specifically, for each event e in γ_v that was added as a result of a simulate-op event in α, associate event e with the time at which the first such simulate-op event occurs in α. For each reset event e in γ_v, associate with event e the same time as the process-reset event in α that resulted in the reset event being added to the execution. For each fail event e in γ_v, associate with event e the same time as

Figure 5-8: Construction of Virtual Node Execution γ_v

1. Initially, set $\gamma_v = s_0$, the initial state of virtual node v.
2. Define the set S as follows:

$$S = \{\langle P, cnt, k\rangle : \ P \in IM(\alpha)_v, \text{ and } \langle P, cnt, k\rangle \text{ is the } \mathsf{fpcast} \text{ message associated with } P\} .$$

3. Repeat while $S \neq \emptyset$:
   (a) Let m be the minimum message in S, according to the total order $\prec_v$.
   (b) If $m = \langle\langle act, v, oid\rangle, \cdot, \cdot\rangle$ then:
   - Let s be the last state in γ_v.
   - If there exists some state s' such that $\langle s, act, s'\rangle \in \delta_v$, i.e., if action act is enabled, then choose the unique s' such that this condition is satisfied and append $act.s'$ to γ_v.

   (c) Else if $m = \langle\langle \mathsf{reset}, v, oid\rangle, \cdot, \cdot\rangle$ then:
   - Let s be the last state in γ_v.
   - Choose the unique s' such that $\langle s, \mathsf{fail}, s'\rangle \in \delta_v$. (Recall that input actions are always enabled.)
   - Choose $s'' = v_0$.
   - Append $\mathsf{fail}.s'.\mathsf{reset}.s''$ to γ_v.

   (d) If m represents a virtual-geocast-rcv event, remove from S every message m' that proposes the identical virtual-geocast-rcv, as identified by the associated unique identifier.
4. If $S = \emptyset$ and at some point in execution α after the last event in S there does not exist any $i \in I$ such that $status_{v,i} = \mathsf{active}$, then add a fail_v to the end of γ_v.

the event preceding it in γ_v.

5.2.3 Constructing the Trace

Next, we construct a hybrid sequence β of externally-visible actions by the clients and virtual nodes, i.e., of externally-visible actions of S'. (Recall we are postponing consideration of the virtual RealWorld until Section 5.2.5.) We show that β is a trace of S', and use it to align the various component executions. As a result, β enables us to paste together the various component executions in a single execution of S'.

More formally, we construct a sequence β such that the following properties holds:

- For all $i \in I$, let $external_i$ be the externally visible actions of client i. For all $i \in I$, $\beta | external_i = trace(\gamma_i)$.
- For all $v \in O$, let $external_v$ be the externally visible actions of virtual node v. For all $v \in O$, $\beta | external_v = trace(\gamma_v)$.

As a result, the trace β is consistent with the individual component executions.

In order to construct this trace, we begin with a trace of the physical system, i.e., a trace of α. We exclude from this trace all the events that are not relevant to clients and virtual nodes, and then substitute some virtual node events for certain emulator events.

More specifically, let β be the trace of α containing all the externally visible client events, i.e., $external_i$ for all $i \in I$. Also, temporarily, include in β all the simulate-op events and process-reset events as well. (We will remove these events after the next step.) Formally, then, let:

$$\begin{aligned} E \;=\; & \{external_i : i \in I\} \\ & \cup\; \{\mathsf{simulate\text{-}op}(act, v, oid)_i : act \in actions[v], v \in O, oid \in U, i \in I\} \\ & \cup\; \{\mathsf{process\text{-}reset}(v, oid)_i : oid \in U, i \in I\} \end{aligned}$$

and let $\beta = \alpha | \langle E, \emptyset \rangle$.

It remains to add the virtual node events, and to remove the simulate-op and process-reset events. For each virtual node v, we add the externally visible events from γ_v to β as follows:

Let e be some externally-visible event from γ_v. If e is not a fail_v event or a reset_v, then by the construction of γ_v, for some $j \in I$, there is a $\mathsf{simulate\text{-}op}(e, v, oid)_j$ event in $IM(\alpha)_v$, and hence in α, that led to event e being added to γ_v. In this case, add event e to β immediately after—and with no time passage—as the first $\mathsf{simulate\text{-}op}(e, v, oid)$ event in β.

If e is a fail_v event followed by a reset_v event, then by the construction of γ_v, for some $j \in I$, there is a $\mathsf{process\text{-}reset}(v, oid)_j$ event in $IM(\alpha)_v$, and hence in α, that led to event e being added to γ_v. Let e' be the first $\mathsf{process\text{-}reset}(v, oid)_j$ event in β, and let e'' be the event immediately preceding e. Add a fail_v event immediately after event e'' with no time passage between event e'' and the fail event. Add a reset_v event to β immediately after event e' with no time passage between e' and the reset event.

If e is a reset_v event, notice by the construction of γ_v that it is immediately preceded by a fail_v event, and hence is covered by the previous case.

If e is a fail_v event that is not followed by a reset_v event, then add it to β after all the other events from γ_v at a point corresponding to the first point in α after the other events in γ_v at which there does not exist a $i \in I$ where $status_{i,v} = \mathsf{active}$. Since such a fail_v event is always the last event in γ_v, placing it at this point is clearly consistent with the other events in γ_v. Assign it the same time as its immediately preceding event.

Finally, remove all the simulate-op events and process-reset events from β. The resulting trace is clearly consistent with the client executions, as both β and the client executions are constructed directly from a restriction of α. It is also consistent with the virtual node executions since every external event in γ_v is added to β, and they are added in the same order and at the same time, i.e., the order of the simulate-op and process-reset events.

5.2.4 Pasting Executions

Next, we invoke a pasting lemma to combine the various component executions to generate an execution of S'. In Section 5.2.5, we construct an execution of U' and perform a second pasting step to produce an executino of $S' \times U'$, i.e., an execution of the Virtual Node Layer.

We begin by stating a basic lemma related to execution-pasting, which is an extension of Theorem 7.3 from [45, 46]:

Lemma 5.2.1. *Let A_1 and A_2 be compatible timed automata, and $A = A_1 \times A_2$. Let α_1 and α_2 be executions of A_1 and A_2, respectively.*

Let β be an $(E, \emptyset)$-sequence, where E is the set of external actions of A. Suppose that $\beta|(E_i, \emptyset) = trace(\alpha_i)$, $i \in \{1, 2\}$.

Then there exists an execution α of A such that $trace(\alpha) = \beta$, and $\alpha_i = \alpha|(A_i, X_i)$, $i \in \{1, 2\}$.

It follows that we can paste together a countable collection of components:

Corollary 5.2.2. *Let $A_1, A_2, \ldots, A_k$ be a finite collection of compatible timed automata, and let $A = A_1 \times A_2 \times \ldots A_k$. Let α_i be an execution of A_i.*

Let β be an $(E, \emptyset)$-sequence, where E is the set of external actions of A. Suppose that $\beta|(E_i, \emptyset) = trace(\alpha_i)$, $i \in \{1, 2, \ldots, k\}$.

Then there exists an execution α of A such that $trace(\alpha) = \beta$, and $\alpha_i = \alpha|(A_i, X_i)$, $i \in \{1, 2, \ldots, k\}$.

Using Corollary 5.2.2, we can paste together all the various components' executions that we have constructed in the preceding sections:

- γ_i, $i \in I$, the clients, and
- γ_v, $v \in O$, the virtual nodes.

Define $\gamma_{S'}$ to be the execution of S' resulting from pasting together these various automata. We now proceed in the following section to consider U'.

5.2.5 RealWorld / Virtual GeoCast Execution

In this section, we construct an execution of U', the RealWorld automaton (and the virtual GeoCast service) and again invoke the pasting lemma to construct an execution γ of $S' \times U'$. We construct the execution of U' as follows: first we notice that we can derive a trace $\beta_{U'}$ of U' from execution $\gamma_{S'}$, since every external action of U' is shared with S'. We show that this hybrid sequence $\beta_{U'}$ satisfies the integrity property, and that the geo-updates to the clients occur sufficiently frequently. By the definition of the virtual RealWorld automaton, there exists an execution of U' that is consistent with this sequence. We then invoke Lemma 5.2.1 to construct execution γ.

The key claim in this section is that the sequence $\beta_{U'}$ satisfies the integrity property, implying that it is a trace of the virtual RealWorld automaton. We show that for every $\mathsf{virtual\text{-}geocast\text{-}rcv}(m,d)_k$ event in $\beta_{U'}$ of U', there is a preceding virtual geocast, i.e., $\mathsf{virtual\text{-}geocast}(m,d)_\ell$. Notice that nodes k and ℓ can be either clients (identified by the set I) or virtual nodes (identified by the set O). The proof of this integrity property follows the same structure as Section 4.2.2, and the main claim, Invariant 5.2.3, is similar in structure to Invariant 4.2.1.

Defining the sequence. We begin by defining the sequence $\beta_{U'}$ of the RealWorld/-GeoCast automaton. Recall that we have already determined an execution β of execution $\gamma_{S'}$ in Section 5.2.3. We begin with the trace β and restrict to the set of externally-visible actions of the virtual RealWorld. Formally, let:

$$\begin{aligned} E_{\mathsf{geo}} \;=\; & \{\mathsf{virtual\text{-}geocast}(m,d)_k : m \text{ is a message}, d \in \mathbb{R}, k \in I \cup O\} \\ & \cup\ \{\mathsf{virtual\text{-}geocast\text{-}rcv}(m,d)_k : m \text{ is a message}, d \in \mathbb{R}, k \in I \cup O\} \\ & \cup\ \{\mathsf{fail}_v : v \in O\} \\ & \cup\ \{\mathsf{fail}_i : i \in I\} \\ & \cup\ \{\mathsf{reset}_v : v \in O\} \\ & \cup\ \{\mathsf{geo\text{-}update}(t,\ell)_i : t \in \mathbb{R}, \ell \in L, i \in I\} \end{aligned}$$

and let $\beta_{U'} = \beta | \langle E_{\mathsf{geo}}, \emptyset \rangle$. Our goal is to show that β' satisfies the integrity property, and hence that there exists an execution of U' with trace $\beta_{U'}$.

Defining partial executions. We now define partial executions that reflect the operations known to a given replica. For every prefix α' of α, for every $i \in I$, define $\overline{\gamma(\alpha', v, i)}$ as follows. We first construct an asynchronous execution, as in Figure 5-8, and then assign a time to each event. Define $\overline{\gamma(\alpha', v, i)}$ as is described in Figure 5-9. First, choose message $\overline{m}$ to be the largest message received by i via the fpcast_v service in α' that is not in $\ell state(\alpha').pending\text{-}ops_{v,i}$. The construction proceeds exactly as in Figure 5-8, with the exception that we only consider messages that precede $\overline{m}$. That is, we iterate over the messages in $IM(\alpha)_v$ that precede $\overline{m}$, considering the messages that represent either an action for v or a reset request. We apply these actions/resets in the specified total-order, omitting those that are not enabled, and inserting fails

Figure 5-9: Defining Partial Execution $\overline{\gamma(\alpha', v, i)}$

Choose $\overline{m}$ to be the largest message received by i via the fpcast_v service in α' that is not in $\ell state(\alpha').pending\text{-}ops_{v,i}$.

1. Initially, set $\overline{\gamma(\alpha', v, i)} = s_0$, the initial state of virtual node v.
2. Define the set S as follows:

$$S = \{\langle P, cnt, k\rangle : \; P \in IM(\alpha)_v, \; \langle P, cnt, k\rangle \text{ is the message associated with } P, \langle P, cnt, k\rangle \leq \overline{m}\} .$$

3. Repeat while $S \neq \emptyset$:
   (a) Let m be the minimum message in S, according to the total order $\prec_v$.
   (b) If $m = \langle\langle act, v, oid\rangle, \cdot, \cdot\rangle$ then:
   - Let s be the last state in $\overline{\gamma(\alpha', v, i)}$.
   - If there exists some state s' such that $\langle s, act, s'\rangle \in \delta_v$, i.e., if action act is enabled, then choose the unique s' such that this condition is satisfied and append $act.s'$ to $\overline{\gamma(\alpha', v, i)}$.

   (c) Else if $m = \langle\langle \mathsf{reset}, v, oid\rangle, \cdot, \cdot\rangle$ then:
   - Let s be the last state in $\overline{\gamma(\alpha', v, i)}$.
   - Choose the unique s' such that $\langle s, \mathsf{fail}, s'\rangle \in \delta_v$. (Recall that input actions are always enabled.)
   - Choose $s'' = v_0$.
   - Append $\mathsf{fail}.s'.\mathsf{reset}.s''$ to $\overline{\gamma(\alpha', v, i)}$.

   (d) If m represents a $\mathsf{virtual\text{-}geocast\text{-}rcv}$ event, remove from S every message m' that proposes the identical $\mathsf{virtual\text{-}geocast\text{-}rcv}$, as identified by the associated unique identifier.

before recoveries.

Notice that the sequence constructed by Figure 5-9 is necessarily a prefix of the sequence constructed by Figure 5-8: both constructions follow an identical process, but the former considers only messages that precede some message $\overline{m}$ (including $\overline{m}$ itself) in the ordering inducted by the fpcast service.

We therefore assign a time to each event in a manner that matches γ_v: if some event e occurs in $\overline{(\gamma(\alpha', v, i))}$, then it also occurs in γ_v; assign event e the same time in the former as it has in the latter. Notice, then, the following observation:

Observation 1. *For every $\alpha' < \alpha$, for every $i \in I$: $\overline{\gamma(\alpha', v, i)}$ is a prefix of γ_v.*

This follows since the partial execution $\overline{\gamma(\alpha', v, i)}$ is defined by set S which is a prefix of the (ordered) set of events $IM(\alpha)_v$.

Notice that every reset event in $\overline{\gamma(\alpha', v, i)}$, for any α' and i, is associated with some $\langle \mathsf{reset}, v, oid \rangle$ message received in α'. We refer to oid as the **reset identifier** for such a reset event.

The main invariant. We now proceed to show that for every $i \in I$, the execution $\overline{\gamma(\alpha', v, i)}$ reflects exactly the state of replica i at the end of α'. In particular, we show that for every active replica i, the replica state $val_{v,i}$ is equal to the last state in $\overline{\gamma(\alpha', v, i)}$ at the end of every prefix α'.

Invariant 5.2.3. *Let execution α' be a finite prefix of execution α, and mobile node $i \in I$. If $\ell state(\alpha').status_{v,i} = \mathsf{active}$, then $\ell state(\alpha').val_{v,i} = \ell state(\overline{\gamma(\alpha', v, i)})$.*

Proof. We show this by induction on the length of α'. Throughout the induction, we also maintain two further invariants. The first, Invariant 5.2.4, show that an active replica is aware of the entire history since the most recent reset event. Notice that for each fpcast-rcv(m) event, message $m = \langle P, cnt, k \rangle$, i.e., consists of some P, which we refer to as the payload, along with two further parameters that ensure each message is unique.

Invariant 5.2.4. *For all $j \in I$, if the following conditions hold for some prefix α' of α:*

- *$\ell state(\alpha').status_{v,j} = \mathsf{active}$,*
- *message m is the largest message such that a fpcast-rcv$(m)_{v,j}$ occurs in α',*
- *message m' is the largest message $\leq m$ such that $m' = \langle \langle \mathsf{reset}, v, \cdot \rangle, \cdot, \cdot \rangle$, or $m' = \perp$ if there is no such message;*

then for every message $m'' : m' < m'' \leq m$ that is received in α via a fpcast-rcv$_{v,j'}$ for $j' \in I$ where $m'' = \langle P, \cdot, \cdot \rangle$, the payload P is in $\ell state(\alpha').pending\text{-}ops_{v,j} \cup \ell state(\alpha').completed\text{-}ops_{v,j}$.

The second, Invariant 5.2.5, shows that the the *reset-id* of a replica correctly tracks the most recent reset identifier. This is used to ensure that a replica only processes operations that have occurred since the last reset. Together with Invariant 5.2.4, this ensures that each operation is only executed once: operations prior to the most recent reset are ignored, and operations since the most recent reset are available in the history of *completed-ops*.

Invariant 5.2.5. *For all $j \in I$, if for some prefix α' of α:*

- $\ell state(\alpha').status_{v,j} = \mathsf{active}$,
- *e is the last* reset *event in α', and*
- *oid is the reset identifier of e;*

then $\ell state(\alpha').reset\text{-}id_{v,j} = oid$. If no such event e exists in execution α', then:

$$\ell state(\alpha').reset\text{-}id_{v,j} = \langle i_0, -1 \rangle \ .$$

For the base case, consider the initial state of the system, before any events occur in α. Let γ_{post} be this empty prefix of α. If i is not in the focal point region, then $status_{v,i} = \mathsf{idle}$, and the claim is trivial. If i is in the focal point, then the state of $val_{v,i}$ is v_0, the initial state of virtual node v. Since i has received no messages, $\overline{\gamma(\gamma_{post}, v, i)}$ is also the empty execution, satisfying the main invariant. Invariant 5.2.4, for the base case, also follows immediately from the fact that there are no messages received in the empty execution. Similarly, for Invariant 5.2.5, γ_{post} has no reset events, and every node has $reset\text{-}id_{v,j} = \langle i_0, -1 \rangle$ in its initial state, implying the desired condition.

There are two types of inductive steps to consider for executions of timed automata: discrete events and trajectories. Notice that for trajectories, the invariants are trivially maintained since all the state elements mentioned in the invariants are discrete and unchanged under time passage. Consider, then, a discrete event.

Let s and s' be the states before and after the new event, respectively. Let γ_{pre} be the previous prefix of α ending in state s, that is, $s = \ell state(\gamma_{pre})$. Let γ_{post} be the new prefix of α after the new event, that is, $s' = \ell state(\gamma_{post})$.

We know, inductively, that for any finite prefix α' of γ_{pre}, for any node j, if $\ell state(\alpha').status_j = \mathsf{active}$, then $\ell state(\alpha').val_j = \overline{\gamma(\alpha', v, j))}$, and also that Invariant 5.2.4 holds. We need to show that $\ell state(\gamma_{post}).val_{v,i} = \overline{\gamma(\gamma_{post}, v, i)}$, and that Invariant 5.2.4 continues to hold. We now consider the different events that can occur which modify the state relevant to the three invariants:

- $\mathsf{fpcast\text{-}rcv}(\langle\langle act, vnid, oid\rangle, \cdot, \cdot\rangle)_{v,i}$: (Figure 5-5, lines 22–28.)
  This event adds $\langle \mathsf{act}, vnid, oid \rangle$ to $pending\text{-}ops_v$, if the status of i is active and if the operation is not already in $pending\text{-}ops_v$ or $completed\text{-}ops_v$.

  Recall that the execution $\overline{\gamma(\gamma_{post}, v, i)}$ includes only messages that have been simulated in γ_{post} and precede the largest message received by i not in the set of pending operations, i.e., $pending\text{-}ops_{v,i}$. There are three cases to consider:

– The event $\langle act, vnid, oid\rangle$ is already in *pending-ops*$_{v,i}$. In this case, the state of *pending-ops*$_{v,i}$ is not modified. The message $m = \langle\langle act, vnid, oid\rangle, \cdot, \cdot\rangle$ becomes the largest message received by i (in the total order); however, since m is already in *pending-ops*$_{v,i}$, the message $\overline{m}$ that is used to define $\overline{\gamma(\gamma_{pre}, v, i)}$ remains unchanged.

  As a result, the execution $\overline{\gamma(\gamma_{post}, v, i)}$ is equal to $\overline{\gamma(\gamma_{pre}, v, i)}$. By induction, we already know that $\overline{\gamma(\gamma_{pre}, v, i)} = \ell state(\gamma_{pre}).val_{v,i}$. Since $val_{v,i}$ remains unchanged, the claim follows.

– The event $\langle act, vnid, oid\rangle$ is not in *pending-ops*$_{v,i}$ or *completed-ops*$_{v,i}$. In this case, the event $\langle act, vnid, oid\rangle$ is added to *pending-ops*$_{v,i}$. As in the previous case, the received message becomes the largest message received by i, but does not result in any changes to $\overline{\gamma(\gamma_{pre}, v, i)}$ since the message is in *pending-ops*$_{v,i}$.

– The event $\langle act, vnid, oid\rangle$ is in *completed-ops*$_{v,i}$. In this case, the largest message received by i, as determined by the total ordering, remains unchanged: some message $\langle\langle act, vnid, oid\rangle, \cdot, \cdot\rangle$ (from a different node) has already been received earlier in the ordering in order to be resident in *completed-ops*$_{v,i}$.

We next proceed to argue that Invariant 5.2.4 is maintained. If the message $\langle\langle act, vnid, oid\rangle, \cdot, \cdot\rangle$ is not the largest message received by i, then the situation is unchanged and Invariant 5.2.4 continues to hold by the inductive hypothesis.

Consider the case where message $\langle\langle act, vnid, oid\rangle, \cdot, \cdot\rangle$ is the largest message received by i from the fpcast_v service in γ_{post}. Let m' be the largest message received by i in γ_{pre} from the fpcast_v service. We show that every message $m'' \;:\; m' < m'' < m$ has its payload either in *pending-ops* or *completed-ops*. (Notice that this proves a property somewhat stronger than the necessary invariant, as it does not exclude messages prior to a reset event that may occur between m'' and m.)

Assume for the sake of contradiction that such a message m'' exists whose payload is not in either set. We argue that this implies that node i has exited the focal point region at some point after receiving message m' and prior to receiving message m: since every message received from the fpcast_v service is added to either *pending-ops* or *completed-ops*, and messages are removed only when i exits the focal point region, there are two possible cases:

– Node i exits the focal point region at some point between receiving m and m', and resets *pending-ops* during the geo-update, removing the payload of message m'' from *pending-ops*.

– Node i does not receive m''. In this case, we conclude by the *consistent delivery* property of the fpcast_v service that i exits the focal point region at some point between receiving m and m'.

In either case, i exits the focal point between receiving message m and m', setting $status_{i,v}$ to idle. Moreover, it is easy to see that i does not reset its *status* to active prior to receiving message m: we know that message m' is the most recent message received by i prior to m (since it is the largest message in γ_{pre}), and hence i did not receive a join-ack between messages m' and m. This contradicts our assumption that $status_{i,v} =$ active.

Finally, consider Invariant 5.2.5: notice that this event has no effect either on the reset events in execution $\gamma(\gamma_{pre}, v, i)$, nor does it modify $reset\text{-}id_{i,v}$; therefore the invariant is maintained.

- simulate-op$(act, v, r\text{-}id, oid)_{v,i}$: (Figure 5-6, lines 76–87.)
  First, when this event occurs, $status_{v,i} =$ active, since that is a precondition to this action (Figure 5-6, line 78). The event removes $\langle act, v, oid\rangle$ from the set $pending\text{-}ops_{v,i}$, and thus the message m which originally caused this operation to be added to $pending\text{-}ops_{v,i}$ is now included in the construction of execution $\overline{\gamma(\gamma_{post}, v, i)}$; in particular, since m no longer refers to the largest message received by i and (no longer) in *pending-ops*, we can conclude that *act* is the last event in $\overline{\gamma(\gamma_{post}, v, i)}$. By construction we can conclude that $\overline{\gamma(\gamma_{post}, v, i)}$ includes every operation specified by every message preceding m that is received in α.

  There are two subcases to consider.

  - Message m contains a payload $\langle act, v, oid\rangle$ included in another message already received and processed. That is, there exists a message delivered earlier in the ordering with the same action, virtual node, and operation identifier. In this case, $\overline{\gamma(\gamma_{post}, v, i)} = \overline{\gamma(\gamma_{pre}, v, i)}$, as the construction includes each payload only once. It remains to show that $val_{v,i}$ is also unmodified as a result of the simulate-op event. If $r\text{-}id \neq reset\text{-}id_{i,v}$, then $val_{v,i}$ is unmodified as desired (Figure 5-6, line 82).

    Assume therefore that $r\text{-}id = reset\text{-}id_{i,v}$. By Invariant 5.2.5, we conclude that either there are no reset events in γ_{pre}, or $r\text{-}id$ is the reset identifier of the most recent reset event in γ_{pre}. Let $m' = \langle\langle \text{reset}, v, r\text{-}id\rangle, \cdot, \cdot\rangle$ be the message in $IM(\alpha')$ associated with that reset event. Notice that message m could not have been received and processed prior to message m' as in that case i would have a different *reset-id* and hence would reject message m. Thus we are interested in the case where i has received some message m'' with the same operation identifier as m such that in the total ordering $m' < m'' < m$.

    By induction, we know from Invariant 5.2.4 that every message $m'' : m' < m'' \leq m$ that is received in α is in $pending\text{-}ops_{v,i} \cup completed\text{-}ops_{v,i}$. Thus, the payload of the earlier message m'' identifying the same operation as m is also in $pending\text{-}ops_{v,i} \cup completed\text{-}ops_{v,i}$. We also know that it is not in $pending\text{-}ops_{v,i}$ since message m is the smallest—and hence oldest—message in *pending-ops* (as it is a queue); from this we conclude that the earlier message is in $completed\text{-}ops_{v,i}$. In this case the simulated-op event

transition does not modify $val_{v,i}$ due to Figure 5-6, line 81, and the main invariant is maintained.

– Message m refers to a new operation, in which case

$$\overline{\gamma(\gamma_{post}, v, i)} = \delta(act, \ell state(\overline{\gamma(\gamma_{pre}, v, i)})) \ .$$

By induction we know that $val_{v,i} = \ell state(\overline{\gamma(\gamma_{pre}, v, i)})$ in state s. After the simulate-op transition, $val_{v,i}$ is set to $\delta(act, val_{v,i})$. This maintains the desired invariant.

Since this event neither results in a new message being received from the fpcast_v service, nor removes any messages from *pending-ops* $\cup$ *completed-ops* (only moving a message from one to the other), Invariant 4.2.2 is also maintained.

Finally, consider Invariant 5.2.5: notice that this event has no effect either on the reset events in execution $\gamma(\gamma_{pre}, v, i)$, nor does it modify $reset\text{-}id_{i,v}$; therefore the invariant is maintained.

- fpcast-rcv$(\langle\langle$join-ack$, jid, v,$ *c-ops*$\rangle, \cdot, \cdot\rangle)_{v,i}$: (Figure 5-5, lines 9–20)
In this event, node i sets $status_{v,i}$ to active (Figure 5-5, line 14), if the message is a response to an outstanding join-req previously sent by i. (If this is not the case, then this action causes no change to $status_{v,i}$, $pending\text{-}ops_{v,i}$, $reset\text{-}id_{v,i}$, or $val_{v,i}$, and adds the message to $completed\text{-}ps_{v,i}$; thus the invariants are trivially maintained.)

The fpcast_v service guarantees that if a message is received, an earlier fpcast occurred at some node j that sent the message (that is, it guarantees message integrity). In particular, some node j previously performed a

$$\mathsf{fpcast}(\langle \text{join-ack}, jid, v, val, r\text{-}id, c\text{-}ops\rangle)_j \ .$$

The only action that causes a join-ack to be sent is a prior join request being received . Therefore, node j previously performed an fpcast-rcv$(\langle$join-req$, jid, v\rangle)_{v,i}$. Let γ'' be the prefix of γ_{pre} ending with the fpcast-rcv$(\langle$join-req$, jid, v\rangle)_j$ event.

Consider the state of node j at the end of γ''. First, the status, $status_{v,j}$, must be active; otherwise node j would not send a response to the join request (Figure 5-6, line 91). Inductively, then, we know that $\ell state(\gamma'').val_{v,j}$ is equal to $\ell state(\overline{\gamma(\gamma'', v, j)})$. Thus, we can conclude that val, the value sent from j to i, is also equal to $\ell state(\overline{\gamma(\gamma'', v, j)})$. By Invariant 5.2.5 we can conclude that $\ell state(\gamma'').reset\text{-}id_{v,j}$ identifies the most recent reset in $\overline{\gamma(\gamma'', v, j)}$.

Since $val_{v,i}$ is set to val when the join-ack message is received, i.e., during the transition in question, it remains only to show that:

$$\overline{\gamma(\gamma_{post}, v, i)} = \overline{\gamma(\gamma'', v, j)} \ , \tag{5.1}$$

and we can conclude that $val_{v,i} = \ell state(\overline{\gamma(\gamma_{post}, v, i)})$, as desired. From this we

will also conclude that the reset identifier *r-id* included in the join-ack message is the most recent reset identifier for $\overline{\gamma(\gamma_{post}, v, i)}$, thus maintaining Invariant 5.2.5.

Let m be the largest message that is received by i via the fpcast_v service in γ_{post} whose payload is not in $s'.\textit{pending-ops}_{v,i}$, that is, m is the message used in the construction of $\overline{\gamma(\gamma_{post}, v, i)}$. We claim that m must correspond to node i's join request, which implies Equation 5.1, as it is easy to see that i's join request is the message used in the construction of $\overline{\gamma(\gamma'', v, j)}$: i's join request is the last message received by j and removed from $\textit{pending-ops}_{v,j}$ prior to sending the join-ack message.

First, we know that i has already received its own join request in γ_{post}: i sets its status to active only if it has previously received its own join request and set its status to listening. Moreover, i does not add its own join request to *pending-msgs* and hence we can conclude that m is no smaller that i's join request.

Second, notice that when i receives its own join request, it sets its status to listening and when i's status is listening, it adds every message it receives (after its own join request) to $\textit{pending-ops}_{v,i}$ (Figure 5-5, line 5). No message is removed from the set $\textit{pending-ops}_{v,i}$ because $\textit{status}_{v,i}$ is not yet active (Figure 5-6, line 78). Therefore m cannot be equal to any message received by i after i's own join request.

We conclude, then, that m is exactly i's join request. Recall that the execution $\overline{\gamma(\gamma_{post}, v, i)}$ is defined as including all the simulated messages prior to m, that is, prior to i's join request.

Notice that i's join request is exactly the last message processed by j in γ'' before sending a response to i. In particular, then, $\overline{\gamma(\gamma'', v, j)}$ is the execution including every operation in $IM(\gamma'')$ prior to i's join request. Therefore, Equation 5.1 holds, and the invariants holds in state s'.

We next argue that Invariant 4.2.2 is also maintained. Notice that when node j sends the join-ack message, it also includes a copy of its set $\textit{completed-ops}_{v,j}$, and that at this point in the execution, Invariant 4.2.2 holds with respect to j; this means that the payload of every message preceding node i's join request and after the most recent reset is in $\textit{pending-ops}_{v,j} \cup \textit{completed-ops}_{v,j}$. Since $\textit{pending-ops}_{v,j}$ is a queue, and since when responding to a join request, all payloads from preceding messages have been removed from $\textit{pending-ops}_{v,j}$, we can conclude that the payload of every message preceding the join request is in $\textit{completed-ops}_{v,j}$. When node i receives the join-ack, it adds all the message payloads in the set *c-ops* to its own set $\textit{completed-ops}_{v,i}$. It remains only to show that every message payload sent after i's join request is in $\textit{pending-ops}_{v,i}$. This follows immediately from the fact that i has remained in the focal point region, and the *consistent delivery* property of the fpcast_v service.

- process-join-req(jid, $vnid_i$: (Figure 5-6, lines 89–96)
  In this case, a join-req message is removed from $\textit{pending-ops}_{v,i}$, which potentially

indicates that $\overline{\gamma(\gamma_{post})} \neq \overline{\gamma(\gamma_{pre})}$.

In fact, the virtual execution prefix is unmodified. Let m be the join-req message $\langle \text{join-req}, jid, vnid \rangle$ that precipitated this event. Let $\overline{m}$ be the largest message received by i via the fpcast service in γ_{pre} that is not in $pending\text{-}ops_{v,i}$ at the end of γ_{pre}, i.e., the message that is used to define $\overline{\gamma(\gamma_{pre})}$. We argue that there are no intervening messages in $IM(\alpha)$ between $\overline{m}$ and the join-req message m.

Assume for the sake of contradiction that such an intervening message $m' \in IM(\alpha)$ exists where $\overline{m} < m' < m$. Choose m' to be the largest such message. By the *consistent delivery* property of the fpcast, either i receives message m', or i leaves the focal point point between receiving $\overline{m}$ and m. In the latter case, however, i sets $status_{v,i}$ to *idle*; by assumption (i.e., the choice of $\overline{m}$) there is no intervening message in γ_{pre} that causes i to reset its status to active, thus contradicting our assumption that i is active. We thus conclude that i receives message m'. By the choice of $\overline{m}$ this implies that m' is in $pending\text{-}ops_{v,i}$, which contradicts the fact that messages are processed in order and hence m' would have been processed prior to m.

Finally, we notice that message m is not in $IM(\alpha)$, as it is a join request, and there are no intervening messages between *overlinem* and m that are in $IM(\alpha)$, and hence the virtual execution prefix is unmodified.

- process-join-ack$(v)_i$: (Figure 5-6, lines 98–103)
  In this case, a join-ack message is removed from $\text{pending-ops}_{v,i}$. The argument is identical to the previous case.

- process-reset$(v, oid)_i$: (Figure 5-6, lines 114–124)
  In this case, the virtual node is reset. In the case where the replica is not in the focal point, the invariants are trivially maintained since the status of i is idle. We thus assume that i is in the focal point and hence executes the reset code.

  Let m be the $\langle \text{reset}, v, oid \rangle$ message that leads to this event. Since messages are processed in order, it is clear that m is the largest message received by i that is not in $pending\text{-}ops_{v,i}$. Thus the last event in $\overline{\gamma(\gamma_{post}, v, i)}$ is a reset event, and hence the final state is v_0, the initial state. Similarly, after the process-reset event, $val_{v,i} = v_0$, and thus the main invariant is maintained.

  Similarly, Invariant 5.2.4 is maintained since every message received after m remains in $pending\text{-}ops_{v,i}$. (The consistent delivery property ensures that there are no gaps in the sequence of received messages.) Messages prior to m are no longer relevant since m is a reset event.

  Finally, Invariant 5.2.5 is maintained since this event updates $reset\text{-}id_{v,i}$ to oid, which is the reset identifier for this reset event.

The rest of the cases are straightforward, having no effect on the status of i, the state of the replica, or the *reset-id*; moreover, every other fpcast message is added to *completed-ops* or *pending-ops*, and no elements are removed (except when i exits the focal point region). □

Relating states to executions. Our goal now is to show that the execution γ_v constructed for virtual node v is consistent with the simulation performed by each replica, i.e., by each VNE-Server automaton. We have already shown that the state of each replica is consistent with the partial executions; we now show that this is consistent with γ_v as well. This follows primarily from the fact that each partial execution is a prefix of γ_v.

In more detail, a replica i simulates an event act in the virtual node execution by performing a $\mathsf{simulate\text{-}op}(act, v, oid)_{v,i}$ step. Thus, we need to show that each such $\mathsf{simulate\text{-}op}$ event in α reflects an event in the execution γ_v. We therefore prove the following lemma:

Lemma 5.2.6. *Let α' be a finite prefix of execution α such that*

$$\alpha'' = \alpha'.\mathsf{simulate\text{-}op}(act, v, oid)_{v,i}$$

is also a prefix of α. Then there exists a prefix γ' of γ_v such that the following hold:

- pre-state: $\ell state(\alpha').val_{v,i} = \ell state(\gamma')$,
- transition: *Let $s'' = \delta(\ell state(gamma'), act)$. Execution $\gamma'.act.s''$ is also a prefix of γ_v.*
- post-state: $s'' = \ell state(\alpha'').val_{v,i}$.

Proof. This lemma follows immediately from Invariant 5.2.3. Choose γ' to be $\overline{\gamma(\alpha', v, i)}$. Notice that γ' is a prefix of γ_v. The *pre-state* claim exactly restates Invariant 5.2.3. The *transition* claim follows from the fact that $\overline{\gamma(\alpha', v, i)}$ is defined in terms of a message $\overline{m}$ that is the largest message received by i in α' whose payload is not in $pending\text{-}ops_{v,i}$; the message $\langle act, vn, oid\rangle$ posited by the precondition of $\mathsf{simulate\text{-}op}$ must in fact be the message immediately following $\overline{m}$ in the total order, and hence act is the next event in the virtual node execution. Finally, the post-state claim again restates Invariant 5.2.3, with respect to α''. □

Proving Integrity. Since the trace $\beta_{U'}$ of the virtual RealWorld automaton is derived from the executions $\gamma_{S'}$, which is itself derived from pasting client executions γ_i, $i \in I$, and virtual node executions γ_v, $v \in O$, we can now conclude that the virtual RealWorld automaton guarantees integrity:

Lemma 5.2.7. *For every $k \in I \cup O$, if a $\mathsf{virtual\text{-}geocast\text{-}rcv}(m, d)_k$ event occurs in $\beta_{U'}$, then there exists some $\ell \in I \cup O$ such that a $\mathsf{virtual\text{-}geocast}(m, d)_\ell$ precedes it in $\beta_{U'}$.*

Proof. There are two main cases, depending on whether k is a client or a virtual node:

- k is a client: In this case, since there is a $\mathsf{virtual\text{-}geocast\text{-}rcv}(m, d)_k$ in $\beta_{U'}$, there must also be a $\mathsf{virtual\text{-}geocast\text{-}rcv}(m, d)_k$ in γ_k, and hence also in α. The VNE-Client automaton at k performs this action only if it has received a $\mathsf{geocast\text{-}rcv}(m, d)_k$ from the underlying GeoCast service. We can conclude by

the integrity of the underlying GeoCast service that there is some $i \in I$ that previously geocast message m in α. There are two possibilities: either the geocast came from another client, i.e., a VNE-Client automaton fom some $\ell \in I$, or from a VNE-Server automaton for some virtual node $v \in O$ and some replica $\ell \in I$.

In the former case, the desired conclusion follows immediately: since some VNE-Client at mobile node $\ell \in I$ performs a $\mathsf{geocast}(m,d)_\ell$, we can conclude that client c_i previously performed a $\mathsf{virtual\text{-}geocast}(m,d)_\ell$ in α. By the construction of γ_ℓ, this virtual GeoCast by client ℓ occurs also in γ_ℓ, and hence also in $\gamma_{S'}$, and hence also in $\beta_{U'}$.

In the latter case, we know that some VNE-Server GeoCast message m. This occurs only when the VNE-Server adds the message to the $\mathsf{geocast\text{-}queue}_{v,i}$ during a simulate-op event, from which we conclude by Lemma 5.2.6 that a $\mathsf{geocast}(m,d)_v$ event occurs in $\overline{\gamma(\alpha', v, i)}$, for appropriate choice of α', and hence in γ_v, and hence in $\gamma_{S'}$, and hence also in $\beta_{U'}$. By the construction of the trace $\beta_{U'}$, we can conclude that the $\mathsf{geocast}(m,d)_v$ in $\gamma_{S'}$ precedes the time at which the message is received.

- k is a virtual node: This case is essentially identical to the previous, with one additional step.

  Since there is a $\mathsf{virtual\text{-}geocast\text{-}rcv}(m,d)_k$ in $\beta_{U'}$ in $\beta_{U'}$, there must also be a $\mathsf{simulate\text{-}op}_k$ in α that led to this event, and hence a $\mathsf{geocast\text{-}rcv}(m,d)_\ell$ at some VNE-Server for some mobile node $\ell \in I$ which proposed the virtual-geocast-rcv. From this point, the argument proceeds as in the previous case. We can conclude by the integrity of the underlying GeoCast service that there is some $i \in I$ that previously geocast message m in α. There are two possibilities: either the geocast came from another client, i.e., a VNE-Client automaton fom some $\ell \in I$, or from a VNE-Server automaton for some virtual node $v \in O$ and some replica $\ell \in I$.

  In the former case, the desired conclusion follows immediately: since some VNE-Client at mobile node $\ell' \in I$ performs a $\mathsf{geocast}(m,d)_{\ell'}$, we can conclude that client c_i previously performed a $\mathsf{virtual\text{-}geocast}(m,d)_{\ell'}$ in α. By the construction of $\gamma_{\ell'}$, this virtual GeoCast by client ℓ' occurs also in $\gamma_{\ell'}$, and hence also in $\gamma_{S'}$, and hence also in $\beta_{U'}$.

  In the latter case, we know that some VNE-Server GeoCast message m. This occurs only when the VNE-Server adds the message to the $\mathsf{geocast\text{-}queue}_{v,i}$ during a simulate-op event, from which we conclude by Lemma 5.2.6 that a $\mathsf{geocast}(m,d)_v$ event occurs in $\overline{\gamma(\alpha', v, i)}$, for appropriate choice of α', and hence in γ_v, and hence in $\gamma_{S'}$, and hence also in $\beta_{U'}$. By the construction of the trace $\beta_{U'}$, we can conclude that the $\mathsf{geocast}(m,d)_v$ in $\gamma_{S'}$ precedes the time at which the message is received.

□

Recall that the virtual RealWorld automaton produces every integrity-producing trace that includes sufficiently frequent geo-updates. Since the trace $\beta_{U'}$ guarantees integrity, and since $\beta_{U'}$ contains sufficiently frequent geo-updates, we conclude that there exists an execution $\gamma_{U'}$ of the virtual RealWorld automaton such that $trace(\gamma_{U'}) = \beta_{U'}$.

Moreover, recall that $\beta_{U'} = \beta|\langle E_{\mathsf{geo}}, \emptyset\rangle$. Thus we conclude also that $trace(\gamma_{U'}) = \beta|\langle E_{\mathsf{geo}}, \emptyset\rangle$.

Finally, we invoke Lemma 5.2.1 a second time to paste executions $\gamma_{S'}$ and $\gamma_{U'}$ and conclude:

Lemma 5.2.8. *There exists an execution γ of $S' \times U'$ such that:*

$$trace(hide_{H'}(\gamma)) = trace(hide_H(\alpha)) \ .$$

Proof. Let A_1 and A_2 be compatible timed automata, and $A = A_1 \times A_2$. Let α_1 and α_2 be executions of A_1 and A_2, respectively.

Let β be an $(E, \emptyset)$-sequence, where E is the set of external actions of A. Suppose that $\beta|(E_i, \emptyset) = trace(\alpha_i)$, $i \in \{1, 2\}$.

Then there exists an execution α of A such that $trace(\alpha) = \beta$, and $\alpha_i = \alpha|(A_i, X_i)$, $i \in \{1, 2\}$.

We have already constructed executions $\gamma_{S'}$ of S' and $\gamma_{U'}$ of U'. We have also already constructed β such that $\beta = trace(\gamma_{S'})$ and $\beta|\langle E_{\mathsf{geo}}, \emptyset\rangle = trace(\gamma_{U'})$. Thus we conclude by Lemma 5.2.1, there exists an execution γ of $S' \times U'$ such that $trace(\gamma) = \beta$.

Moreover, we can conclude that if $actions_{S'}$ are the set of actions in S' and $X_{S'}$ are the set of variables in S', then $\gamma|\langle actions_{S'}, X_{S'}\rangle = \gamma_{S'}$.

Since the set of hidden actions H' includes all the non-client actions in S', we can conclude that:

$$trace(hide_{H'}(\gamma)) = trace(hide_{H'}(\gamma_{S'})) \ .$$

Since $\gamma_{S'}$ is itself constructed by pasting various γ_i, $i \in I$, executions (along with other component executions), and since each γ_i is defined as a restriction of α to the events of client i, we conclude that:

$$trace(hide_{H'}(\gamma_{S'})) = trace(hide_H(\alpha)) \ .$$

Together, these two equalities lead to the desired conclusion. □

From this, we conclude the main theorem of this section:

Theorem 5.2.9. *The Virtual Node Emulator guarantees a safe emulation of the Virtual Node Layer.*

5.3 Analysis: Liveness of the Emulation

In this section, we show that the emulation satisfies the liveness properties of the Virtual Node Layer. Specifically, this entails showing that the virtual RealWorld, in

particular the GeoCast subcomponent, guarantees reliable and timely message delivery. In order to guarantee liveness properties in the virtual infrastructure emulation, we need to make some further liveness assumptions for the underlying physical system. Specifically, we assume that:

- Any enabled action at a mobile node is executed immediately.
- For every fpcast event, the ϵ specified by the "Reliable Delivery" property is bounded by d_{fp}. This implies that every message broadcast using the fpcast service is delivered within time d_{fp}.
- For every geocast event, the ϵ specified by the "Reliable Delivery" property is bounded by ϵ_{geo}.
- Every message broadcast using the geocast service is delivered within time d_{geo}.

Also, recall from Section 3.2.3 that a focal point is said to be correct when some mobile node arrives at least time t_{join} prior to the previous mobile nodes leaving the focal point, and remains at least t_{join} afterwards. We assume that $t_{\mathsf{join}} = 4d_{\mathsf{fp}} + \epsilon_{\mathsf{geo}} + t_{\mathsf{upd}}$.

We also make one timing related restriction to the reset protocol. As presented in Figure 5-6, a mobile node could choose to reset a virtual node with very few restrictions. In order to show that the virtual node continues to operate throughout some period of time, we need to assume that the virtual node is not reset unnecessarily. That is, we assume that a node only initiates a reset by performing an initiate-reset if it has remained in *status* = *joining* for time $2t_{\mathsf{fp}}$ without receiving a join response. Otherwise, without this additional restriction, a node might initiate a reset non-deterministically, even when unnecessary.

The main lemma we prove in this section is that execution γ satisfies the reliable delivery property for the virtual GeoCast service. We also, at the same time, prove that each virtual geocast message is delivered within a bounded time.

Theorem 5.3.1. *There exists some ϵ_v such that if node $k \in I \cup O$ performs a* virtual-geocast$(m, d)_i$ *action at time t in γ, then there exists some time $t' > t + t_{upd}$ such that if $\ell \in I \cup O$ and satisfies the following:*

- *node ℓ is within distance R_{Vgeo} of location d at time t';*
- *node ℓ remains within distance R_{Vgeo} of location d until time $t' + \epsilon_v$; and*
- *node ℓ does not fail prior to time $t' + \epsilon_v$,*

then a virtual-geocast-rcv$(m, d)_\ell$ *event occurs at some point in the interval $[t', t' + \epsilon_v]$, delivering the message to node ℓ. Moreover, the message is delivered within time $d_{\mathsf{geo}} + \epsilon_v$.*

Proof. Notice that by assumption the ϵ associated with the underlying physical-layer GeoCast service is bounded by ϵ_{geo}. We choose $\epsilon_v = \epsilon_{\mathsf{geo}} + d_{litfp}$. There are two cases depending on whether k is a client or a virtual node. In each case we trace the virtual GeoCast from the sender to the receiver and show that it is received as required.

First, assume that node k is a client. In this case, by assumption, there is a virtual-geocast$(m, d)_k$ at time t, and hence we conclude that the VSE-Client for node k performs a geocast$(m, d)_k$ at time t. Let t' be the time specified by the reliable delivery property of the underlying GeoCast service. If ℓ is a client, then the claim follows immediately from the reliable delivery of the underlying GeoCast service.

Assume, then, that ℓ is a virtual node. By assumption, virtual node ℓ does not fail prior to some $t' + \epsilon_v$. From this we argue that there is some mobile node $i \in I$ that has $status_{v,i} =$ active throughout the interval $[t', t' + \epsilon_v]$ and does not fail prior to time $t' + \epsilon_v$.

This follows from the assumption that v is correct in this interval, implying that v is populated for at least $2t_{\mathsf{join}}$ prior to time t' through time $t' + \epsilon_v$.

We next argue that node j_0, as specified by the definition of a focal point being populated, as status active sufficiently prior to time t'. If any node has status active, then it is easy to see that the join protocol for j_0 completes successfully, if j_0 is not already active. Otherwise, consider the case where there are no nodes with status active for v at time $t' - 2t_{\mathsf{join}}$. Since v is populated from that point through time $t' + \epsilon_v$, we can conclude that some node resets v by performing an initiate-reset during that interval. In particular node j_0, as specified by the definition of a focal point being populated, becomes active within time $4t_{\mathsf{fp}}$: after waiting time 3_{fp} for its join protocol to complete, it either succeeds in resetting v in the next t_{fp} time, or it completes the join protocol. Thus we can assume without loss of generality that node j_0 is active at least time t_{join} prior to its departure, and at least time ϵ_{geo} prior to t'.

It is then easy to see (inductively) that each of the subsequent j_ℓ becomes active no later than time ϵ_{geo} prior to the previous $j_{\ell-1}$ departing, as the previous $j_{\ell-1}$ remains active and can respond to the join protocol.

Finally, since each j_ℓ remains at least t_{join} after the previous $j_{\ell-1}$ departs, there is some j_ℓ that is active and non-failed during the interval $[t', t' + \epsilon_v]$, as claimed. Fix i to be this mobile node.

We conclude that mobile node i receives the geocast of message m by time $t' + \epsilon_{\mathsf{geo}}$, and immediately performs a fpcast$_v$ of message m. This arrives within time d_{fp}, and immediately results in a simulate-op of that operation. By the construction of γ (and Lemma 5.2.6), we conclude that the virtual-geocast-rcv event occurs at exactly that instant, i.e., within time ϵ_v of t'. Moreover, we know that t' occurs at most time d_{geo} after time t, leading to the desired conclusion. This completes the case where k is a client.

Next, consider the case where k is a virtual node. We first identify the geocast(m, d) associated with this virtual-geocast, and the rest follows as in the previous case. By the construction of execution γ, we can identify a simulate-op event that is associated with the virtual-geocast$(m, d)_k$ of interest, and this event adds a message to a geocast-queue. Immediately thereafter, this message is geocast with the underlying physical geocast service. The remainder of this case is identical to the case where k is a client: if ℓ is a client, the result follows from the underlying geocast service guarantees; if ℓ is a virtual node, it follows from the fact that some mobile node replicating the virtual node receives the message and processes it. □

We conclude this chapter with one further claim regarding the conditional liveness of a virtual node:

Theorem 5.3.2. *If the precondition for some action for some virtual node v is enabled at some time t in γ, and if the virtual node is correct during the interval $[t, t + d_{\mathsf{fp}}$, then within time $t + t_{\mathsf{fp}}$ either the event in question occurs or the action is no longer enabled.*

Proof. This claim follows from the fact that if an event is enabled, than some replica performs a initiate-op event, which results immediately in an fpcast of a message suggesting that event. If the action is still enabled when the message arrives, and if the virtual node has not failed, then every replica that receives the message simulates that event. Thus the total latency is a single fpcast, i.e., t_{fp}. □

Chapter 6

The GeoQuorums Protocol
Atomic Storage in the Virtual Object Layer

In this chapter we present the GeoQuorums Operation Manager, an algorithm for implementing reliable, reconfigurable, atomic read/write storage in a mobile ad hoc network. That is, we implement a read/write shared memory that is reliable, in the sense of not failing under certain conditions, and reconfigurable, in the sense that it can be tuned for improved performance and availability.

The GeoQuorums Operation Manager presented in this chapter serves as an example of the *geoquorums approach*: the problem of implementing atomic memory is divided into two parts: (1) choosing a type of virtual infrastructure, and (2) developing a protocol targeted at that specific virtual infrastructure. In this case, the GeoQuorums Operation Manager is designed to execute in the Virtual Object Layer, as described in Chapter 3.

The goal of the GeoQuorums Operation Manager is to emulate a reliable, reconfigurable, atomic read/write object with in a mobile ad hoc network. Our implementation is described for a single read-write object; the composition of all the read/write objects results in a distributed shared read/write memory.

The GeoQuorums Operation Manager is designed for the Virtual Object Layer, which significantly simplifies the algorithm. There is no notion of mobility in the Virtual Object Layer, and as a result, the GeoQuorums Operation Manager avoids much of the complexity usually associated with an ad hoc mobile network. There is no need to handle nodes joining and leaving in any special way, as the only interprocess communication is through the virtual objects. If at most f of the underlying virtual objects fail, then the Operation Manager ensures that the resulting read/write memory is robust.

The GeoQuorums Operation Manager is a quorum-based algorithm. By replicating data at multiple virtual objects, and performing read and write operations on quorums of virtual objects, the GeoQuorums Operation Manager ensures that the data is maintained reliably and consistently.

The GeoQuorums Operation Manager relies on the variable type of the virtual objects, which we call the put/get variable type. These objects support specially defined operations, put, get, and two others, that allow clients to send information to

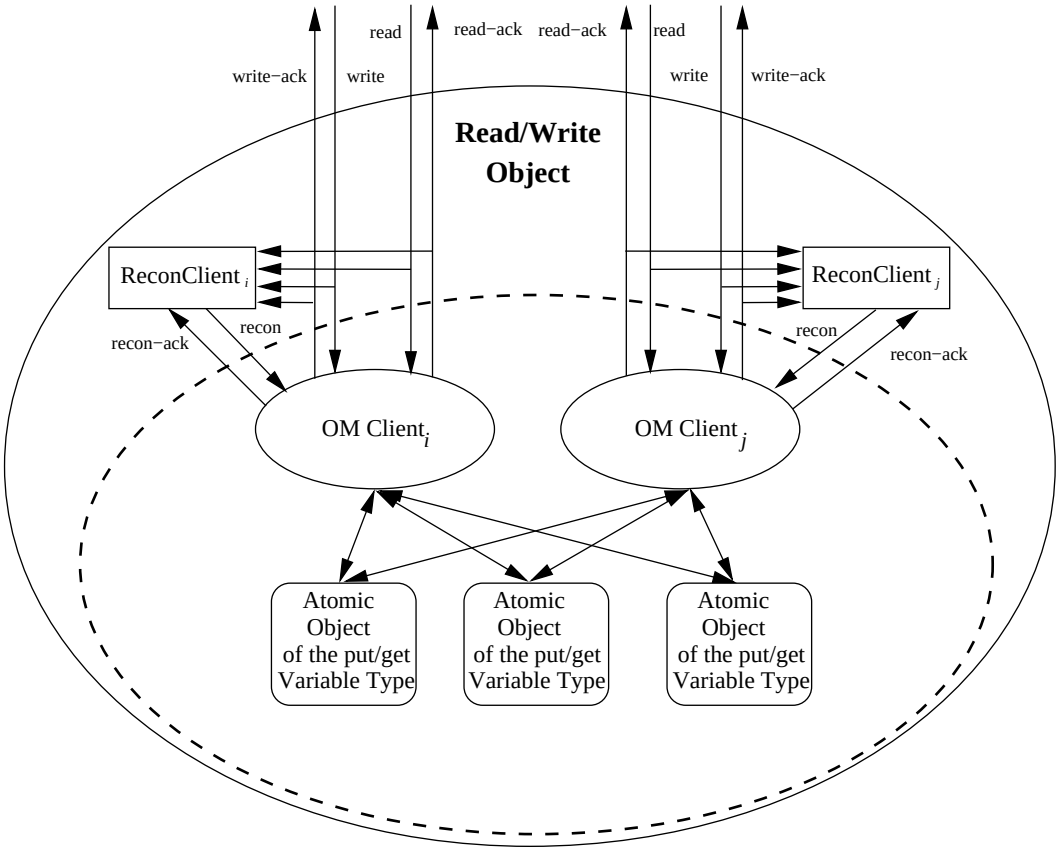

Figure 6-1: Implementing atomic read/write memory in the Virtual Object Layer. This figure depicts how the GeoQuorums Operation Manager is implemented in the Virtual Object Layer, as depicted in Figure 3-1. The dotted oval depicts the boundary between the Virtual Object Layer and the external clients that submit read and write requests. The solid oval depicts the boundary of the read/write atomic memory. The client automata of Figure 3-1 are here the OM Clients (Operation Manager Clients), and the virtual objects are put/get objects.

the objects and retrieve information from the objects, thus exchanging information.

Each read and write operation uses a different port on the virtual objects, so that earlier operations do not interfere with later ones. (In fact, an operation may use two different ports for two different phases during the same operation.) This allows for improved performance, since even if a given object is very slow during one operation (perhaps never responding, due to a lost or severely delayed message), it may be used in a later operation.

Figure 6-1 depicts the various components of the GeoQuorums Operation Manager. The dashed black oval represents the boundary interface of the GeoQuorums Operation Manager. Notice that this interface includes three operations: read, write and recon. Our goal is to show that the GeoQuorums algorithm implements an atomic read/write object; the recon interface should be hidden from the external environment that will use this as a read/write object.

We assume that there exists a set of reconfiguration automata, one for each mobile node, which we call $ReconClient_i$, for all i in I. These reconfiguration automata generate recon requests, and they receive recon-ack responses. They are not a part of the algorithm presented in this chapter, but rather a component specified by a client of the GeoQuorums algorithm. We place only one restriction on the *ReconClient* automata: the reconfiguration clients are required to respect the environmental well-formedness requirement that a recon request is issued at node i only if there is no ongoing read, write, or reconfiguration at node i. Notice that there still may be concurrent operations at other nodes. (It would be a relatively simple modification to completely decouple the *ReconClient* automata from the read/write environment, by allowing concurrent reconfigurations and read/write operations, as is done in [38, 73]. We impose this restriction primarily for simplicity of presentation.)

When the GeoQuorums Operation Manager is composed with both the *ReconClient* automata and the virtual objects, the recon and recon-ack actions are hidden, as are the invoke and the respond actions on the put/get objects. This results in an external interface consisting only of read/read-ack actions and write/write-ack actions, as depicted by the solid black oval in Figure 6-1. This matches the external signature of a read/write object, as specified in Figure A-1.

We begin in Section 6.1 with some preliminaries, defining "configurations," "quorums," etc. We proceed in Section 6.2 to present the detailed implementation of the GeoQuorums Operation Manager. We prove the protocol correct in Section 6.3, and present some (conditional) performance results in Section 6.4. We discuss some further extensions in Section 6.5.

An extended abstract of this chapter appeared in the 17th International Symposium on Distributed Computing (DISC 2003) [32], and a full version appeared in Distributed Computing [33].

6.1 Preliminaries

In this section we present some preliminary definitions. (See Figure 2-1 for a summary of notation used in Part I of this thesis.)

A **configuration**, c, consists of three components: *members*(c), *get-quorums*(c), and *put-quorums*(c). The set *members*(c) $\subseteq O$ is a set of virtual identifiers, and determines which virtual objects are part of the configuration. (Recall that each virtual has an identifier in O.)

The sets *put-quorums*(c) and *get-quorums*(c) are collections of quorums; each quorum is a set of virtual identifiers. Each virtual object in a quorum is a member of the configuration. That is, each quorum is a subset of *members*(c).

Every get-quorum intersects every put-quorum. That is, if $G \in$ *get-quorums*(c) and $P \in$ *put-quorums*(c), then

$$G \cap P \neq \emptyset .$$

Moreover, if any f virtual objects fail, then at least one get-quorum and one put-quorum survive intact. That is, for any set F of f virtual object identifiers, there exists a quorum $G \in$ *get-quorums*(c) and $P \in$ *put-quorums*(c) such that:

- $F \cap G = \emptyset$
- $F \cap P = \emptyset$.

Thus, an algorithm based on these quorums can tolerate f focal points failing. We assume for the purpose of this chapter that the Virtual Object Layer ensures that no more than f virtual objects fail in any execution.

We assume a fixed set of configurations that is finite, ordered, and known in advance to all mobile nodes. Each configuration is assigned a name in M (the set of configuration names).

Each configuration proposal is identified by a tuple of three components: a time when the configuration is proposed (according to a local clock), the node ($\in I$) that proposed the configuration, and the name of the configuration ($\in M$) that is being proposed. We refer to such a tuple (i.e., an element of C) as a *configuration identifier*.

As long as no mobile node proposes more than one configuration at a given instant, then every configuration identifier (i.e., every proposal) created during an execution is unique. The configuration identifiers are ordered lexicographically, based first on comparing the time components, then comparing the process identifiers, and then comparing the configuration names.

Practical Aspects.

We propose one set of configurations that may be particularly useful in practical implementations. In this case, we use two configuration c_0 and c_1. We take advantage of the fact that accessing nearby focal points is usually faster than accessing distant focal points. The focal points can be grouped into clusters, using some geographic technique [26]. Figure 6-2 illustrates the relationship among mobile nodes, focal points, and clusters. For configuration c_0, the *get-quorums* are defined to be the clusters. The *put-quorums* consist of every set containing one focal point from each cluster. Configuration c_1 is defined in the opposite manner. Assume, for example, that

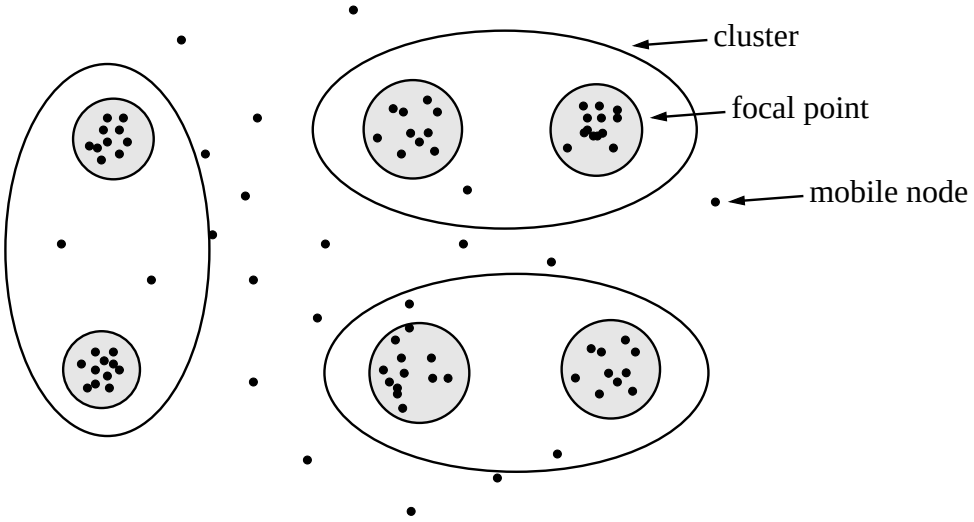

Figure 6-2: Clusters of focal points, each of which represents a virtual object. Each cluster comprises a *get-quorum*, while each set containing one focal point from each cluster comprises a *put-quorum*.

read operations are more common than write operations (and most read operations only require one phase). If the clusters are relatively small and are well distributed (so that every mobile node is near to every focal point in some cluster), then configuration c_0 is quite efficient. On the other hand, if write operations are more common than read operations, configuration c_1 is quite efficient. Our algorithm allows the system to switch safely between two such configurations.[1]

Another difficulty in implementation might be agreeing on the focal points and ensuring that every mobile node has an accurate list of all the focal points and configurations. Some strategies have been proposed to choose focal points: for example, the mobile nodes might send a token on a random walk, to collect information on geographic density [28]. The simplest way to ensure that a mobile node has access to a list of focal points and configurations is to depend on a centralized server, through transmissions from a satellite or a cell-phone tower. Alternatively, the GeoCast service itself might facilitate finding other mobile nodes, at which point the definitive list can be discovered.

6.2 GeoQuorums Operation Manager

In this section we present the GeoQuorums Operation Manager (OM), an algorithm built on the Virtual Object Layer. As the Virtual Object Layer contains two entities, virtual objects and mobile nodes, we present two specifications, one for the objects (each depicted as a "virtual object" in Figure 3-1) and one for the clients running on the mobile nodes (each depicted as a "client" in Figure 3-1):

- put/get *variable type* (Figure 6-3): the variable type of the virtual objects in the Virtual Object Layer.
- *Operation Manager Client* (Figure 6-4, 6-5, 6-6, and 6-7): an automaton that receives read, write, and recon requests from clients and manages quorum accesses to implement these operations.

Figure 6-1 depicts the various GeoQuorums Operation Manager components. The GeoQuorums Operation Manager (OM) is the collection of all the Operation Manager Clients (OM_i, for all i in I). It is composed with the virtual objects, each of which is an atomic object with the put/get variable type.

6.2.1 The put/get Variable Type

The put/get variable type supports four operations: put, get, confirm, and recon-done. The variable type is specified in Figure 6-3. The put and get operations are used to set and retrieve the value. We first describe the various state components of the variable type, and then explain the different operations and how they modify the state.

[1] Since the original publication of the GeoQuorums protocol in [32], there has been further research into cluster-based quorum systems for geographic networks [18].

Figure 6-3: Definition of the put/get variable type τ.

```
1   State:
2       tag ∈ T, initially ⟨0, i_0⟩
3       value ∈ V, initially v_0
4       config-id ∈ C, initially ⟨0, i_0, c_0⟩
5       confirmed-set ⊆ T, initially ∅
6       recon-ip, a Boolean, initially false
7
8   Operations:
9
10  put(new-tag, new-value, new-config-id)
11      if (new-tag > tag) then
12          value ← new-value
13          tag ← new-tag
14      if (new-config-id > config-id) then
15          config-id ← new-config-id
16          recon-ip ← true
17      return put-ack(config-id, recon-ip)
18
19  get(new-config-id)
20      if (new-config-id > config-id) then
21          config-id ← new-config-id
22          recon-ip ← true
23      confirmed ← (tag ∈ confirmed-set)
24      return get-ack(tag, value, confirmed, config-id, recon-ip)
25
26  confirm(new-tag)
27      confirmed-set ← confirmed-set ∪ {new-tag}
28      return confirm-ack()
29
30  recon-done(new-config-id)
31      if (new-config-id = config-id) then
32          recon-ip ← false
33      return recon-done-ack()
```

Variable Type State Components. The put/get variable type is used to maintain a *value*, which is therefore the primary component of its state. The variable type also contains a *tag* component in its state. Each tag consists of a nonnegative real number (the time at which the tag was determined) and a unique process identifier (i.e., $T = \mathbb{R}^{\geq 0} \times I$, see Figure 2-1). A tag is associated with every value, and the tags determine an ordering on the values that are stored by the put invocations (the only invocations that modify the *value* component of the state). Ordering these values allows us to order the high-level write operations that create these values, which is necessary for guaranteeing atomic consistency. The put and get invocations take a configuration identifier, *new-config-id*, as a parameter (Figure 6-3, lines 10 and 19). The put/get variable type includes a *config-id* in its state, corresponding to the largest configuration identifier that any put or get invocation has used. The *confirmed-set* is a set of tags, indicating whether a tag has been confirmed. We explain later in Section 6.2.2 when a tag is confirmed. The *recon-ip* flag indicates whether the virtual object believes that a reconfiguration is in progress; this is set to true when the object learns about a new configuration, and is set to false when a recon-done indicates that the configuration is fully installed.

Variable Type Transitions. The put/get variable type supports four types of invocations and responses. A put invocation includes three parameters: the *new-value*,

Figure 6-4: Operation Manager Automaton: Signature and State
for node i where τ is the put/get variable type.

1 **Input**:
2 write$(val)_i$, $val \in V$
3 read$()_i$
4 recon$(cid)_i$, $cid \in C$
5 respond$(resp)_{obj,p}$, $resp \in responses(\tau)$, $obj \in O$, $p = \langle *, *, i \rangle \in S$
6 geo-update$(t, l)_i$, $t \in R^{\geq 0}$, $l \in L$
7
8 **Output**:
9 write-ack$()_i$
10 read-ack$(val)_i$, $val \in V$
11 recon-ack$(cid)_i$, $cid \in C$
12 invoke$(inv)_{obj,p}$, $inv \in invocations(\tau)$, $obj \in O$, $p = \langle *, *, i \rangle \in S$
13
14 **Internal**:
15 read-2$()_i$
16 recon-2$(cid)_i$, $cid \in C$
17
18 **State**:
19 *confirmed* $\subseteq T$, a set of tag ids, initially $\emptyset$
20 *conf-id* $\in C$, a configuration id, initially $\langle 0, i_0, c_0 \rangle$
21 *recon-ip*, a Boolean flag, initially false
22 *clock* $\in \mathbb{R}^{\geq 0}$, a time, initially 0
23 *ongoing-invocations* $\subseteq O \times S$, a set of objects and ports, initially $\emptyset$
24 *current-port-number* $\in \mathbb{N}^{>0}$, used to invoke objects, initially 1
25 *op*, a record with the following components:
26 *type* $\in$ {read,write,recon}, initially read
27 *phase* $\in$ {idle,get,put}, initially idle
28 *tag* $\in T$, initially $\langle 0, i_0 \rangle$
29 *value* $\in V$, initially v_0
30 *recon-ip*, a Boolean flag, initially false
31 *recon-conf-id* $\in C$, a configuration id, initially $\langle 0, i_0, c_0 \rangle$
32 *acc* $\subseteq O$, a set of data objects, initially $\emptyset$

a value to be stored in the state, the *new-tag*, a tag, and *new-config-id*, a configuration identifier. The put invocation modifies the *value* component of the state only if the invocation's tag, *new-tag*, is larger than the *tag* stored in the state (i.e., the tag of the last successful put invocation, Figure 6-3, line 11). The put invocation also modifies *config-id* if the invocation's configuration identifier, *new-config-id*, is larger than the identifier *config-id* stored in the state (line 14). Whenever the put invocation causes *config-id* to be modified, we assume that a reconfiguration is in progress and set *recon-ip* to true (line 16).

A put invocation results in a put-ack response. The response includes the configuration identifier stored in the state, *config-id*, and an indication of whether a reconfiguration is in progress, *recon-ip*.

A get invocation takes a single parameter: *new-config-id*, a configuration identifier. The get modifies the state only if the invocation's configuration identifier, *new-config-id*, is larger than the *config-id* stored in the state. In this case, the objects' configuration identifier, *config-id*, is set to the invocation's configuration identifier, *new-config-id*. As in the case of a put invocation, if the *config-id* is modified, *recon-ip* is set to true.

A get invocation results in a get-ack response. This response includes the *tag* and *value* stored in the state, as well as the *config-id* and an indication of whether a reconfiguration is in progress, that is, *recon-ip*. It also includes a boolean flag

Figure 6-5: Operation Manager invoke **Transitions for node** i

```
1   Output invoke(⟨get, config-id⟩)_{obj,p}
2   Precondition:
3       p = ⟨current-port-number, get, i⟩
4       ⟨obj, p⟩ ∉ ongoing-invocations
5       obj ∉ op.acc
6       op.phase = get
7       config-id = conf-id
8   Effect:
9       ongoing-invocations ← ongoing-invocations ∪ {⟨obj, p⟩}
10
11  Output invoke(⟨put, tag, val, config-id⟩)_{obj,p}
12  Precondition:
13      p = ⟨current-port-number, put, i⟩
14      ⟨obj, p⟩ ∉ ongoing-invocations
15      obj ∉ op.acc
16      op.phase = put
17      tag = op.tag
18      val = op.value
19      config-id = conf-id
20  Effect:
21      ongoing-invocations ← ongoing-invocations ∪ {⟨obj, p⟩}
22
23  Output invoke(⟨confirm, tag⟩)_{obj,p}
24  Precondition:
25      p = ⟨k, confirm, i⟩
26      ⟨obj, p⟩ ∉ ongoing-invocations
27      tag ∈ confirmed
28  Effect:
29      ongoing-invocations ← ongoing-invocations ∪ {⟨obj, p⟩}
30
31  Output invoke(⟨recon-done, config-id⟩)_{obj,p}
32  Precondition:
33      p = ⟨k, recon-done, i⟩
34      ⟨obj, p⟩ ∉ ongoing-invocations
35      recon-ip = false
36      config-id = conf-id
37  Effect:
38      ongoing-invocations ← ongoing-invocations ∪ {⟨obj, p⟩}
```

indicating whether the tag is confirmed. That is, it returns true if the *tag* is in the *confirmed-set* stored in the state. Effectively, this indicates whether a confirm invocation has previously indicated that the tag is confirmed.

A confirm invocation takes one parameter: a *new-tag*. The *confirmed-set* component of the state is modified, adding the tag *new-tag* to this set. The confirm invocation results in a confirm-ack response.

A recon-done invocation includes a single parameter: a *new-config-id*, a configuration identifier. The *recon-ip* component of the state is modified if the configuration identifier, *new-config-id*, matches the *config-id* stored in the state. In this case, *recon-ip* is set to false. This indicates that the configuration associated with that configuration identifier is installed, that is, that the reconfiguration that proposed the configuration identifier is complete. This invocation results in a recon-done-ack response. (Note that if *new-config-id* is not equal to *config-id*, stored in the state, the invocation is ignored. While it may improve performance to allow the recon-done action to modify *config-id*, we do not do this in the interests of simplicity.)

Figure 6-6: Operation Manager respond **Transitions for node** i

```
40  Input respond(⟨get-ack, tag, val, confirmed, new-cid, new-rip⟩)_{obj,p}
41  Effect:
42      if (⟨current-port-number, get, i⟩ = p) then
43          op.acc ← op.acc ∪ {obj}
44          if (tag > op.tag) then
45              op.tag ← tag
46              op.value ← val
47          if (new-cid > conf-id) then
48              conf-id ← new-cid
49              op.recon-ip ← true
50              recon-ip ← new-rip
51          else if (new-cid = conf-id) then
52              recon-ip ← recon-ip ∧ new-rip
53          if (confirm = true) then
54              confirmed ← confirmed ∪ {tag}
55      ongoing-invocations ← ongoing-invocations − {⟨obj, p⟩}
56
57  Input respond(⟨put-ack, new-cid, new-rip⟩)_{obj,p}
58  Effect:
59      if (⟨current-port-number, put, i⟩ = p) then
60          op.acc ← op.acc ∪ {obj}
61          if (new-cid > conf-id) then
62              conf-id ← new-cid
63              op.recon-ip ← true
64              recon-ip ← new-rip
65          else if (new-cid = conf-id) then
66              recon-ip ← recon-ip ∧ new-rip
67      ongoing-invocations ← ongoing-invocations − {⟨obj, p⟩}
68
69  Input respond(⟨confirm-ack⟩)_{obj,p}
70  Effect:
71      ongoing-invocations ← ongoing-invocations − {⟨obj, p⟩}
72
73  Input respond(⟨recon-done-ack⟩)_{obj,p}
74  Effect:
75      ongoing-invocations ← ongoing-invocations − {⟨obj, p⟩}
```

6.2.2 Operation Manager Client Specification

The Operation Manager Client uses the atomic objects (with the put/get variable type) provided by the Virtual Object Layer as replicas, invoking put operations to update the virtual objects and get operations to retrieve the value (and associated information) from the virtual objects. Replication allows the Operation Manager Clients to guarantee fault-tolerance, tolerating the failure of up to f virtual objects. Figure 6-1 depicts the implementation of a read/write atomic object in the Virtual Object Layer.

Signature. We first describe the signature of the Operation Manager Client, contained in Figure 6-4. The external signature consists of read actions, write actions, and recon actions, to initiate the appropriate operations, and read-ack, write-ack, and recon-ack actions to indicate that the operation have completed.

The external signature also includes invoke and respond actions, to communicate with the virtual objects. Each of these actions is performed on some port, p, for some object, obj.

There are also two internal actions: read-2 and recon-2. The first of these begins the second phase of a read operation. The latter begins the second phase of a

reconfiguration operation. (We describe these operations later in this section.)

The Operation Manager Clients are composed with two sets of automata: the virtual objects and the *ReconClient* automata. The invoke/respond actions allow the Operation Manager to communicate with the virtual objects. The recon and recon-ack allow the Operation Manager to communicate with the *ReconClient* automata.

State. The state of an Operation Manager Client consists of two parts: some general state that is maintained throughout the execution, and the *op* record, which maintains state specific to an ongoing operation.

The *confirmed-set* is a set of tags associated with operations that have completed. That is, if a tag is in *confirmed-set*, then some read or write operation associated with that tag has completed.

The *conf-id* is the largest configuration identifier that the Operation Manager Client has received. The Operation Manager Client receives configuration identifiers from respond actions for get-ack and put-ack responses.

The *recon-ip* flag indicates whether the Operation Manager Client believes that a reconfiguration is in progress. The Operation Manager Client sets this flag to true whenever it receives a new, larger configuration identifier (from a respond action), and sets it to false when it receives an indication that the reconfiguration is complete (also from a respond action).

The *clock* is the current real time, as specified by the Geosensor component of the RealWorld.

The *ongoing-operations* is a set of objects and ports, indicating that an operation has been invoked on the specified port of that object, and that a response has not yet occurred. This is used to ensure the well-formedness condition that atomic objects require: there is only one operation ongoing at any given time on a given port of a given object.

Each invocation of a virtual objects uses a port, which consists of a sequence number, an operation identifier, and a node identifier. The *current-port-number* stores the sequence number component of the port. An invocation by node i, then, uses the port $\langle \textit{current-port-number}, op, i \rangle$, where *op* is either put, get, confirm, or recon-done.

Every time a new phase of an operation is begun, the *current-port-number* is incremented. Since only one operation can take place on a port at a time, incrementing the port number allows the new phase to perform invocations, even if old invocations on the prior port have not completed.

The *op* record maintains information specific to a given operation. The *op.type* field indicates the type of the ongoing operation. The *op.phase* field indicates the phase of the operation. (Operations may go through two phases: a get phase and a put phase; a write operation performs only a put phase.) The *op.tag* field indicates the largest tag discovered during the get phase of an operation. The *op.value* field indicates the value associated with that tag.

The *op.recon-ip* field indicates whether a reconfiguration is in progress. Notice that, unlike the general *recon-ip* flag, the *op.recon-ip* flag is never reset to false until the phase completes. Once a reconfiguration occurs concurrently with some phase of

an operation (and some Operation Manager Client receives information about this reconfiguration), the *op.recon-ip* flag is set to true for the rest of the phase.

The *op.recon-conf-id* field is used to keep track of the configuration being installed by an ongoing reconfiguration. While the reconfiguration occurs, a new reconfiguration may be initiated at some other mobile node. This may cause the node's configuration identifier, *conf-id*, to be modified. The *op.recon-conf-id*, however, is not modified until the ongoing reconfiguration is complete.

The *op.acc* set is an accumulator that maintains the set of object identifiers of objects that have performed a respond during the phase of an operation. A phase completes when *op.acc* contains a large enough set of object identifiers; in particular, it completes when object identifiers that are associated with appropriate quorums are contained within the *op.acc* set.

Read/Write Operations. The code for read/write operations is presented in Figures 6-5, 6-6, 6-7, and 6-8. We first explain how a write operation proceeds, and then go on to explain read operations.

Each read and write operation consists of either one or two phases. A write operation requires only a single phase, a "put phase" that propagates the new value to at least one quorum of virtual objects. Some read operations require only a single phase, a "get phase", that retrieves the value from at least one quorum of virtual objects. Other read operations require two phases: a "get phase", that retrieves the value, followed by a "put phase", that propagates the retrieved value.

Assume that the read or write operation is initiated at node i. During each phase of the operation, node i invokes put and get operations on the virtual objects. Each invocation and subsequent response uses a port.

Each phase of each operation uses a unique port. When a phase begins, node i chooses a new port to use during that phase by incrementing the *current-port-number* (for example, Figure 6-8, line 51).

The choice of port serves two purposes. First, it ensures that the Operation Manager Client respects the well-formedness requirement of the virtual objects. Well-formedness requires that only one operation may occur at a time on each port of each object. By choosing a new port for each phase, we ensure that node i can perform invocations during that phase, regardless of earlier ongoing operations without violated well-formedness.

Second, the use of a unique port, p, for each phase allows node i to be sure that any response received on port p is the result of an invocation during the phase associated with port p. Any response on any other port (i.e., a port that is not identified by *current-port-number*) is ignored (see Figure 6-5, line 42, for example), since it results from an earlier (completed) phase.

A write operation begins at node i when OM_i receives a write$(val)_i$ request. Node i then examines its clock to choose a new tag for the operation (Figure 6-8, line 52). For example, if the write is initiated at time t, then the tag is chosen to be $\langle t, i \rangle$. At this point, the *current-port-number* is incremented, choosing port p for this phase of the operation.

The OM_i automaton then begins a put phase, which performs put invocations on the virtual objects (Figure 6-5, lines 11–21). We allow invocations to happen, nondeterminstically, on all the virtual objects. In most cases, of course, there is no need to contact all the virtual objects; it is simpler, however, to allow nondeterministic invocations and assume that an optimized implementation may impose further restrictions.

The phase completes when the OM_i automaton receives "sufficient" responses from the objects on port p (Figure 6-5, lines 57–67). Assume that when the operation begins, the automaton is in the configuration identified by $cid = \langle *, *, c \rangle$ (i.e., $cid = \textit{conf-id}_i$). If all responses indicate that c is the most recent configuration identifier and no reconfiguration is in progress, then the operation terminates when OM_i receives at least one response from each object in some put-quorum, $P \in \textit{put-quorums}(c)$ (Figure 6-8, line 60).

On the other hand, if any response indicates that a reconfiguration is in progress, then OM_i waits until it receives responses from objects in quorums of every configuration. Specifically, the phase completes when for every configuration c' in M, there is some quorum, $P \in \textit{put-quorums}(c')$ such that every object in P has responded to node i during the phase (Figure 6-8, line 58).

After the operation the OM_i may notify objects that the tag has been *confirmed*, indicating that the previous operation is complete (Figure 6-5, lines 23–29). The confirm invocation uses the port $\langle \textit{current-port-number}, \mathsf{confirm}, i \rangle$, thus ensuring that it does not conflict with put and get invocations.

A read request can complete in one of two ways: if the value being read has been confirmed, the operation completes in one phase; otherwise, the operation completes in two phases. When the OM_i automaton receives a read request, it first begins a get phase (Figure 6-7, line 4) and performs get invocations on the atomic objects (Figure 6-5, lines 1–9). Again, assume that when the operation begins, the automaton is in the configuration identified by $cid = \langle *, *, c \rangle$ (i.e., $cid = \textit{conf-id}_i$).

If all responses indicate that c is the most recent configuration identifier, then the get phase terminates when OM_i receives a response from each object in some quorum $G \in \textit{get-quorums}(c)$. Otherwise, the phase completes when for every configuration c' in M, OM_i receives a response from each object in some quorum $G \in \textit{get-quorums}(c')$.

At this point, OM_i chooses the value associated with the largest tag from any of the responses and determines if the operation is complete, or whether a second phase is necessary. If the chosen tag has been *confirmed*, then the operation completes (Figure 6-7, lines 6–16).

Otherwise, OM_i begins a second phase, a put phase. The put phase is similar to the protocol for the write operation (Figure 6-7, lines 18–31): the *current-port-number* is incremented, choosing port p for this phase of the operation; the OM_i automaton then begins a put phase, which performs put invocations on the virtual objects (Figure 6-5, lines 11–21); the phase completes when the OM_i automaton receives "sufficient" responses from the objects on port p (Figure 6-5, lines 57–67).

The knowledge of the *confirmed* tags is used to short-circuit the second phase of certain read operations. The second phase is required only when a prior operation with the same tag has not yet completed. By notifying objects when the tag has been

confirmed, the algorithm allows later operations to discover that a second phase is unnecessary.

Reconfiguration. The code for the reconfiguration algorithm is presented in Figure 6-5 (where Figure 6-5, lines 31–38 are used by the reconfiguration mechanism, while the rest is used also by the read/write mechanism) and Figure 6-8, lines 66–93.

The reconfiguration algorithm differs from the reconfiguration processing presented in the RAMBO algorithm [38, 73]. The new algorithm eliminates the *Recon* service and the associated consensus service, while limiting the number of configurations the system can support. In RAMBO, an arbitrary new configuration can be proposed, while upgrading to the new configuration requires knowledge about all active preceding configurations. The *Recon* service in RAMBO uses consensus to agree on the order of configurations, while the configuration-upgrade operation in RAMBO uses the knowledge of the order and local information about active configurations.

The new reconfiguration algorithm works with a known finite set of possible configurations. The algorithm does not use consensus because all possible preceding configurations are known. The configuration identifiers determine a total ordering on the installed configurations, however it is not necessary that a mobile node be aware of all prior configuration identifiers in the total order. It is sufficient for the reconfiguration algorithm to simply contact all configurations in order to ensure that all configurations preceding it in the total order are contacted. Because this simplification obviates the need for a consensus service, it significantly improves efficiency.

A reconfiguration operation is a two-phase operation similar to a two-phase read operation; it includes a get phase and a put phase. In each phase it requires contacting appropriate quorums of objects from certain configurations.

A reconfiguration begins when the Operation Manager Client receives a $\mathsf{recon}(c)$ input, where c names one of the configurations in M. For the sake of this discussion, assume that the recon is initiated at mobile node i.

First, the Operation Manager Client chooses a new, unique configuration identifier, by examining the local clock, and using its node identifier (i.e., node i) and the name of the new configuration (i.e., configuration c). Specifically, if the $\mathsf{recon}(c)_i$ occurs at time t, then the configuration identifier is $cid = \langle t, i, c\rangle$ (Figure 6-8, line 68). At the same time, node i sets its $\textit{conf-id}_i$ to the new configuration identifier ($\langle t, i, c\rangle$) and sets $\textit{recon-ip}_i$ to true, to indicate that a reconfiguration is in progress (Figure 6-8, line 69).

The OM_i then chooses a new port for the operation, incrementing the counter tracking the current port number, i.e., $\textit{current-port-number}_i$ (Figure 6-8, line 70). This event starts a get phase. During the get phase, several $\mathsf{invoke}(\mathsf{get}, \ldots)_{obj,p}$ events occur (Figure 6-5, lines 1–9) for objects *obj* in quorums of all configurations in M.

When a $\mathsf{respond}(\mathsf{get\text{-}ack}, \ldots)_{obj,p}$ event occurs (on the same port p), *obj* is added to $op.acc$. The phase completes when i has received a response from every object in at least one put-quorum and one get-quorum of each configuration in M.

At this point, a $\mathsf{recon\text{-}2}(cid)_i$ event occurs (Figure 6-8, lines 73–82) and the Operation Manager Client chooses a new port, p' (Figure 6-8, line 80). This event

begins the put phase. During the put phase, several $\mathsf{invoke}(\mathsf{put}, \ldots)_{obj,p'}$ events occur (Figure 6-5, lines 11–21) for objects *obj* in quorums of the new configuration, *c*.

When a $\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_{obj,p'}$ event occur (on the same port p'), *obj* is added to *op.acc*. The phase completes when node *i* has received responses from every object in at least one put-quorum of the new configuration, *c* (Figure 6-8, line 88).

At this point, a $\mathsf{recon\text{-}ack}(cid)_i$ event occurs (Figure 6-8, lines 84–93), ending the reconfiguration.

If $conf\text{-}id_i$ is equal to *op.recon-conf-id*, then $recon\text{-}ip_i$ is set to false (Figure 6-8, line 92). Otherwise, a new configuration with a larger configuration identifier has been discovered by node *i*, and a reconfiguration for this new configuration identifier may be in progress elsewhere. Therefore, in this case, $recon\text{-}ip_i$ is left unchanged.

When a reconfiguration is not in progress, node *i* may notify virtual objects that the reconfiguration for a certain configuration identifier is done, with recon-done invocations (Figure 6-5, lines 31–38).

Finally, notice that the reconfiguration algorithm proceeds in the same way, regardless of whether the newly proposed configuration (i.e., the configuration with name *c*) is the same as the old configuration: whenever the new configuration identifier is different from the old one, a reconfiguration occurs.

6.3 Analysis of the Operation Manager

In this section, we show that the Operation Manager guarantees atomic consistency. We show that the Operation Manager correctly implements an atomic read/write object by showing that a partial ordering of operations exists with the properties required by Theorem A.3.3. We first define some notation, in Section 6.3.1. We then define a partial order, in Section 6.3.2. Next, we prove some preliminary lemmas, in Section 6.3.3. We then outline the main part of the proof in Section 6.3.4, and then move on to the main body of the proof in Section 6.3.5.

6.3.1 Notation

We first define some notation that we use during the proof. Throughout this section, we fix α to be an execution of the entire system: the Operation Manager, the virtual objects, the reconfiguration clients, and the well-formed environment, U. Additionally, we assume that every read and write operation in α completes. Let Π be the set of read and write operations in α.

There are two ways in which a read operation may conclude: after two phases (see Figure 6-7, lines 33–43), or after a single phase (see Figure 6-7, lines 6–14). In the first case, at the end of the read operation when the read-ack occurs, $op.phase = \mathsf{put}$, indicating that a "put" phase has completed. In the second case, at the end of the read operation, $op.phase = \mathsf{get}$, indicating that only a single phase, a "get" phase, has completed. In this case, the tag, *op.tag*, is in the set *confirmed* immediately before the read completes, so the operation completes after only the get phase.

Every read operation begins with a read action and ends with a read-ack action. We say that a read operation $\pi \in \Pi$ that takes place at node i is a *two-phase read* operation if a read-2$_i$ event occurs between the read$_i$ event and read-ack$_i$ event. Operation π is a *one-phase read* operation if no read$_{2i}$ event occurs.

We now associate a configuration identifier with each phase of a read or write operation, π, based on the value of the *conf-id* of the operation's initiator at the end of that phase. Specifically, if π is a one-phase read operation initiated by node i, then the "get configuration" of π, *get-conf-id*(π), is the value of *conf-id*$_i$ when π's read-ack$_i$ event occurs, ending the get phase. If π is a two-phase read operation, then *get-conf-id*(π) is the value of *conf-id*$_i$ when operation π's read-2$_i$ event occurs, ending the get phase. (If π is a write operation, then π has no get phase, so *get-conf-id*(π) is undefined.)

If for some operation π, *get-conf-id*(π) = $\langle t, i, c \rangle$, then we define *get-conf*(π) to be c, the name of the configuration identified by the *get-conf-id*(π). We say the *get-conf*(π) is the "get configuration" of π.

If π is a two-phase read operation (respectively, a write operation), then the "put configuration identifier" of π, the *put-conf-id*(π), is the value of the configuration identifier *conf-id*$_i$ when π's read-ack$_i$ (respectively, write-ack$_i$) event occurs. If π is a reconfiguration operation, then the configuration identifier *put-conf-id*(π) is equal to *op.conf-id*$_i$ when the *recon-ack* event occurs. (If π is a one-phase read operation, then π has no put phase, so *put-conf-id*(π) is undefined.)

If for some operation π, *put-conf-id*(π) = $\langle t, i, c \rangle$, then we define *put-conf*(π) to be c, the name of the configuration identified by the *put-conf-id*(π). We say the *put-conf*(π) is the "put configuration" of π.

Next, we associate a "recon-in-progress" flag with each phase of a read or write operation, based on the value of *op.recon-ip* at the end of that phase. Specifically, if π is a one-phase read operation initiated by node i, then we define *get-rip*(π) to be the value of *op.recon-ip*$_i$ when operation π's read-ack$_i$ event occurs, ending the get phase. If π is a two-phase read operation, then *get-rip*(π) is the value of *op.recon-ip*$_i$ when π's read-2$_i$ event occurs, ending the get phase.

If π is either a two-phase read operation or a write operation, then we define *put-rip*(π) to be equal to the value of the *op.recon-ip*$_i$ flag when π's read-ack$_i$ or write-ack$_i$ event occurs.

The *get-rip* and *put-rip* flags indicate whether node i detects a reconfiguration in progress during the get or put phase of the operation. It is sufficient to consider the value of the *op.recon-ip* flag at the end of the phase, since the flag is never set to false during the phase: none of the invoke/respond actions set *op.recon-ip* to false, only the write, read, read-2, recon, and recon-2 event might have this effect, if *recon-ip* is true. If at any time during the phase *recon-ip* is set to true, which happens only during a respond event, then *op.recon-ip* is set to true at the same time (for example, see Figure 6-5, lines 49 and 50), and it is therefore true at the end of the phase.

During the proof, if s is a state during the execution and *obj* is a virtual object, we use the terminology *s.obj* to refer to the state of the object. If x is a component of the state of the object, we use the terminology *s.obj.field* to refer to the *field* component of the object. For example, *s.obj.tag* refers to the *tag* of the object *obj* in state s.

6.3.2 Partial Order

We now construct an appropriate partial ordering, and then show that it meets the necessary requirements of Theorem A.3.3. For a read or write operation, $\pi \in \Pi$, initiated at mobile node i, we define $tag(\pi)$ as follows: $tag(\pi) = op.tag_i$ immediately after the acknowledgment of π occurs, that is, when the $\mathsf{read\text{-}ack}_i$ or $\mathsf{write\text{-}ack}_i$ event occurs. (In fact, the tag is often fixed earlier in the operation, as we show in Lemma 6.3.1.) For a reconfiguration operation, ρ, we define $tag(\rho) = op.tag_i$ immediately after the $\mathsf{recon\text{-}2}$ event occurs. We then define the partial order $\prec$:

- For any two operations π_1 and π_2:

$$\text{if } tag(\pi_1) < tag(\pi_2) \text{ then } \pi_1 \prec \pi_2 \ .$$

- For any write operation π_1, and any read operation π_2:

$$\text{if } tag(\pi_1) = tag(\pi_2) \text{ then } \pi_1 \prec \pi_2 \ .$$

We show in Theorem 6.3.16 that this partial order, $\prec$, satisfies the three conditions of Theorem A.3.3. The key condition to prove about the partial ordering is that it is consistent with the ordering of operations in α. That is, we need to show Property 2 of Theorem A.3.3, that if π_1 and π_2 are two operations, and π_1 completes before π_2 begins, then π_2 does not precede π_1 in the partial order.

6.3.3 Preliminary Lemmas

Before beginning the main part of the proof, we prove a few preliminary lemmas. First we examine when during an operation the tag of the operation is fixed. Then we prove some general lemmas about the propagation of tags and values during a put phase and the retrieval of tags and values during a get phase.

Recall that for operation π at node i, $tag(\pi)$ is defined as the value of $op.tag_i$ when the operation completes. In fact, if the operation has a put phase, the tag is fixed prior to the put phase of the operation.

Lemma 6.3.1. *If π is a write operation at node i, then $tag(\pi) = op.tag_i$ immediately after the write_i event. If π is a two-phase read operation, then $tag(\pi) = op.tag_i$ immediately after the $\mathsf{read\text{-}2}_i$ event.*

Proof. Assume π is a write operation. In this case, OM_i performs only put invocations. Notice that the response action, $\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_i$, does not update $op.tag_i$ Therefore $op.tag_i$ does not change after the write_i event until the $\mathsf{write\text{-}ack}_i$ event that concludes the operation and defines the $tag(\pi)$.

Assume π is a read operation. Similarly, after the $\mathsf{read\text{-}2}_i$ event, the OM_i only performs put invocations, so again the tag $op.tag_i$ does not change after the $\mathsf{read\text{-}2}_i$ event, until the $\mathsf{read\text{-}ack}_i$ that concludes the operation and defines the $tag(\pi)$. □

We next note that the *tag* component of the virtual object's state is nondecreasing:

Lemma 6.3.2. *For every virtual object, obj, the tag of obj is nondecreasing. If s and s' are two states during execution α, and s precedes s', then $s.obj.tag \leq s'.obj.tag$.*

Proof. Immediate by examination of the code that modifies *tag*. The *tag* is modified only in Figure 6-3, line 13, which is executed only if *new-tag* > *tag*. □

Next we consider how tag information is propagated during read and write operations to virtual objects. We show that after the put phase of an operation completes, there exists a specific quorum of objects each of which has a tag no smaller than that of the operation.

Lemma 6.3.3. *Let π be a two-phase read operation, a write operation, or a reconfiguration that occurs at node i. Then there exists a put-quorum, P, in put-conf(π) such that for every object, obj, in P, $tag(\pi) \leq obj.tag$ anytime after π completes.*

Proof. This lemma follows from the termination condition of the put phase of an operation. Assume that when the put phase of π begins (i.e., immediately after the write, read-2, or recon-2 event), $p = \langle current\text{-}port\text{-}number, \mathsf{put}, i\rangle$, the port number that is used throughout the phase. Also, assume that $cid = put\text{-}conf\text{-}id(\pi) = \langle *, *, c\rangle$.

We divide the proof of into two subcases: the case where $put\text{-}rip(\pi) = \mathsf{false}$, and where $put\text{-}rip(\pi) = \mathsf{true}$.

First, consider the case where $put\text{-}rip(\pi) = \mathsf{false}$. Recall that if π is a read or write operation, then the configuration identifier *cid* (which is equal to $put\text{-}conf\text{-}id(\pi)$) is equal to the configuration identified by $conf\text{-}id_i$ when the operation completes; if π is a reconfiguration, then *cid* is equal to the configuration identifier $op.conf\text{-}id_i$ when the operation completes. (Notice that our use of *c* is consistent with the notation used in Figure 6-8, line 56, 35, and 87.)

Then the precondition for the put phase ending is that there exists a put-quorum $P \in put\text{-}quorums(c)$ such that $P \subseteq op.acc_i$ (see Figure 6-8, line 60, 39, and 88.

An object *obj* is added to $op.acc$ only when a $\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_{obj,p}$ event occurs (see Figure 6-5, lines 57–67). The Virtual Object Layer guarantees that each respond event is caused by a unique preceding invoke event, that is: $\mathsf{invoke}(\mathsf{put}, t, v, cid)_{obj,p}$. Since the invocation takes place on port *p*, this means that it must occur after the beginning of the put phase. Therefore, the tag, *t*, is in fact equal to $tag(\pi)$, the tag at the beginning of the put phase, by Lemma 6.3.1 and the definition of $tag(\pi)$. The Virtual Object Layer guarantees that at some point between the invocation and the response, the put transition was executed on the object's state, thus ensuring that the tag of the object is no smaller than *t*.

We conclude, then, by Lemma 6.3.2, that for each object, $obj \in P$, $tag(\pi) \leq obj.tag$ after operation π completes.

We now consider the case where $put\text{-}rip(\pi) = \mathsf{true}$. Assume, then, that π is a two-phase read operation or a write operation. In this case, the precondition for the put phase ending is that for every configuration c', there exists a put-quorum $P \in put\text{-}quorums(c')$ such that $P \subseteq op.acc_i$ (see Figure 6-7, line 37, Figure 6-8, line 58). Fix $c' = c$.

By the same argument as before, we can conclude that for every object, $obj \in P$, $tag(\pi) \leq obj.tag$ when operation π completes. □

When *put-rip* is true there is a stronger version of this lemma for read and write operations: there exists at least one put-quorum for each configuration where every object in the put-quorum has a tag no smaller than the tag of the operation.

Lemma 6.3.4. *Let π be a two-phase read operation or a write operation, and assume it occurs at node i. If put-rip(π) = true, then for every $c \in M$, there exists a put-quorum, $P \in$ put-quorums(c), such that for every object, obj, in quorum P, $tag(\pi) \leq obj.tag$ anytime after π completes.*

Proof. This lemma follows from the termination condition of the put phase of an operation. Assume that when the put phase of π begins (i.e., immediately after the write, read-2, or recon-2 event), $p = \langle$*current-port-number*$, \mathsf{put}, i\rangle$, the port that is used throughout the phase. Fix any arbitrary $c \in M$.

The precondition for the put phase ending is that there exists a put-quorum $P \in$ *put-quorums*(c) such that $P \subseteq op.acc_i$ (see Figure 6-7, line 37, Figure 6-8, line 58).

An object, *obj* is added to *op.acc* only when a $\mathsf{respond}(\mathsf{put\text{-}ack}, \ldots)_{obj,p}$ event occurs (see Figure 6-5, lines 57–67). The Virtual Object Layer guarantees that each respond event is caused by a unique preceding invoke event: $\mathsf{invoke}(\mathsf{put}, t, v, c)_{obj,p}$. Since this invocation occurs on port p, this means that it must occur after the beginning of the put phase. Therefore, the tag, t, is in fact equal to $tag(\pi)$, the tag at the beginning of the put phase, by Lemma 6.3.1 and the definition of $tag(\pi)$. The Virtual Object Layer guarantees that at some point between the invocation and the response, the put transition was executed on the object's state, thus ensuring that the tag of the object is no smaller than t.

We conclude, then, by Lemma 6.3.2, that for each object, $obj \in P$, $tag(\pi) \leq obj.tag$ after operation π completes. Since for every $c \in M$ there exists such a put-quorum, P, the lemma holds. □

We next show that a get phase effectively retrieves information on the tags from a quorum of a certain configuration.

Lemma 6.3.5. *Let π be a two-phase read operation that occurs at node i. Then there exists a get-quorum, G, in get-conf(π) such that for every object, obj in G, obj.tag when π begins is $\leq tag(\pi)$.*

Proof. This lemma is similar to Lemma 6.3.3, and follows from the termination condition of the get phase of an operation.

Assume that when the get phase begins (i.e., immediately after the read event), $p = \langle$*current-port-number*$, \mathsf{get}, i\rangle$, the port that is used throughout the phase. Also, assume that configuration identifier $cid = $ *get-conf-id*$(\pi) = \langle *, *, c\rangle$.

We divide the proof into two subcases: the case where *get-rip*$(\pi) =$ false, and where *get-rip*$(\pi) =$ true.

First, consider the case where *get-rip*$(\pi) =$ false. Recall that *cid*, the *get-conf-id*(π), is equal to the configuration identifier $conf\text{-}id_i$ when the get phase of the operation completes. (Notice that our use of c is consistent with the notation used in Figure 6-7, line 20.)

Then the precondition for the get phase ending is that there exists a get-quorum $G \in \textit{put-quorums}(c)$ such that $G \subseteq op.acc_i$ (see Figure 6-7, line 24).

An object, *obj* is added to $op.acc$ only when a $\mathsf{respond}(\mathsf{get\text{-}ack}, t, v, \ldots)_{obj,p}$ event occurs (see Figure 6-5, lines 40–55). The Virtual Object Layer guarantees that each respond event is caused by a unique preceding invoke event: $\mathsf{invoke}(\mathsf{get}, \ldots)_{obj,p}$.

Since the invocation takes place on port p, this means that it must occur after the beginning of the get phase. The Virtual Object Layer guarantees that the get transition occurs sometime after the invocation and prior to the response. Therefore, the tag t in the response is greater than or equal to $obj.tag$ when the invocation occurs. We therefore conclude, by observing Figure 6-5, lines 44–45, that $obj.tag \leq tag(\pi)$ when the phase begins.

We now consider the case where $\textit{get-rip}(\pi) = \mathsf{true}$. In this case, the precondition for the get phase ending is that for every configuration c', and in particular for the case where $c' = c$, there exists a get-quorum $G \in \textit{get-quorums}(c')$ such that $G \subseteq op.acc_i$ (see Figure 6-7, line 22).

By the same argument as before, we can conclude that for every object, $obj \in G$, $obj.tag \leq tag(\pi)$ when the phase begins. □

Again, in the case where $\textit{get-rip}(\pi)$ is true, we can show a stronger property: the get phase retrieves tag information from at least one get-quorum of each configuration.

Lemma 6.3.6. *Let π be a two-phase read operation that occurs at node i. If $\textit{recon-ip}_i =$ true at the end of the get phase, then for every configuration $c \in M$, there exists a get-quorum, $G \in \textit{get-quorums}(c)$ such that for every object, obj in G, $obj.tag$ when π begins is $\leq tag(\pi)$.*

Proof. This lemma is similar to Lemma 6.3.4, and follows from the termination condition of the get phase of an operation.

Assume that when the get phase of operation π begins (i.e., immediately after the read event occurs), the port $p = \langle \textit{current-port-number}, \mathsf{get}, i \rangle$, the port that is used throughout the phase. Fix any arbitrary $c \in M$.

The precondition for the get phase ending is that there exists a get-quorum $G \in \textit{get-quorums}(c)$ such that $G \subseteq op.acc_i$ (see Figure 6-7, line 22).

An object, *obj* is added to $op.acc$ only when a $\mathsf{respond}(\mathsf{get\text{-}ack}, t, v, \ldots)_{obj,p}$ event occurs (see Figure 6-5, lines 40–55). The focal point object model guarantees that each respond event is caused by a unique preceding invoke event: $\mathsf{invoke}(\mathsf{get}, \ldots)_{obj,p}$.

Since the invocation takes place on port p, this means that it must occur after the beginning of the get phase. The Virtual Object Layer guarantees that the get transition occurs sometime after the invocation and prior to the response. Therefore, the tag t in the response is greater than or equal to $obj.tag$ when the invocation occurs. We therefore conclude, by observing Figure 6-5, lines 44–45, that $obj.tag \leq tag(\pi)$ when the phase begins. Since for every $c \in M$ there exists such a get-quorum, G, the lemma holds. □

6.3.4 Outline of the Operation Manager Proof

Our goal is to show that if we have two operations, π_1 and π_2, and π_1 completes before π_2 begins, then $\pi_2 \not\prec \pi_1$. We break this proof into a number of cases:

1. Operation π_2 is a write operation (Lemma 6.3.7).
2. Operation π_2 is a read operation and operation π_1 is either a two-phase read operation or a write operation.
   (a) $\textit{put-ip}(\pi_1) \vee \textit{get-ip}(\pi_2) = \mathsf{true}$.
   Either the put phase of π_1 or the get phase of π_2 detects a reconfiguration in progress (Lemma 6.3.8).
   (b) $\textit{put-ip}(\pi_1) \vee \textit{get-ip}(\pi_2) = \mathsf{false}$.
   Neither the put phase of π_1 nor the get phase of π_2 detect a reconfiguration in progress.
      i. $\textit{put-conf-id}(\pi_2) = \textit{get-conf-id}(\pi_2)$.
      The put configuration identifer of π_2 is equal to the get configuration identifer of π_2 (Lemma 6.3.9).
      ii. $\textit{put-conf-id}(\pi_1) > \textit{get-conf-id}(\pi_2)$.
      The put configuration identifier of π_1 is strictly larger than the get configuration identifier of π_2 (Lemma 6.3.11).
      iii. $\textit{put-conf-id}(\pi_1) < \textit{get-conf-id}(\pi_2)$.
      The put configuration identifier of π_1 is strictly smaller than the get configuration identifier of π_2 (Lemma 6.3.12).
3. Operation π_2 is a read operation and operation π_1 is a one-phase read operation (Lemma 6.3.14).

6.3.5 Proving the Operation Manager Correct

We now proceed to examine the various cases, as outlined above.

Case 1: Write Operation. We first consider the case where π_2 is a write operation:

Lemma 6.3.7. *If π_1 is a read or write operation, and π_2 is a write operation, and π_1 completes before π_2 begins, then $\pi_1 \prec \pi_2$.*

Proof. Assume that operation π_2 occurs at node i. The result follows immediately by the choice of $\textit{tag}(\pi_2)$. The tag $\textit{op.tag}_i$ is chosen during the $\mathsf{write}(v)_i$ action (see Figure 6-8, line 52). It is chosen using the real-time clock (along with a process identifier to break ties). The tag of π_1 must have been chosen at the beginning of a prior write operation, or must be the initial value. If the tag of π_1 is the initial value, then the tag of π_2 is necessarily larger. Assume, then, that the tag of π_1 originates with a prior write operation.

This prior write operation must take some time strictly greater than zero to complete, since the write operation requires performing at least one invocation on a virtual

object and receiving a response from that invocation. The virtual object model guarantees that each operation consisting of an invocation and a response on a virtual object takes some time to complete: the invocation and the response do not occur at the same time. Therefore the write operation must take some time strictly greater than zero to complete. Since π_2 begins after π_1 ends, it begins at some time strictly greater than zero after the prior write operation begins.

This ensures that $tag(\pi_1) < tag(\pi_2)$, which immediately implies that $\pi_1 \prec \pi_2$, as desired. □

For the rest of the proof we assume that π_2 is a read operation.

Case 2: Two-Phase Read and Write Operations. We now consider the case where π_1 is either a two-phase read operation or a write operation, and π_2 is a read operation. We postpone until later the case where π_1 is a one-phase read operation.

There are two subcases to consider, depending on whether at least one of the flags *put-rip*(π_1) or *get-rip*(π_2) is true (Case 2(a)), or both flags are false (Case 2(b)).

We first consider the case where at least one of the put phase of π_1 or the get phase of π_2 detects a reconfiguration in progress (Case 2(a)). That is, if i initiates operation π_1 and j initiates π_2, then we assume that at least one of the following two conditions holds:

- At the end of the put phase of π_1, $op.\textit{recon-ip}_i = \mathsf{true}$.
- At the end of the get phase of π_2, $op.\textit{recon-ip}_j = \mathsf{true}$.

Lemma 6.3.8 (Case 2(a))**.** *Assume operation π_1 is a two-phase read or write operation at node i. Assume that π_2 is a read operation initiated at node j, and that π_1 completes before π_2 begins. Assume that at least one of the following is true:*

- *put-rip*$(\pi_1) = \mathsf{true}$*, or*
- *get-rip*$(\pi_2) = \mathsf{true}$*.*

Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result $\pi_2 \not\prec \pi_1$.

Proof. In this case, at least one of the two nodes detects a reconfiguration in progress: node i during the put phase and/or node j during the get phase. We divide this case into two subcases, depending on whether it is node i or node j that detects the reconfiguration.

Subcase 1: First, assume that *put-rip*$(\pi_1) = \mathsf{true}$. This implies that node i detects the reconfiguration during the put phase of π_1.

Choose $c' =$ *get-conf*(π_2). Lemma 6.3.4 guarantees that there exists a put-quorum, $P \in$ *put-quorums*(c'), such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ when π_1 completes (since it guarantees this for every $c' \in M$).

Lemma 6.3.5 guarantees that there exists a get-quorum, G, in *get-quorums*(c'), the get configuration of π_2, such that for every object, $obj \in G$, $obj.tag \leq tag(\pi_2)$ when π_2 begins.

Then there must exist an object, $obj \in G \cap P$, since both are quorums of the same configuration c' and one is a get-quorum and the other is a put-quorum.

We already know that $tag(\pi_1) \leq obj.tag$ when π_1 completes. And $obj.tag$ when π_2 begins is $\leq tag(\pi_2)$. Since π_1 completes before π_2 begins, we conclude that $tag(\pi_1) \leq tag(\pi_2)$.

Subcase 2: Next, assume that $get\text{-}rip(\pi_2) = \mathsf{true}$. This implies that node j detects the reconfiguration during the get phase of π_2.

Choose $c' = put\text{-}conf(\pi_1)$. Lemma 6.3.6 guarantees that for every $c' \in M$, there exists a quorum, $G \in get\text{-}quorums(c')$, such that for every object, $obj \in G$, $obj.tag \leq tag(\pi_2)$ when π_2 begins (since it guarantees this for every $c' \in M$).

Lemma 6.3.3 guarantees that there exists a put-quorum, P, in $put\text{-}quorums(c')$, the put configuration of π_1, such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ when π_1 completes.

Then there must exist an object, $obj \in G \cap P$, since both are quorums of the same configuration c' and one is a get-quorum and the other is a put-quorum.

We already know that $tag(\pi_1) \leq obj.tag$ when π_1 completes. And $obj.tag \leq tag(\pi_2)$ when π_2 begins. Since π_1 completes before π_2 begins, we conclude that $tag(\pi_1) \leq tag(\pi_2)$. □

In the next case (Case 2(b)), we assume that neither the put phase of operation π_1 nor the get phase of operation π_2 detects the reconfiguration in progress. Thus for the next set of lemmas, we assume that $put\text{-}rip(\pi_1)$ and $get\text{-}rip(\pi_2)$ are both false. This case has three subcases, depending on the relationship of the put configuration identifier of π_2 and the get configuration identifier of π_1. First, we assume that these configurations identifiers are the same.

Lemma 6.3.9 (Case 2(b).i)**.** *Assume that operation π_1 is a two-phase read operation or a write operation at node i, and that π_2 is a read operation at node j. Assume that π_1 completes before π_2 begins.*

Also, assume that the flags $put\text{-}rip(\pi_1)$ and $get\text{-}rip(\pi_2)$ are both false *and that the identifier $put\text{-}conf\text{-}id(\pi_1) = get\text{-}conf\text{-}id(\pi_2)$. Then we conclude that $tag(\pi_1) \leq tag(\pi_2)$, and as a result, $\pi_2 \not\prec \pi_1$.*

Proof. Let $cid = put\text{-}conf\text{-}id(\pi_2) = get\text{-}conf\text{-}id(\pi_1)$. Assume that $cid = \langle *, *, c \rangle$.

Lemma 6.3.3 guarantees that there exists a put-quorum, P such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ when π_1 completes.

Lemma 6.3.6 guarantees that there exists a get-quorum, $G \in get\text{-}quorums(c)$ such that for every object, $obj \in G$, the tag $obj.tag \leq tag(\pi_2)$ when π_2 begins.

Since P is a put-quorum and G is a get-quorum of the configuration identified by c, there must exist some object, $obj \in P \cap G$.

We already know that $tag(\pi_1) \leq obj.tag$ when π_1 completes. And $obj.tag \leq tag(\pi_2)$ when π_2 begins. Since π_1 completes before π_2 begins, we conclude that $tag(\pi_1) \leq tag(\pi_2)$. □

We now consider the case (Case 2(b).ii) where the put configuration identifier of π_1 is larger than the get configuration identifier of π_2. That is, we consider the case where *put-conf-id*$(\pi_1) >$ *get-conf-id*(π_2). It turns out that this case cannot occur.

We first need to show that when the *recon-ip* flag at node i is set to false, this correctly indicates that the configuration identified by *conf-id* is fully installed, meaning that the reconfiguration that created *conf-id* has completed.

Since we assume in this case (Case 2(b).ii) that the flag *get-rip*(π_2) is false, this lemma shows that the configuration identified by *get-conf-id*(π_2) is fully installed prior to the start of π_2.

Lemma 6.3.10. *Let α' be a prefix of α, and let c be some configuration that is not the initial configuration:*

$$c \neq \langle 0, i_0, c_0 \rangle \ .$$

Assume that at the end of α', conf-id$_i$ = cid $= \langle *, *, c \rangle$ and recon-ip$_i$ = false*. Then for some node j, a* recon-ack$(cid)_j$ *event occurs in α'.*

Proof. Assume, without loss of generality, that α' is the shortest prefix of α such that for any node k, *conf-id*$_k$ = *cid* and *recon-ip*$_k$ = false at the end of α'.

There are only two ways in which i can have configuration identifier *conf-id*$_i$ = *cid* and flag *recon-ip*$_i$ set to false: either i performs a recon-ack$(cid)_i$ action (see Figure 6-8, line 92), or i receives a put-ack or get-ack response from an object specifying configuration c and *new-rip* = false (see Figure 6-5, lines 50 and 64). (The recon$(c)_i$ event does result in *conf-id*$_i$ = *cid*, however *recon-ip*$_i$ is set to true.)

Assume, however, that i receives a put-ack or get-ack from some object, *obj*, specifying configuration *new-cid* and flag *new-rip* set to false. Then we know that an invocation event, invoke(recon-done, *new-cid*)$_{obj,\langle *,j' \rangle}$, must occur prior to the put-ack or get-ack from *obj*, as this is the only event that can set *obj.recon-ip* to false.

But we assumed that i was the first node to be in this state (i.e., α' is the shortest prefix ending with some node i in this state), so this recon-done invocation cannot occur. Therefore i must perform a *recon-ack*$(cid)_i$. The node i, then, satisfies the required properties of node j. □

We can now show that the get configuration identifier of π_2 is always greater than or equal to the put configuration identifier of π_1. Therefore, the second case (Case 2(b).ii) can never occur.

Lemma 6.3.11 (Case 2(b).ii)**.** *Assume operation π_1 occurs in α at node i before operation π_2 begins at node j. Assume that π_1 is a two phase read or write operation, and π_2 is a read operation.*

Assume that put-rip(π_1) and get-rip(π_2) are both false*. Then put-conf-id$(\pi_1) \leq$ get-conf-id(π_2).*

Proof. If *put-conf-id*$(\pi_1) = \langle 0, i_0, c_0 \rangle$ (the smallest possible value for a configuration identifier), then clearly this result is true. Assume, therefore, that *put-conf-id*$(\pi_1) > \langle 0, i_0, c_0 \rangle$.

It is clear that *recon-ip*$_i$ is false at the end of the put phase of π_1, since *op.recon-ip*$_i$ = false: whenever *recon-ip*$_i$ is set to true, so is *op.recon-ip*$_i$, and *op.recon-ip* is not reset to false until the phase is completed.

Lemma 6.3.10 then implies that for some node, k, a reconfiguration acknowledgment, recon-ack(*put-conf-id*(π_1))$_k$, occurs prior to the end of the second phase of π_1. In particular, the recon-ack occurs prior to the beginning of π_2.

In order for the recon to complete, a recon-2 must occur. This event completes the get phase of reconfiguration. The precondition of recon-2 requires that for every configuration $c' \in M$, there exists a quorum $P \in$ *put-quorums*(c') such that $P \subseteq$ *op.acc*. This implies that each object in quorum P responds to a get invocation in the first phase of the recon operation. (Notice that during a reconfiguration, there are get invocations made on objects in put-quorums. This is the one exception to the general rule that get operations are invoked on objects in get quorums and put operations are invoked on objects in put quorums.)

Choose $c' =$ *get-conf*(π_2), and let the put-quorum $P \in$ *put-quorums*(c') be the put-quorum (described above) contacted by node k prior to the recon-2 event, and therefore prior to the start of operation π_2.

When the get phase of π_2 completes, there exists some put-quorum of objects, $G \in$ *put-quorums*(*get-conf*(π_2)), such that every object, *obj* $\in G$ has responded to a get invocation during the first phase of π_2.

There must be some object *obj* in $G \cap P$, as both G and P are quorums of the same configuration, *get-conf*(π_2), and one is a get-quorum and the other is a put-quorum.

Recall that the reconfiguration is installing the configuration *put-conf-id*(π_1). As a result of the invocation of object *obj* during the get phase of the reconfiguration, it is clear that at the end of the get phase, *put-conf-id*$(\pi_1) \leq$ *obj.config-id*.

As a result of the response of object *obj* during the get phase of π_2, it is clear that *obj.config-id* $\leq$ *get-conf-id*(π_2) at the beginning of the get phase.

We thus conclude: *put-conf-id*$(\pi_1) \leq$ *get-conf-id*(π_2). □

The next case to consider is when the put configuration identifier of π_1 is strictly smaller than the get configuration identifier of π_2. This is the most complicated part of the proof, and relies on showing that an intervening reconfiguration operation – the one that creates configuration *get-conf-id*(π_2) – relays information from π_1 to π_2.

Lemma 6.3.12 (Case 2(b).iii)**.** *Assume operation π_1 is a two-phase read or write operation at node i. Assume that π_2 is a read operation initiated at node j, and that π_1 completes before π_2 begins.*

Also, assume that put-conf-id$(\pi_1) <$ get-conf-id(π_2). Finally, assume that put-rip(π_1) and get-rip(π_2) both equal false*. Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result $\pi_2 \not\prec \pi_1$.*

Proof. Some reconfiguration must occur in order to create configuration *get-conf-id*(π_2). We first identify the reconfiguration, ρ, that creates the new configuration. We then show that $tag(\rho) \leq tag(\pi_2)$. Finally, we show that $tag(\pi_1) \leq tag(\rho)$, concluding the proof.

Notice that *get-conf-id*$(\pi_2) \neq \langle 0, i_0, c_0 \rangle$, since it is strictly larger than *put-conf-id*(π_1), and $\langle 0, i_0, c_0 \rangle$ is the smallest possible value for configuration identifier *put-conf-id*(π_1).

Since $op.$*recon-ip*$_j$ = false at the *end* of the get phase of π_2, this means that *recon-ip*$_j$ = false at the *beginning* of the get phase of π_2: this is because no action resets $op.$*recon-ip*$_j$ to true during an operation.

Consider the prefix α' of α whose last event is the read event that begins π_2. Then by Lemma 6.3.10, there exists some node k that performs a recon-ack(*get-conf*$(\pi_2))_k$ in α', that is, prior to the read$_j$ of π_2. Let ρ be the recon operation concluding with this recon-ack event.

We now show that $tag(\rho) \leq tag(\pi_2)$. Lemma 6.3.3 guarantees that there exists some put-quorum, P, in the put configuration of ρ such that for each object, $obj \in P$, $tag(\rho) \leq obj.tag$ at the end of the reconfiguration. Note that P is in *put-quorums*(c), where c is a part of the put and get configuration identifiers; that is, $\langle *, *, c \rangle =$ *put-conf-id*(ρ) and $\langle *, *, c \rangle =$ *get-conf-id*(π_2).

Lemma 6.3.5 guarantees that there exists some get-quorum, $G \in$ *get-quorums*(c) such that for every object, $obj \in G$, $obj.tag \leq tag(\pi_2)$ when π_2 begins.

Since G is a get-quorum and P is a put-quorum of the configuration identified by *get-conf-id*(π_2), there exists an object, $obj_1 \in G \cap P$.

We have already shown that $tag(\rho) \leq obj_1.tag$ when ρ completes. And we have already shown that $obj_1.tag$ when π_2 begins is $\leq tag(\pi_2)$. Since ρ completes before π_2 begins, we conclude from Lemma 6.3.2 that $tag(\rho) \leq tag(\pi_2)$.

We next show an ordering of the tags, i.e., that $tag(\pi_1) \leq tag(\rho)$. Consider the following reconfiguration event recon-2(*get-conf-id*$(\pi_2))_k$ that occurs as part of reconfiguration ρ, ending the get phase of the reconfiguration. The precondition for the recon-2 action requires that for every configuration $c' \in M$, there exists a get-quorums $G \in$ *get-quorums*(c') such that $G \subseteq op.acc_k$ when the event occurs. This implies that for every object, $obj \in G$, an invoke(get, $\ldots)_{obj,p}$ event and a respond(get-ack, $\ldots)_{obj,p}$ event occur during the get phase, where p is the port number during the get phase of the reconfiguration. As part of this get operation, a perform event occurs at the automaton for obj.

Choose $c' =$ *put-conf-id*(π_1), and let quorum G be the get-quorum determined by the end of the get phase of the reconfiguration. Lemma 6.3.3 guarantees that there exists some put-quorum, $P \in$ *put-quorums*(c') such that for every object, $obj \in P$, $tag(\pi_1) \leq obj.tag$ at the end of π_1.

Since G is a get-quorum of the configuration identified by c' and P is a put-quorum of the configuration identified by c', there exists some object, $obj_2 \in G \cap P$. We know that $tag(\pi_1) \leq obj_2.tag$ when π_1 completes, since an invocation during the put phase ensure that $obj_2.tag$ is at least $tag(\pi_1)$. And we know that $obj_2.tag \leq tag(\rho)$ when ρ begins.

At this point, however, we do not know which event came first: the invocation during the put phase of π_1 or the response during the get phase of ρ.

Since obj_2 is an atomic object, it must process these two invocations – doing perform steps in the canonical automaton – in one order or the other. Assume that obj_2 processes the invocation by ρ first, that is, the perform step in response to ρ precedes the perform step in response to π_1. In this case, the response to π_1 includes

a configuration identifier no smaller than *get-conf-id*(π_2), the configuration being installed by ρ. As a result: *put-conf-id*(π_1) $\geq$ *get-conf-id*(π_2). This contradicts our assumption that the put configuration identifier of π_1 is less than the get configuration identifier for π_2.

Therefore, we can conclude that the invocation of obj_2 for π_1 precedes the response of obj_2 for π_2. It follows, then, that $tag(\pi_1) \leq tag(\rho)$. Combining the two inequalities, we conclude that $tag(\pi_1) \leq tag(\pi_2)$, which implies that $\pi_2 \not\prec \pi_1$. □

Case 3: One-Phase Read Operations. We now address the case of single-phase read operations. We assume that π_1 is a one-phase read operation, that is, it does not have a put phase.

Notice that in Lemma 6.3.12, we depended significantly on π_1 propagating its tag to a put-quorum of objects. Since a one-phase read operation has no put phase, we cannot use this.

Instead, we rely on the fact that a one-phase read operation, π occurs only when the tag of operation π is *confirmed*, indicating that another two-phase operation, π' has already propagated the tag of π to a put-quorum.

We first need a lemma showing that if a tag, t, is confirmed, then there exists a two-phase operation that propagated tag t to a put-quorum.

Lemma 6.3.13. *Let α' be a prefix of α, and assume that at some node i, at the end of α', the tag $t \in confirmed_i$. Then there exists an operation, π, in α' such that $tag(\pi) = t$ and π is either a two phase read-operation or a write operation.*

Proof. Without loss of generality, assume that execution α' is the shortest prefix of α such that at the end of α', for some node i, $t \in confirmed_i$.

There are two ways in which a tag can be added to the confirmed set of i: either a response from some object indicates that a tag is confirmed (see Figure 6-5, line 54), or i itself completes a two-phase operation and adds t to the confirmed set (Figure 6-8, lines 43 and 64).

In the first case, this implies that there exists some object, *obj*, that has $t \in obj.confirmed\text{-}set$ at some point in α'. However, this would imply that some other node k had performed a confirm invocation on *obj* (Figure 6-3, line 27) in α', as this is the only way in which a tag can be added to an object's confirmed set.

This, then, implies that $t \in confirmed_k$ in α', when the confirm invocation occurs. This violates the assumption that α' is the shortest prefix to end with some node, i, containing t in $confirmed_i$.

Therefore, node i must perform a read-ack$_i$ or write-ack$_i$ in α' that adds t to $confirmed_i$ (Figure 6-8, lines 43 and 64). The value $op.tag_i$ must be equal to t, because t is added to $confirmed_i$. Further, if the operation is a read operation, then a precondition of the read-ack$_i$ is that $op.phase =$ put, implying that it is a two-phase read operation. The node i, then, satisfies all the required properties of node j. □

Now we can show that with one-phase reads, the partial ordering induced by the tags is consistent with the real ordering of the operations:

Lemma 6.3.14 (Case 3)**.** *Assume operation π_1 is a one-phase read operation, and occurs at node i. Assume that π_2 is a read operation initiated at node j, and that π_1 completes before π_2 begins. Then $tag(\pi_1) \leq tag(\pi_2)$, and as a result $\pi_2 \not\prec \pi_1$.*

Proof. Since π_1 is a one-phase read operation, the associated tag is confirmed ($tag(\pi_1) \in confirmed_i$) when the read-ack$_i$ event occurs (Figure 6-7, line 14). Recall that the tag, $tag(\pi_1)$, is the value of tag_i when the read-ack$_i$ event occurs.

By Lemma 6.3.13, a two-phase operation π' must complete prior to the read-ack$_i$ event of π_1 and $tag(\pi_1) = tag(\pi')$, the tag_i when the read-ack$_i$ event occurs.

Since π' completes before the end of π_1, it also completes before π_2 begins. Therefore, by Lemma 6.3.12, $tag(\pi') \leq tag(\pi_2)$; as a result, $tag(\pi_1) \leq tag(\pi_2)$. □

Main Result. Combining the lemmas from all the various cases (that is, Lemma 6.3.7, Lemma 6.3.8, Lemma 6.3.9, Lemma 6.3.11, Lemma 6.3.12, and Lemma 6.3.14), we conclude:

Theorem 6.3.15. *If π_1 and π_2 are two operations, and π_1 completes before π_2 begins, then $\pi_2 \not\prec \pi_1$.*

We now claim that the Operation Manager, composed with the virtual objects, the ReconClients, and a well-formed environment, implements a read/write atomic object.

Theorem 6.3.16. *Let U be a well-formed environment. Let S be the composition of the Operation Manager, the virtual objects, and the ReconClients, where all input and output actions are hidden except for* read, read-ack, write, *and* write-ack*. Let A be the canonical atomic read/write object, an object of the variable type presented in Figure A-1. Then $traces(S \times U) \subseteq traces(A \times U)$).*

Proof. First, notice that S has the appropriate input and output actions. Next, we go through the three conditions required by Theorem A.3.3:

1. Follows from the uniqueness of the tags: each is chosen by examining the local clock, and using process identifiers to break ties.

2. Follows from Theorem 6.3.15.

3. Follows from the way in which the partial order is defined, as a read operation is ordered directly after the write operation whose value it returns.

□

Finally, we observe that if we execute the Operation Manager and virtual objects in the context of a Virtual Object Layer implemented as per the Virtual Object Emulator in Chapter 4, then the result is a correct implementation of an atomic object:

Theorem 6.3.17. *Let U be a well-formed environment. Let S be the composition of the Operation Manager, the ReconClients, and the Virtual Object Emulator (described in Chapter 4, instantiated with virtual objects of the put/get variable type, where all input and output actions are hidden except for* read, read-ack, write, *and* write-ack. *Let A be the canonical atomic read/write object, an object of the variable type presented in Figure A-1. Then: $traces(S \times U) \subseteq traces(A \times U)$).*

Proof. This follows immediately from Theorem 6.3.16 and Theorem 4.2.6. □

6.4 Performance Discussion

The performance of the GeoQuorums Operation Manager depends on the performance of the underlying Virtual Object Layer.

We first examine the performance of read and write operations, as executed by the Operation Manager. Assume that the Virtual Object Layer guarantees that all invocations result in a response within time T, if a mobile node at location ℓ invokes an operation on a correct virtual object h and remains within a small distance of ℓ for sufficiently long.

Then if no more than f virtual objects fail, each read or write operations takes at most time $2T$: each operation requires at most two phases, and each phase can be completed in time T, as all the objects can be invoked concurrently. A write operation, however requires on a single phase. Similarly, many read operations only require a single phase. These operations require at most time T. Similarly, a reconfiguration operation takes at most time $2T$.

If the Virtual Object Layer is implemented by the Virtual Object Emulator described in Chapter 4 in an underlying physical model described in Chapter 2, and if the underlying model delivers GeoCast messages within time d_{geo} and fpcast messages within time d_{fp}, then Theorem 4.3.1 indicates that $T \leq 2d_{\mathsf{geo}} + d_{\mathsf{fp}}$.

We then conclude that if a mobile node i initiates a read or write operation and remains near to the location at which it initiated the operation for sufficiently long, and if no more than f virtual objects fail, then the operation completes within time $8(d_{\mathsf{geo}} + d_{\mathsf{fp}})$. A write operation, or a one-phase read operation, completes within time $4(d_{\mathsf{geo}} + d_{\mathsf{fp}})$.

The algorithm as specified also allows the implementation to trade-off message complexity and latency. In each phase of a read or write operation, the node initiating the operation must perform invocations on a quorum of virtual objects; each invocation is going to cause message traffic in the network. It can achieve this goal by performing invocations on all virtual objects concurrently, thereby ensuring the fastest result, at the expense of a high message complexity. Alternatively, the node can invoke only the virtual objects in a single quorum. If some of these virtual objects have failed, and they do not all respond, the node can perform invocations on another quorum, and continue until it receives a response from every object in some quorum. This leads to lower message complexity, but may take longer.

6.5 Discussion

In this chapter we have presented an algorithm, the GeoQuorums Operation Manager, that implements a read/write atomic shared memory using the Virtual Object Layer.

It remains an interesting question to consider how configurations should be chosen, and when reconfiguration should occur. For example, how many configurations should the algorithm use? The more configurations that are made available to the algorithm, the better the performance of read and write operations, if the correct configuration is installed. On the other hand, more configurations means slower reconfiguration and slower operations during reconfiguration.

Moreover, if there are many possible configuration, the choice of a configuration becomes more difficult. Since there are only a finite number of configurations to choose from, it should be possible for the mobile nodes to determine which configuration is optimal for a given set of read and write operations. Using the techniques of competitive analysis it may be possible to determine the optimal configuration for a sequence of read and write operations, even without knowing the sequence in advance.

Figure 6-7: Operation Manager Client Transitions for node i

Input read()$_i$
Effect:
    current-port-number $\leftarrow$ *current-port-number* $+1$
    $op \leftarrow \langle$read, get, $\perp$, $\perp$, *recon-ip*, $\langle 0, i_0, c_0 \rangle$, $\emptyset\rangle$

Output read-ack(v)$_i$
Precondition:
    conf-id $= \langle$*time-stamp*, *pid*, $c\rangle$
    if *op.recon-ip* **then**
        $\forall c' \in M, \exists\ G \in$ *get-quorums*$(c') : G \subseteq$ *op.acc*
    else
        $\exists\ G \in$ *get-quorums*$(c) : G \subseteq$ *op.acc*
    $\langle$*op.phase*, *op.type*, *op.value*$\rangle = \langle$get, read, $v\rangle$
    op.tag $\in$ *confirmed*
Effect:
    op.phase $\leftarrow$ idle

Internal read-2()$_i$
Precondition:
    conf-id $= \langle$*time-stamp*, *pid*, $c\rangle$
    if *op.recon-ip* **then**
        $\forall c' \in M, \exists\ G \in$ *get-quorums*$(c') : G \subseteq$ *op.acc*
    else
        $\exists\ G \in$ *get-quorums*$(c) : G \subseteq$ *op.acc*
    $\langle$*op.phase*, *op.type*$\rangle = \langle$get, read$\rangle$
    op.tag $\notin$ *confirmed*
Effect:
    current-port-number $\leftarrow$ *current-port-number* $+1$
    op.phase $\leftarrow$ put
    op.recon-ip $\leftarrow$ *recon-ip*
    op.acc $\leftarrow \emptyset$

Output read-ack(v)$_i$
Precondition:
    conf-id $= \langle$*time-stamp*, *pid*, $c\rangle$
    if *op.recon-ip* **then**
        $\forall c' \in M, \exists\ P \in$ *put-quorums*$(c') : P \subseteq$ *op.acc*
    else
        $\exists\ P \in$ *put-quorums*$(c) : P \subseteq$ *op.acc*
    $\langle$*op.phase*, *op.type*, *op.value*$\rangle = \langle$put, read, $v\rangle$
Effect:
    op.phase $\leftarrow$ idle
    confirmed $\leftarrow$ *confirmed* $\cup \{$*op.tag*$\}$

Input geo-update(t, l)$_i$
Effect:
    clock $\leftarrow t$

Figure 6-8: Operation Manager Client Transitions for node i

```
49  Input write(val)_i
50  Effect:
51      current-port-number ← current-port-number +1
52      op ← ⟨write, put, ⟨clock, i⟩, val, recon-ip, ⟨0, i_0, c_0⟩, ∅⟩
53
54  Output write-ack()_i
55  Precondition:
56      conf-id = ⟨time-stamp, pid, c⟩
57      if op.recon-ip then
58          ∀c' ∈ M, ∃ P ∈ put-quorums(c') : P ⊆ op.acc
59      else
60          ∃ P ∈ put-quorums(c) : P ⊆ op.acc
61      ⟨op.phase, op.type⟩ = ⟨put, write⟩
62  Effect:
63      op.phase ← idle
64      confirmed ← confirmed ∪ {op.tag}
65
66  Input recon(conf-name)_i
67  Effect:
68      conf-id ← ⟨clock, i, conf-name⟩
69      recon-ip ← true
70      current-port-number ← current-port-number +1
71      op ← ⟨recon, get, ⊥, ⊥, true, conf-id, ∅⟩
72
73  Internal recon-2(cid)_i
74  Precondition
75      ∀c' ∈ M, ∃ G ∈ get-quorums(c') : G ⊆ op.acc
76      ∀c' ∈ M, ∃ P ∈ put-quorums(c') : P ⊆ op.acc
77      ⟨op.phase, op.type⟩ = ⟨get, recon⟩
78      cid = op.recon-conf-id
79  Effect:
80      current-port-number ← current-port-number +1
81      op.phase ← put
82      op.acc ← ∅
83
84  Output recon-ack(c)_i
85  Precondition:
86      cid = op.recon-conf-id
87      cid = ⟨time-stamp, pid, c⟩
88      ∃ P ∈ put-quorums(c) : P ⊆ op.acc
89      ⟨op.phase, op.type⟩ = ⟨put, recon⟩
90  Effect:
91      if (conf-id = op.recon-conf-id) then
92          recon-ip ← false
93      op.phase ← idle
```

Chapter 7

Virtual Infrastructure and Local Communication

In this chapter we consider wireless networks that only have the capacity for local communication. That is, each mobile node can communicate only with its nearby neighbors. By contrast, the GeoCast service introduced in Chapter 2 allowed for mobile nodes to send messages across the network to other devices that may be far away. The fpcast service, though in many ways "local," also guarantees more than your typical wireless MAC layers. Thus, in this chapter we assume only a simple local broadcast service that allows nearby nodes to communicate reliably.

The goal of this chapter is to provide a brief overview of how to modify the abstractions and protocols described in the previous sections for a wireless network equipped only with a simple local broadcast service. We begin in Section 7.1 by describing the behavior of the local broadcast service. (The other attributes of the physical model remain unchanged.) Next, in Section 7.2, we briefly describe a virtual infrastructure model that includes only local communication. In Section 7.3, we describe how to implement the fpcast service (described in Chapter 2) using the simple local broadcast service. Finally, in Section 7.4, we describe the few modifications to the emulator algorithm from Chapter 5 needed to implement the local-only virtual infrastructure model described in this chapter.

There are two reasons to focus on local communication. First, it is a more realistic model for wireless networks. Most wireless communication is local in nature, and thus it makes sense to focus on what can be accomplished using only local communication. Second, a local virtual node layer may enable the construction of many interesting non-local communication mechanisms. For example, we can build long-distance GeoCast protocols on top of a local virtual node layer; these GeoCast protocols may have advantages (in terms of simplicity and reliability, say) over GeoCast protocols built directly on a mobile ad hoc network.

7.1 Local Broadcast

We begin by considering a modification of the physical model presented in Chapter 2: instead of a GeoCast service and several fpcast services, we assume only a "local broadcast" service. All other aspects of the model remain unchanged. Formally, the lbcast service is part of the RealWorld automaton; we describe it here in terms of its safety and liveness properties.

The local broadcast service lbcast is parameterized by two values: a radius, R_{lb} and a message delay d_{lb}. When some node i performs a $\mathsf{lbcast}(m)_i$, the lbcast service delivers the message—via a $\mathsf{lbcast\text{-}rcv}(m)_j$—to every mobile node j within a radius R_{lb} of the sender; every message is delivered within d_{lb} time. Formally, the service has the following properties:

***Reliable Delivery*:** Assume that the mobile node i performs a $\mathsf{lbcast}(m)_i$ in location ℓ at time t. Then for every mobile node j that is within distance R_{lb} of location ℓ at time t and remains non-failed within distance R_{lb} of location ℓ until time $t + d_{lb}$, a $\mathsf{lbcast\text{-}rcv}(m)_j$ event occurs by time $t + d_{lb}$, delivering the message to node j.

***Integrity*:** For any message m and mobile node i, if a $\mathsf{lbcast\text{-}rcv}(m)_i$ event occurs, then a $\mathsf{lbcast}(m)_j$ event precedes it, for some mobile node j.

***Latency Lower Bound*:** Each message takes some time $> t_{\mathsf{upd}}$ to be delivered.

Intuitively, sending a message using this service should be thought of as making a single wireless broadcast (with a small number of retries, if necessary, to avoid collisions). For small R_{lb}, this service is a reasonable (if simplistic) model of sending and receiving messages using wireless broadcast.

7.2 Local Virtual Node Layer

We next introduce a virtual infrastructure abstraction that contains only local communication. As in the Virtual Node Layer described in Chapter 3, we consider an abstraction consisting of clients and virtual nodes. Each client is associated with a mobile node in I; each virtual node is associated with a focal point in O.

The clients and virtual nodes communicate with each using a virtual lbcast service. Unlike the Virtual Node Model presented in Chapter 3 in which clients and virtual nodes could send messages to each other via geocast, in this case, all communication is local. If a client or a virtual node performs an virtual-lbcast, only nearby clients and virtual nodes receive the message.

For the purpose of this revised virtual node model, we define the focal points more specifically. Each focal point is defined in terms of its focal point center—which may move through the network. The focal point region is defined as a circle of some fixed radius around the focal point center. Recall that R_{lb} is the radius of the local broadcast service, v_{max} is the maximum velocity of a mobile node, and $v_{\mathsf{fp\text{-}max}}$ is the maximum velocity of a focal point. Formally, if, at time t, the focal point center is at

$\mathsf{fp\text{-}center}(t)_h$, we define $\mathsf{fp\text{-}region}(t)_h$ as a circle of radius $R_{\mathsf{lb}}/2 - t_{\mathsf{upd}}(v_{\mathsf{max}} + v_{\mathsf{fp\text{-}max}})$. As in Chapter 2, the additional buffer zone inside the circle defined by the local broadcast service ensures that nodes can communicate using the lbcast service as long as they believe they are in a focal point.

7.3 The Focal Point Broadcast Service

A key building block for the protocols presented in Chapters 4 and 5 is the fpcast service which guarantees reliable delivery of messages in some fixed total order to mobile nodes in a focal point. Since most real networks are not by default equipped with such a service, we have describe a local broadcast service lbcast that captures in a simplified manner the behavior of a reliable wireless MAC layer.

In this section, we show how to implement the fpcast service in networks that provide only a local broadcast service using a standard ordering technique originally suggested by Lamport [58]. We fix one focal point $h \in O$ for the remainder of this section, and discuss the implementation of $\mathsf{fpcast}_{\mathsf{h}}$.

For the purpose of this section, we assume that t_{upd} is arbitrarily small, i.e., that each mobile node has access to precisely real time at any instant. Formally, this is equivalent to restricting our attention to executions in which a geo-update event occurs immediately prior to every other event. It is relatively straightforward to extend the claims in this section to tolerate slight inaccuracies in the time.

We now proceed to describe the protocol. Pseudocode for the fpcast protocol can be found in Figure 7-1.

Implementing fpcast. The lbcast service already guarantees integrity and reliable delivery. The key property that needs to be enforced is the total ordering property. This ordering is enforced through the use of timestamps.

Recall that each mobile node receives frequent updates via geo-update events of the most recent time. Each message is tagged with the time when it is sent. Messages are delivered in order (with ties broken arbitrarily and deterministically, see Figure 7-1, lines 41–43) at time d_{lb} after the time at which they are tagged (see Figure 7-1, line 40). If a mobile node ever moves too far away from the focal point center, i.e., if it ever exits the focal point, then all waiting messages that have not yet been delivered are dropped (see Figure 7-1, lines 60–61). If a node ever receives a messages that is too old, i.e., its timestamp is earlier than $t - d_{\mathsf{lb}}$, where t is the current time according to the receiving node, then the message is discarded (see Figure 7-1, lines 24–26). (Notice that a message may be delivered late, despite the reliable delivery deadline, if, say, the sending node is farther than distance R_{lb} away, for example.) If a node was not in the focal point when a message was sent, it is discarded (Figure 7-1, line 25).

It follows immediately that the fpcast service guarantees integrity:

Lemma 7.3.1. fpcast_h *satisfies the integrity property.*

Proof. A message m is delivered only if it is received from the lbcast service, and by the integrity of the lbcast service we can conclude that m was previously broadcast

Figure 7-1: Automaton $\mathsf{fpcast}_\mathsf{h}$ for focal point h and client i.

Signature:
Input:
  $\mathsf{lbcast\text{-}rcv}(\langle m, t, j\rangle)_i$, m a message, $t \in \mathbb{R}$, $j \in I$
  $\mathsf{fpcast}(m)_i$, m a message
  $\mathsf{geo\text{-}update}(t, \ell)_i$, $t \in \mathbb{R}$, $\ell \in \mathbb{R} \times \mathbb{R}$

Output:
  $\mathsf{lbcast}(\langle m, t, j\rangle)_i$, m a message, $t \in \mathbb{R}$, $j \in I$
  $\mathsf{fcast\text{-}rcv}(m)_i$, m a message

Internal:
  $\mathsf{discard\text{-}old}(\langle m, t, j\rangle)_i$, m a message, $t \in \mathbb{R}$, $j \in I$

State:
  lbcast-queue, a queue of $\langle m, t, j\rangle$, where m is a message, $t \in \mathbb{R}$ a tag, and $j \in I$ a node
  waiting-msgs, a set of $\langle m, t, j\rangle$, where m is a message, $t \in \mathbb{R}$ a tag, and $j \in I$ a node
  last-entrance $\in \mathbb{R}$
  time $\in \mathbb{R}$

Transitions:

Input $\mathsf{lbcast\text{-}rcv}(\langle m, t, j\rangle)_i$
Effect:
  if $(time \geq t - d_{\mathsf{lb}})$ **then**
    if $(t \leq last\text{-}entrance)$ **then**
      $waiting\text{-}msgs \leftarrow waiting\text{-}msgs \cup \{\langle m, t, j\rangle\}$

Output $\mathsf{lbcast}(m)_i$
Precondition:
  $m \in lbcast\text{-}queue$
Effect:
  $lbcast\text{-}queue \leftarrow lbcast\text{-}queue - \{m\}$

Input $\mathsf{fpcast}(m)_i$
  $lbcast\text{-}queue \leftarrow lbcast\text{-}queue \cup \langle m, time, i\rangle$

Output $\mathsf{fpcast\text{-}rcv}(m)_i$
Precondition:
  $\langle m, t, j\rangle \in waiting\text{-}msgs$
  $t + d_{\mathsf{lb}} = time$
  $\nexists\ t' < t : \langle \cdot, t', \cdot\rangle \in waiting\text{-}msgs$
  $\nexists\ j' < j : \langle \cdot, t, j'\rangle \in waiting\text{-}msgs$
  $\nexists\ m' < m : \langle m', t, j\rangle \in waiting\text{-}msgs$
Effect:
  $waiting\text{-}msgs \leftarrow waiting\text{-}msgs - \{\langle m, t, j\rangle\}$

Internal $\mathsf{discard\text{-}old}(m)_i$
Precondition:
  $\langle m, t, j\rangle \in waiting\text{-}msgs$
  $time > t + d_{\mathsf{lb}}$
Effect:
  $waiting\text{-}msgs \leftarrow waiting\text{-}msgs - \{\langle m, t\rangle\}$

Input $\mathsf{geo\text{-}update}(t, \ell)_i$
  $time \leftarrow t$
  if $(last\text{-}entrance = \perp)$ **and** $(\ell \in fp\text{-}region(t)_h)$ **then**
    $last\text{-}entrance \leftarrow t$
  if $(last\text{-}entrance \neq \perp)$ **and** $(\ell \notin fp\text{-}region(t)_h)$ **then**
    $last\text{-}entrance \leftarrow \perp$
  if $(|\ell - \mathsf{fp\text{-}center}_h| > R_{lb}/2 + t_{\mathsf{upd}}(v_{\mathsf{max}} + v_{\mathsf{fp\text{-}max}}))$ **then**
    $waiting\text{-}msgs \leftarrow \emptyset$

Trajectories:
stops when
  $\exists\ \langle m, t, j\rangle \in waiting\text{-}msgs : t + d_{\mathsf{lb}} \leq time$
    or
  $lbcast\text{-}queue \neq \emptyset$

by some node i. □

It is also easy to see that messages are delivered in the same order at all mobile nodes. Specifically, they are delivered in the order specified by their time stamps:

Lemma 7.3.2. fpcast_h *satisfies the total order property.*

Proof. Consider two messages m_r and m_s tagged with (and sent at) times t_r and t_s respectively. Assume without loss of generality that $t_r \leq t_s$, and that some node i delivers both messages m_r and m_s. Then we argue that i delivers m_r prior to m_s. Specifically, notice that i delivers message m_s at time $t_s + d_{\mathsf{lb}}$, by the rule on when i delivers messages. Moreover, i delivers m_r only if m_r is received no later than time $t_r + d_{\mathsf{lb}}$; otherwise, m_r is discarded by the rule discarding late messages. Since $t_r \leq t_s$, we conclude that i receives m_r prior to delivering m_s. Thus, because i delivers messages in order according to timestamps, we conclude that i delivers m_r prior to delivering m_s. □

Next, we argue that $\mathsf{fpcast}_\mathsf{h}$ guarantees reliable delivery. In particular, we argue that within time d_{lb} after a message m is broadcast, every nearby node receives it:

Lemma 7.3.3. $\mathsf{fpcast}_\mathsf{h}$ *satisfies the reliable delivery property.*

Proof. Assume that for some mobile node i and some message m, a $\mathsf{fpcast}(m)_{h,i}$ event occurs at time t, and that i is in focal point h at time t. Moreover, assume that node j is also in focal point h at time t and remains in focal point h and non-failed through time $t + d_{\mathsf{lb}}$. We need to show that j delivers message m by time $t + d_{\mathsf{lb}}$.

First, notice that i re-broadcasts the message m with the lbcast service immediately at time t, and t is thus the tag associated with message m in the local broadcast. Since focal point h has diameter $\leq R_{\mathsf{lb}}/2$, and since i is in focal point h when the lbcast occurs, and since j is in the focal point through time $t + d_{\mathsf{lb}}$, we conclude from the reliable delivery property of the lbcast service that by time $t + d_{\mathsf{lb}}$ node j receives the message m via a $\mathsf{lbcast\text{-}rcv}_j$. Since j has remained in the focal point from the time at which m was broadcast through time $t + d_{\mathsf{lb}}$, we conclude that $\mathit{last\text{-}entrance}_i \leq t$. Thus message m is added to *waiting-msgs* and delivered. □

Next, we argue that $\mathsf{fpcast}_\mathsf{h}$ satisfies the consistent delivery property. Specifically, we show that if some non-failed mobile node i receives messages m_1 and m_3 and does not leave the focal point between receiving messages m_1 and m_3, and if there is some messages $m_2 : m_1 < m_2 < m_3$ according to the order induced by the tags, then that mobile node receives messages m_2.

Lemma 7.3.4. $\mathsf{fpcast}_\mathsf{h}$ *satisfies the consistent delivery property.*

Proof. Let m_1, m_2, and m_3 be three messages such that a $\mathsf{fpcast}(m_r)$ occurs for each $r \in \{1, 2, 3\}$. Moreover, assume that the total ordering property induces an order $m_1 < m_2 < m_3$ on these messages. Recall (as was shown in the proof of Lemma 7.3.2) that this order is induced by the tags t_1, t_2, t_3, respectively, tags associated with each message when it is rebroadcast using the lbcast survice.

Assume that some node i delivers messages m_1 and m_3, and that node i does not exit the focal point region for h from the point at which i receives m_1 through the point at which i receives m_3. We show that i receives message m_2.

First, we argue that i is in the focal point when m_2 is sent. There are three cases depending on when m_2 is sent with respect to when m_1 is first received by i, and when m_1 is delivered. Consider the case where m_2 is broadcast prior to m_1 first being received by i: then we know that $\textit{last-entrance}_i < t$, and hence i last entered the focal point prior to m_1 and m_2 being broadcast. Next, consider the case where m_2 is broadcast after m_1 is received by i but before m_1 is delivered by i: we know that i does not exit the focal point during this interval, since it discards all the *waiting-msgs* when it exits the focal point. Finally, consider the case where m_2 is broadcast after m_1 is delivered by i: then by assumption i remains in the focal point until m_3 is delivered; since $t_1 \leq t_2 \leq t_3$, we know that m_2 is broadcast prior to m_3 being delivered.

Next, notice that i remains in the focal point through time $t_2 + d_{\mathsf{lb}}$, i.e., long enough for m_2 to be delivered. Specifically, we know that i does not exit the focal point prior to $t_3 + d_{\mathsf{lb}}$, or m_3 would be discarded, and $t_2 \leq t_3$. We therefore conclude that by the reliable delivery property of the fpcast service in Lemma 7.3.3, message m_3 is delivered. □

7.4 Remaining Implementation Details

In this section we discuss the remaining changes that need to be made to the Virtual Node Emulator described in Chapter 5 in order to emulate the Local Virtual Node Layer. We have already shown in Section 7.3 how to implement the fpcast service. The Virtual Node Emulator also makes use of a geocast service—which is not available in the revised model.

Notice, however, that the emulator only uses the geocast service to forward messages from the VNE-Client automata to the VNE-Server automata. Instead, we use the lbcast service to transmit messages between these components of the emulator. As a result, a message broadcast by a client (in the abstraction) is only received by a VNE-Server in the emulation if the client is near to the focal point.

This, however, satisfies exactly the local communication property desired of the revised virtual infrastructure abstraction. The analysis of the resulting protocol is exactly identical to that described in Chapter 5, with the only modification being the revised "reliable delivery" argument: the emulator ensures reliable delivery since if a client is close to a focal point, then its message reaches a gen-server and the argument proceeds as before.

Part II

Virtual Infrastructure: Collision-Prone Networks

Introduction

There are several challenges associated with developing algorithms for ad hoc networks. In the first part of this thesis, we proposed the idea of virtual infrastructure to address the difficulties arising from unreliable devices and unpredictability. In this second part of the thesis, we address the problem of unreliable communication. We begin in Chapter 8 by introducing a new model for wireless networks that includes unreliable communication, as well as node failures and mobility. We then define a new type of virtual infrastructure in Chapter 9. Next, we present an algorithm that implements this form of virtual infrastructure (Chapter 10), and prove that the algorithm is correct (Chapter 11). The possibility of lost messages introduces several difficulties, and leads to a much more involved emulation algorithm[1].

Overview

It is an unfortunate fact that communication in wireless networks is unreliable: collisions and other wireless interference cause significant message disruption in real deployments of ad hoc wireless networks. Several recent experimental studies [37, 52, 105, 108] suggest that even with sophisticated collision avoidance mechanisms (e.g., 802.11 [1], B-MAC [86], S-MAC [107], and T-MAC [101]), and even under low traffic loads, the fraction of messages being lost can be as high as $20 - 50\%$.

By contrast, the virtual infrastructure algorithms presented in Part I of this thesis depend significantly on reliable communication. Recall that in order to implement a virtual node (or even a virtual object), the state of the virtual node is replicated at a set of mobile nodes, and the mobile nodes run a replicated-state-machine protocol in order to maintain a consistent replicated state. The main protocol depends on the reliable dissemination of messages in order to ensure that each replica receives the same set of messages. Moreover, the replicated-state-machine protocol in Part I depends on an even stronger property: it assumes that messages are delivered to each mobile node in the same order; alternatively (as is discussed in Chapter 7), it assumes that messages are delivered in a timely fashion. Real wireless networks provide neither of these guarantees.

Thus we conclude that, while the idea of virtual infrastructure, as proposed in Part I, may be effective at coping with unreliable mobile devices and unpredictable

[1]Notice that although the emulation algorithm itself is quite involved, the abstraction presented to the developer building applications atop the virtual node layer remains relatively simple, much as in Part I of this thesis.

patterns of mobility, the actual implementations of virtual infrastructure presented in Part I are not directly suitable for deployment in real wireless ad hoc networks. The primary goal of Part II is to design and implement a virtual infrastructure that can tolerate unreliable communication, and thus is more suitable for real wireless networks. We now proceed to present an overview of Part II.

The Basic Model for Wireless Ad Hoc Networks

We begin Part II by introducing (in Chapter 8) a new model for wireless ad hoc networks that allows for unreliable communication. As in Part I, the network consists of a set of mobile nodes moving arbitrarily in the two-dimensional plane. The key difference from Part I, however, is that the mobile nodes communicate via a synchronous, *unreliable* communication service. That is, each mobile node broadcasts and receives messages in a synchronous, round-based fashion, and messages may be lost.[2] Each mobile node is modelled by a timed I/O automaton.

To cope with unpredictable message loss, we augment the system with receiver-centric *collision detectors*. Intuitively, collision detectors attempt to notify a mobile node when a message that it should have received was lost. Notice that a collision detector does not provide any information to the original transmitter as to whether its messages were received. Intuitively, a collision detector monitors the broadcast medium and attempts to deliver notifications when messages are lost; a simple implementation of a collision detector might be based on "carrier sensing," a technique that raises a warning whenever a certain signal threshold is reached. Collision detectors provide no information on the number of lost messages or the identities of the nodes that send the messages. Moreover, there is no guarantee that a node performing a transmission can detect collisions. A collision detector answers only the binary question of whether there appears to have been at least one message lost.

Moreover, we consider collision detectors that are themselves unreliable, as first introduced in [23, 79]. There are two types of unreliable behavior: *false negatives*, in which the collision detector fails to detect a collision, and *false positives*, when the collision detector reports a collision inaccurately. In practice, we believe it possible to detect collisions correctly most of the time. In Part II, we focus primarily on faulty collision detectors that detect occasional false positives, but eventually stabilize and become reliable[3].

To this point, we have made no assumptions on the reliability of message delivery. In order for an algorithm to achieve progress, however, eventually some messages must be delivered. In wireless networks that share a single channel of of the electromagnetic spectrum, messages can be delivered when there is no interference on the channel.

[2]Notice that it is possible to view the communication service in Part I as, essentially, synchronous: the suggested implementation for the the fpcast service relies on a timely communication service, which is effectively equivalent to a synchronous system.

[3]As elsewhere in this thesis, when we consider a stabilization point after which the collision detectors become reliable, in practice we simply require the collision detectors to behave correctly for sufficiently long, i.e., long enough for our protocols to make progress.

That is, if only one node in a region broadcasts, there is the possibility that its message is delivered. We assume that, *eventually*, if the broadcast contention is low enough in a given round, then there are no collisions and the message is delivered to every nearby mobile node. We refer to this property as "eventual collision freedom." Since the property holds only in an eventual sense, however, and since the nodes are unaware of when the network stabilizes, a transmitting node can never be certain that its messages are received.

Finally, we assume that the nodes have access to a set of *contention managers* which provide advice on when a node should broadcast and when a node should be silent, leaving the channel free for its neighbors to broadcast. Again, the contention managers are unreliable: they often give bad advice, and there is no guarantee that a transmitting node that is advised to broadcast will succeed in avoiding interference. We assume, however, that eventually the contention managers stabilize and provide good advice.

The Virtual Infrastructure System

In Chapter 9 we present the virtual infrastructure system. Much like the virtual infrastructure described in Part I, the abstraction consists of *virtual nodes*, residing at fixed points in the network, and *clients*, that move arbitrarily. The clients and virtual nodes communicate using a *broadcast service* much like that available to the underlying mobile nodes: communication is unreliable, and the broadcast service guarantees eventual collision freedom. Each virtual node and each client receives feedback from a collision detector as to when messages are lost, and advice from a contention manager as to when to broadcast and when to be silent. In general, the virtual infrastructure system strongly resembles the underlying network model, with the main difference being the existence of additional virtual nodes that are more reliable and predictable than the underlying mobile nodes. The additional robustness of the virtual nodes, in contrast to the unreliable mobile nodes, provides significant benefit to developers attempting to build application atop a virtual infrastructure layer.

The Emulator Algorithm

In Chapter 10 we present an algorithm to emulate the virtual infrastructure in the underlying network model. The algorithms itself is quite involved, and we introduce it in a series of steps. The chapter begins with an overview of the algorithm (Section 10.1), and then proceeds to present the algorithm formally and in more detail (Section 10.2). The chapter concludes with an argument that the resulting automata are well-formed, satisfying the appropriate requirements of the basic model (Section 10.3).

As in Part I, each virtual node is emulated by a set of replicas that happen to be near to the virtual node's designated location. This replication guarantees fault tolerance: as mobile nodes enter the virtual node's region, they join the emulation; as mobile nodes leave the virtual node's region, they leave the emulation.

As in Part I, the key difficulty in implementing virtual infrastructure is maintaining the consistency of the replicas. Unlike in Part I, however, we cannot assume that the broadcast service is reliable (or that it guarantees totally-ordered message delivery), and hence a more involved scheme is needed to maintain the replica consistency.

In order to maintain consistency, the replicas participate in repeated instances of an "agreement protocol": each instance of the agreement protocol represents one (virtual) round of the virtual infrastructure emulation. The goal of each instance is for the replicas to agree on a specific execution for the virtual node up to and including the (virtual) round associated with that instance. When the agreement protocol completes successfully, all the replicas share the same virtual node execution, and hence share the same virtual node state.

Each instance of the agreement protocol takes a fixed number of (basic) rounds. When the network is "unstable," agreement may not be reached within the alotted number of rounds: lost messages, false positives from a collision detector, or bad contention management advice can result in a failed agreement instance. In this case, the decision as to the appropriate behavior for the associated (virtual) round is postponed, and the next agreement instance is begun. Eventually, once the network has stabilized, the agreement instances complete successfully, and the entire execution up to the current virtual round—including all previously undetermined rounds—is determined.

Each replica needs to distinguish (virtual) rounds in which the agreement instance has completed successfully, and (virtual) rounds in which the agreement instance has failed. To this end, the agreement protocol is similar, in ways, to "three-phase commit" protocols (see, e.g., [95,96]): each replica determines a color for each round—*green*, *yellow*, *orange*, or *red*—which determines the level of success for that instance of agreement. At any two replicas, the color differs by at most one shade. Thus, if a replica determines that a (virtual) round is green, it can safely conclude that the agreement instance was successful, as no replica designated the round as red. Similarly, if a replica determines that a (virtual) round is red, it can safely conclude that the agreement instance failed.

The resulting iterated agreement protocol ensures that at the end of each (virtual) round, all the replicas behave in a consistent manner, and that eventually, when the network stabilizes, the virtual node simulated by the replicas successfully broadcasts and receives messages without inducing collisions.

Proof of Correctness

In Chapter 11 we prove that the protocol presented in Chapter 10, when executed in the basic model described in Chapter 8, implements the virtual infrastructure system presented in Chapter 9.

The proof consists of two main parts. In the first part of the proof (Sections 11.2–11.8), we prove a series of properties about the round colors, and the associated states of all the replicas. We show that the color of a round at two replicas differs by at most one shade (Lemma 11.5.9 and Corollary 11.5.7), and that after a "good" round, all the replicas agree on the state of the virtual node (Lemma 11.7.11). Finally, we show

that eventually, when the network is stable, all the rounds are good (Lemma 11.8.12).

In the second part of the proof (Sections 11.9–11.15), we explicitly construct the execution of the virtual infrastructure that is emulated by a given execution of the emulator protocol. We construct each of the component executions separately, defining an execution for each virtual node (Section 11.9), for each client and contention manager (Section 11.10), for the collision detector (Section 11.12), and finally for the broadcast service (Section 11.13). In the process, we prove that the resulting executions satisfy a series of necessary properties, such as communication integrity (Section 11.11). Finally, we "paste" the various components executions together (Section 11.14), resulting in an execution of the entire virtual infrastructure system. We then conclude by proving some properties about the entire system: we show that the resulting system satisfies eventual collision freedom (Section 11.15), and analyze when the virtual nodes fail (Section 11.16). Specifically, we show that a virtual node fails only when it is depopulated by mobile nodes acting as replicas, and that it recovers when it is repopulated.

Chapter 8

Modelling a Wireless Ad Hoc Network

In this chapter, we present a new model for wireless ad hoc networks. This model is designed to capture the problems that arise from lost messages (caused by collisions), as well as the challenges that arise from mobility and crash failures. In Section 8.1 we present a general model for wireless networks, and in Section 8.2 we present a specialized variant of this model, the *Basic System*, which we use throughout Part II of this thesis.

8.1 General Systems

In this section, we describe the general class of systems discussed in this thesis. Each system models a wireless network consisting of a fixed but *a priori* unknown collection of nodes that may move on a two-dimensional plane. The nodes communicate using a *synchronous* broadcast service. Each node is provided with information from a "collision detector", which provides feedback on collisions, and information from a set of "contention managers", which help the node to reduce collisions. In this section we formally define these components, modeling a general system as a composition of timed I/O automata. See Appendix A for a brief overview of timed I/O automata theory; for more details, see [45, 46]. In general, the goal of this section is describe a typical synchronous model (with the addition of collision detection and contention management) using the TIOA formalism[1].

We begin in Section 8.1.1 by enumerating the various parameters that define a general system: a set of nodes, a set of ports per node, a series of parameters that determine the behavior of the radio broadcast, a set of processes that run on the nodes, an algorithm that maps nodes to processes, and a collision detector. In the rest of this section, we discuss these parameters in more detail, as well as defining the automata that they parameterize.

[1]Most prior formalizations of synchronous systems do not include the tools needed in Part II of this thesis. In particular, they do not include the capacity to compose synchronous automata, or the capacity to construct simulations across multiple levels of abstraction.

In Section 8.1.2 we define nodes, processes and algorithms, the basic computational components of the system. We also present a schema for translating a traditional synchronous protocol into an algorithm for a general system, and we describe a toy example algorithm.

In Section 8.1.3, we define collision detectors, and specify certain classes of collision detectors that will be used throughout this thesis. The main class of collision detectors defined here is the class of "eventually accurate, always complete" collision detectors ($\Diamond\mathcal{A}$-$\mathcal{C}$). In Section 8.1.4, we define contention managers, and present the canonical contention manger. A contention manager provides advice to the nodes on how to reduce the contention on the broadcast channel, and can be used to encapsulate the workings of randomized backoff protocols, as well as other methods of contention resolution.

In Section 8.1.5 we define the broadcast service that is used by the nodes to communicate. Additionally, the broadcast service maintains some global state as to the status of the system, such as the location of the nodes, the failure status of the nodes, etc. We specify the broadcast service as an automaton, and describe its properties.

In Section 8.1.6, we define general system $\mathcal{G}$ which consists of process automata, a broadcast service, and a set of contention managers. We also define the synchronous round structure, associating each event with a round number, and discuss finite and infinite executions of a general system.

In Section 8.1.7, we describe some message-delivery guarantees that are provided by the general system. Specifically, we discuss integrity, self-delivery, and per-round delivery properties. We also discuss the relationship between the collision detector guarantees and the broadcast service message-delivery guarantees.

In Section 8.1.9, we introduce an additional assumption on the broadcast service, called "eventual collision freedom," which ensures that eventually, if there is no contention on the broadcast channel, then messages will be delivered (Definition 8.1.56).

In Section 8.1.8, we specify some additional useful properties of a contention manager. We describe four different properties of contention managers: eventual non-interference, eventual fairness, eventual contention fairness, eventual regional fairness. We also specify what it means for a contention manager to be "conservative."

Finally, in Section 8.1.10, we provide a brief discussion on modifying the general system to model devices that can control their own motion, such as mobile robots.

8.1.1 Parameters for a General System

A general system is defined by eleven parameters:

Definition 8.1.1. *We define the* ***parameters for a general system*** *as follows:*

1. *I, a non-empty, finite set of node names;*
2. *process-ports, a non-empty, finite set of port names; we define the* ***broadcast ports*** *bcast-ports to be $I \times$ process-ports; we say that $j \in$ bcast-ports is* ***associated*** *with node $i \in I$ if there exists some port $p \in$ process-ports such that $j = \langle i, p \rangle$;*

3. *part, a partition of the broadcast ports;*

4. *msgs, a non-empty set of messages;*

5. *CM-names, a non-empty, finite set of contention manager names;*

6. $R \in \mathbb{R}$ *and* $R' \in \mathbb{R}$*, broadcast and interference radii, respectively, where* $R > 0, R' > 0$*;*

7. *RndLength* $\in \mathbb{R}$*, the length of a synchronous round, where* $RndLength > 0$*;*

8. *P, a non-empty set of processes for* $\langle I, \textit{process-ports}, \textit{msgs}, \textit{CM-names}\rangle$ *(see Definition 8.1.2);*

9. *A, an algorithm for* $\langle I, P\rangle$*, mapping nodes in I to processes in P (see Definition 8.1.5);*

10. *CD, a collision detector for bcast-ports (see Section 8.1.3); and*

11. $\{CM_d \mid d \in \textit{CM-names}\}$*, a non-empty, finite set of contention managers for* $\langle I, \textit{process-ports}, \textit{msgs}, \textit{CM-names}\rangle$ *(see Section 8.1.4).*

A general system, which we define formally in Definition 8.1.19, is a set of composed automata that are defined by these parameters, specifically (1) a set of (remapped) processes, as defined in Definition 8.1.9, Section 8.1.2, (2) a broadcast service, parameterized by $\langle R, R', \textit{msgs}, \textit{RndLength}, \textit{bcast-ports}, \textit{part}, \textit{CD}\rangle$, as defined in Section 8.1.5, and (3) a set of contention managers, $\{CM_d \mid d \in \textit{CM-names}\}$, as defined in Section 8.1.4.

8.1.2 Nodes, Processes, and Algorithms

In this section, we describe the nodes that constitute a system, the processes that execute on the nodes, and the algorithms that specify the behavior of the nodes.

Informally, a system consists of a set of fault-prone physical devices. These devices may be mobile and may move in an arbitrary manner, with only the limitation that their velocity is bounded by v_{max}. The algorithms running on a node may provide control information to guide the motion, as in the case of mobile robots, or may have no control over the motion.

Each physical device executes a "process" (i.e., a restricted TIOA), which captures the computational capacity of the device. We therefore think of an "algorithm" as assigning a process to each device. When a device fails, the associated process takes no further (locally-controlled) steps. A device may also recover at some later time, reset to its initial state. For the most part, we will assume that nodes do not recover. (A recovered node can simply be treated as a new node that has just arrived in the region.) We include the possibility of recovery here as it will be useful later in Chapter 9 when defining the notion of a "virtual node."

In this thesis, we model protocols that are unaware of the number of devices in the system: while the universe of possible devices may be known *a priori*, an unknown, arbitrarily large, subset may have failed at the very beginning of the execution.

We also model "anonymous" protocols, in which the nodes in the system do not have unique identifiers. By separating processes—which may not have unique identifiers—from nodes—which have unique names—we are able to capture both anonymous and name-aware protocols.

Processes

In this section, we define a **process**, which models the computation executed by a particular mobile node in the physical system. Formally, a process is defined to be a timed I/O automaton [45, 46]; furthermore, we assume that a process TIOA is *internally progressive*, which means that in any finite interval of time, if there are only a finite number of inputs, then there are only a finite number of locally-controlled actions.

A process is parameterized by three of the general system parameters: *process-ports*, *msgs*, and *CM-names*. The set *process-ports* represents the a set of communication ports used by the process; *msgs* is the communication alphabet; and *CM-names* names the contention managers that the process may utilize.

Definition 8.1.2. *A **process** for $\langle$process-ports, msgs, CM-names$\rangle$ is an internally progressive timed I/O automaton A that satisfies the following restrictions:*

Static Restrictions:

1. (Input actions) *Automaton A must have two types of input actions:*
   - $\mathsf{recv}(M, cd, cm, loc)_i$, *where* $M \subseteq$ *msgs is a set of messages,* $cd \in \{\pm, \mathsf{null}\}$, $cm \in \{\mathsf{active}, \mathsf{passive}\}$, $loc \in (\mathbb{R} \times \mathbb{R})$, *and* $i \in$ *process-ports;*
   - $\mathsf{fail}()$.

   In addition, automaton A may have a third type of input action:
   - $\mathsf{recover}()$.

   Automaton A has no other type of input action.
2. (Output actions) *Automaton A must have one type of output action:*
   - $\mathsf{bcast}(m, cm)_i$, *where* $m \in msgs_\perp$ *is a message or* $\perp$, $cm \in$ *CM-names*$_\perp$, *and* $i \in$ *process-ports.*

   Automaton A has no other type of output action.
3. (Discrete variables) *Automaton A has only discrete state variables, implying that all trajectories of A are constant*[2]

Dynamic Restrictions:

[2] *A trajectory defines how an analog variable evolves with time in between discrete actions. See Appendix A for a brief review, or [45, 46] for more details.*

4. (Immediate response) *For any execution α of automaton A, the following two conditions hold for each $i \in$ process-ports:*

   (a) *Each* $\mathsf{bcast}(\ldots)_i$ *event in α is preceded by a* $\mathsf{recv}(\ldots)_i$ *event, and there is no other intervening* $\mathsf{bcast}(\ldots)_i$ *event.*

   (b) *If a* $\mathsf{recv}(\ldots)_i$ *event occurs at time $t \geq 0$, and $\ell time(\alpha) > t$, then either: (i) α contains a* $\mathsf{bcast}(\ldots)_i$ *event at time t, after the* $\mathsf{recv}(\ldots)_i$ *event, or (ii) α contains a* $\mathsf{fail}()$ *event at some time $\leq t$ and no intervening* $\mathsf{recover}()$ *event.*

   (See, for example, Figure 8-6; notice that the round 4 bcast *event immediately follows the round 3* recv *event.)*

5. (Failure) *In any execution α of automaton A, no locally-controlled action is preceded by a* fail *event with no intervening* recover *event.*

6. (Recovery) *In any execution α of automaton A, if a* recover *event occurs in α, then immediately after the* recover *event, state $\in$ start.*

The first three restrictions are static restrictions on the form of the automaton, while the last three restrictions limit its dynamic behavior, enforcing a round-based, failure-respecting execution.

Restrictions 1 and 2 on input and output actions ensure that a process has the typical broadcast/receive interface. The parameters of the broadcast and receive actions are described later in Sections 8.1.3, 8.1.4 and 8.1.5, where we discuss the collision detector, contention manager, and broadcast service, respectively.

Informally, when a process performs a $\mathsf{bcast}(m, cm)$, it is attempting to broadcast message m, and indicating an attempt to "contend" using contention manager cm in the following round; that is, the process is requesting that contention manager cm advise it to be active in the following round. If $m = \perp$, the process is indicating that it has no message to send. If $cm = \perp$, the process is indicating that it does not want to contend; that is, every contention manager should advise it to be passive.

Informally, when a $\mathsf{recv}(M, cd, cm, loc)_i$ event occurs, the broadcast service delivers a set of messages M and collision information cd; it advises the process to be either active or passive in the following round, depending on the value of the "advice", cm; the last parameter, loc reflects the current location of the process.

The fail and recover inputs described in Restriction 1 control when the process has failed and recovered:

Definition 8.1.3. *A process p is* ***recoverable*** *if it has an input action* recover*. Otherwise, it is* ***unrecoverable****.*

In Section 8.2, we will restrict our attention to "basic systems" in which processes running on nodes are unrecoverable. In Chapter 9, however, we will consider systems containing processes running on virtual nodes that may recover.

Definition 8.1.4. *For an execution α of unrecoverable process p, we say that p is* ***correct*** *in α if no* fail *event occurs in α. Otherwise, we say that p is* ***faulty*** *in α.*

Restriction 3—variables are discrete—captures the assumption that state is updated only when discrete events occur: recall (see Appendix A) that a discrete variable in TIOA is one that does not change as time passes. That is, for every trajectory, the variable is constant.

Restriction 4 ensures that for each recv event, a matching bcast event follows immediately. Restriction 4(a) ensures a one-to-one matching between broadcast and (preceding) recv events, while Restriction 4(b) ensures that the broadcast event occurs immediately after the preceding recv event. Together, these conditions ensure that the process, in cooperation with the broadcast service, maintains an alternating sequence of bcast and recv events, starting with a recv.

Restriction 5 ensures that a failed process takes no locally-controlled steps. When a process recovers, it can again take steps. Restriction 6 ensures that a recovered process restarts in an initial state.

Transformation Schema

In order to illustrate the connection between our description of a synchronous system and a typical synchronous model, we briefly discuss how to translate a typical synchronous protocol, such as those described in [70], Chapters 2–6, into our model. Synchronous algorithms are often described in terms of two functions: (1) a recv-function which takes the current state and the current set of messages and produces a new state for the next round, and (2) a bcast-function which takes the current state and produces the message to send in the next round. (See, for example, our prior work on consensus in wireless networks [23, 79].) The automaton in Figure 8-1 illustrates how to translate a synchronous automaton specified in this manner into a legal process meeting the restrictions described above. (Assume, for the purpose of the example, that the set *states* represents the states of the synchronous automaton, and *start* is an initial state.)

Example Process

As another example, Figure 8-2 presents a simple toy process, CounterProcess, which illustrates one way to describe a process syntactically using TIOA formalism. Note that this simple process does little of interest; it is for illustrative purposes only. For the purpose of this example, *CM-names* = {CM}. The CounterProcess maintains a simple counter. Whenever it receives a set of messages and no collisions (line 21, Figure 8-2), it adds the values of the messages to its count (lines 22–23, Figure 8-2). When it is active (line 42, Figure 8-2), it broadcasts its count (line 43, Figure 8-2); otherwise, it broadcasts $\perp$ (line 44, Figure 8-2). Occasionally, non-deterministically, it resets its count (lines 49–54, Figure 8-2).

First, we argue that the automaton in Figure 8-2 is a timed I/O automaton, specifically, that it is input-enabled and time-passage enabled. The former is immediate, as input actions have no preconditions in TIOA syntax. Observe that the automaton is time-passage enabled since whenever the stopping condition in the trajectories (line 58, Figure 8-2) is true, then the bcast(m, cm) event is enabled for some

Automaton SynchronousSchema

State:
  discrete $s \in states$, initially $s \in start$
  discrete $do\text{-}bcast \in \{\textbf{true}, \textbf{false}\}$, initially false
  discrete $failed \in \{\textbf{true}, \textbf{false}\}$, initially false

Signature:
Input:
  recv$(M, cd, cm, loc)_i$, $M \subseteq msgs$, $cd \in \{\pm, \text{null}\}$, $cm \in \{\text{active}, \text{passive}\}$, $loc \in \mathbb{R} \times \mathbb{R}$
  fail$()_i$
  recover$()_i$
Output:
  bcast$(m, cm)_i$, $m \in msgs_\perp$, $cm \in CM\text{-}names_\perp = \{\text{CM}, \perp\}$

Transitions:

Input recv$(M, cd, cm, loc)_i$
Effect:
  $s \leftarrow$ recv-function(s, M, cd, cm, loc)
  $do\text{-}bcast \leftarrow$ **true**

Input fail$()_i$
Effect:
  $failed \leftarrow$ **true**

Input recover$()_i$
Effect:
  $s \leftarrow s' : s' \in start$
  $do\text{-}bcast \leftarrow$ **false**
  $failed \leftarrow$ **false**

Output bcast$(m, cm)_i$
Precondition:
  $m =$ bcast-function(s)
  $do\text{-}bcast =$ **true**
  $cm =$ CM
  $failed =$ **false**
Effect:
  $do\text{-}bcast \leftarrow$ **false**

Trajectories:
stops when
  ($do\text{-}bcast =$ **true**) **and** ($failed =$ **false**)

Figure 8-1: An automaton schema for transforming a classical synchronous automaton into a process, i.e., a synchronous device in a general system as described in this chapter. In this case, the port set $process\text{-}ports = \{i\}$, and the set of $CM\text{-}names = \{\text{CM}\}$. Assume that the function bcast-function indicates which message should be sent, and that the function recv-function indicates how to update the state upon receiving a set of messages, collision detection information, and contention management information. The set *states* is the set of possible states, and *start* is the initial state. The boxes indicate the key transition functions that define the synchronous automaton.

m and cm. Specifically, when *do-bcast* = true and *failed* = false, if the local variable *active* = true, then bcast(*count*, CM) is enabled; otherwise bcast($\perp$, CM) is enabled.

Next, we argue that the example in Figure 8-2 meets the additional restrictions of a process. First, note that it has the requisite signature, consisting of broadcast and receive actions, along with fail and recover actions. It is recoverable and movement oblivious. There is an additional internal reset action, reset, which is allowed. As per Restriction 3, all the variables are discrete.

Restriction 4 is enforced by the flag *do-bcast*. Whenever a recv event occurs, the flag is set to true. The *stops when* condition in the trajectories (line 58, Figure 8-2) ensures that if the *do-bcast* flag is set and the automaton has not failed, then no time can pass. The precondition for bcast ensures that a broadcast event occurs only when the *do-bcast* is set.

Restriction 5 is enforced by the *failed* flag, which is set by the fail input, unset by the recover input, and checked by the precondition of each locally-controlled action. Restriction 6 is met by the recover action, which resets each variable to an initial state.

Finally, notice that the automaton is internally progressive. Specifically, the internal event reset and the output event bcast can only happen once for every recv event: the reset event can occur only when *count* > 0, and only a bcast can increment the count; the bcast event can occur only in response to a recv event as per Restriction 4 discussed above. Thus if there are a finite number of input events in an execution, then there are also a finite number of locally-controlled events, which implies that the automaton is internally progressive.

Algorithms

An **algorithm** specifies how the processes in P execute on the mobile nodes I. Formally, we define an algorithm as a mapping from I to P, which assigns a process to each mobile node.

Definition 8.1.5. *An **algorithm** for $\langle I, P\rangle$ is a mapping from I to processes in P.*

Notice that if all mobile nodes in I are mapped to the same process, then the algorithm is effectively "anonymous," as all the processes are identical:

Definition 8.1.6. *We say that algorithm A is **anonymous** if for all $i, j \in I$, $A(i) = A(j)$.*

By contrast, if each mobile node $i \in I$ is mapped to a distinct process, then the processes may contain unique identifiers that indicate which mobile node each process is running on. In this thesis, we mainly consider anonymous algorithms that do not require unique identifiers.

For the purposes of modelling a system, we want to disambiguate which automata are associated with which nodes, even when the system is anonymous, i.e., the nodes are executing identical automata. Thus, we need to identify the actions of each process with the mobile node that maps to it. We thus rename the actions as follows:

Automaton CounterProcess

```
1  State:
2      discrete count ∈ ℕ0, initially 0
3      discrete active ∈ {true, false}, initially false
4      discrete do-bcast ∈ {true, false}, initially false
5      discrete failed ∈ {true, false}, initially false
6
7  Signature:
8  Input:
9      recv(M, cd, cm, loc), M ⊆ msgs, cd ∈ {±, null}, cm ∈ {active, passive}, loc ∈ ℝ× ℝ
10     fail()
11     recover()
12 Output:
13     bcast(m, cm), m ∈ msgs⊥, cm ∈ CM-names⊥ = {CM, ⊥}
14 Internal:
15     reset()
16
17 Transitions:
18
19 Input recv(M, cd, cm, loc)
20 Effect:
21     if (cd ≠ ±) then
22         for all m ∈ M do
23             count ← count +m
24     active ← cm
25     do-bcast ← true
26
27 Input fail()
28 Effect:
29     failed ← true
30
31 Input recover()
32 Effect:
33     count ← 0
34     active ← false
35     do-bcast ← false
36     failed ← false
37
38 Output bcast(m, cm)
39 Precondition:
40     do-bcast = true
41     cm = CM
42     if active = true
43         then m = count
44         else m = ⊥
45     failed = false
46 Effect:
47     do-bcast ← false
48
49 Internal reset()
50 Precondition:
51     count > 0
52     failed = false
53 Effect:
54     count ← 0
55
56 Trajectories:
57 stops when
58     (do-bcast = true) and (failed = false)
```

Figure 8-2: An example process. In this simple example, the process is designed to maintain a counter, which is occasionally reset to 0. Whenever it receives a message, if there is no collision then it adds every value received to its count. When it is active, it broadcasts its count.

Definition 8.1.7. *Given a process p and a node $i \in I$, the function* **Remap(p, i)** *produces a new automaton q in which each action $a \in p$ is renamed to a_i in q.*

As a typographic convention, if action a already contains subscripts, we will place the node identifier as the first subscript. For example, if $a = \mathsf{bcast}(m, cm)_v$, then we will write the remapped action a_i as $\mathsf{bcast}(m, cm)_{i,v}$.

A "remapped process" is the result of remapping a process to a mobile node:

Definition 8.1.8. *We say that q is a* ***remapped process*** *if there exists some process p and some node $i \in I$ such that $q = \mathsf{Remap}(p, i)$. In this case, we say that automaton q is* ***remapped*** *from process p and* ***associated*** *with node i.*

The system as a whole will contain the set of remapped processes as defined by A:

Definition 8.1.9. *Let $P(A, I) = \{\mathsf{Remap}(A(i), i) : i \in I\}$.*

Notice that given an algorithm A and a set of mobile nodes I, the set of remapped processes $P(A, I)$ is compatible (see Definition A.2.10, Appendix A), and thus can be composed.

8.1.3 Collision Detectors

We assume that every node i receives *collision detector* information that indicates when messages are lost[3]. The collision detector may not be wholly reliable, and it delivers relatively limited information. Yet the addition of even unreliable collision detection capacity significantly expands the set of problems that can be solved in a wireless network. (See, for example, [23, 79].) In this section, we formally define a collision detector. The definitions and notation used here were first introduced in [23,79]. We discuss the integration of collision detection into the broadcast service in Section 8.1.7.

Formally, we define a collision detector rule *CD-rule* as follows:

Definition 8.1.10. *A* ***collision detector rule*** *for bcast-ports is a function from bcast-ports $\times \mathbb{N} \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$ to $\{\pm, \mathsf{null}\}$.*

A collision detector rule will be invoked in the context of a synchronous broadcast service; the six arguments, intuitively, are used to communicate the behavior of some particular round of the broadcast service. In more detail, the six arguments of *CD-rule* are as follows:

1. $i \in$ *bcast-ports*, a port,
2. $rnd \in \mathbb{N}$, a round number,
3. *num-sent* $\in \mathbb{N}_0$, intuitively, the number of messages sent by nodes within distance R of port i in round rnd,

[3]The definitions and notation from this section were originally developed in [23, 79]

4. *num-rcved* $\in \mathbb{N}_0$, intuitively, the number of messages received by port i in round *rnd* that were sent by nodes within distance R of port i in round *rnd*.

5. *num-sent-interfere* $\in \mathbb{N}_0$, intuitively, the number of messages sent by nodes within distance R' of port i in round *rnd*,

6. *num-rcved-interfere* $\in \mathbb{N}_0$, intuitively, the number of messages received by port i in round *rnd* that were sent by nodes within distance R' of port i in round *rnd*.

Notice that the difference between *num-sent* and *num-sent-interfere* is related to the radius of interest: *num-sent* will be used with respect to messages sent by nodes within radius R, the broadcast radius; *num-sent-interfere* will be used with respect to messages sent by nodes with radius R', the interference radius. (The same holds for *num-rcved* and *num-rcved-interfere*.)

For a given collision detector rule, the output depends only on the number of messages sent and received in a specific round, irrespective of the contents of those messages, and irrespective of the prior history.

Definition 8.1.11. *A **collision detector** CD for bcast-ports is a non-empty set of collision detector rules for bcast-ports.*

We classify collision detectors based on *completeness*—lack of false negatives, and *accuracy*—lack of false positives. Completeness measures the reliability with which collisions are detected: a complete collision detector will detect every lost message. Accuracy measures the reliability with which non-collisions are detected: an accurate collision detector will report a collision only when a message is lost. More formally:

Definition 8.1.12. *We say that collision detector rule CD-rule is **complete** if the following property holds:*

- *For every rnd* ≥ 0*, for every* $i \in$ *bcast-ports, for num-sent* $\in \mathbb{N}_0$*, num-rcved* $\in \mathbb{N}_0$*:*
  if num-sent $>$ *num-rcved* ≥ 0*, then CD-rule*$(i, rnd, num\text{-}sent, num\text{-}rcved, \cdot, \cdot) = \{\pm\}$.

Notice that completeness depends on messages sent within radius R, the broadcast radius: the detector should warn of messages sent by nearby nodes that are lost. That is, completeness depends on *num-sent* and *num-rcved*.

Definition 8.1.13. *We say that collision detector rule CD-rule is **accurate** if the following property holds:*

- *For every rnd* ≥ 0*, for every* $i \in$ *bcast-ports, for num-sent-interfere* $\in \mathbb{N}_0$*, num-rcved-interfere* $\in \mathbb{N}_0$*:*
  if num-sent-interfere $\leq$ *num-rcved-interfere,*
  then CD-rule$(i, rnd, \cdot, \cdot, num\text{-}sent\text{-}interfere, num\text{-}rcved\text{-}interfere) = \{\mathsf{null}\}$.

Accuracy depends on messages sent within radius R': if a message is sent by a node within distance R', and that message is not received, then the detector may raise an alarm. Thus, accuracy depends on *num-sent-interfere* and *num-rcved-interfere*.

Due to occasional false positives, it may be hard to guarantee perpetual accuracy. Thus, we also define collision detectors that guarantee eventual accuracy:

Definition 8.1.14. *We say that collision detector rule CD-rule is* ***eventually accurate*** *if the following property holds:*

- *There exists a round r_{acc} such that for every round $rnd \geq r_{acc}$, for every $i \in$ bcast-ports, for num-sent-interfere, num-rcved-interfere $\in \mathbb{N}_0$:*
  if num-sent-interfere $\leq$ num-rcved-interfere,
  then CD-rule$(i, rnd, \cdot, \cdot,$ num-sent-interfere, num-rcved-interfere$) = \{\mathsf{null}\}$.

In this thesis, we will focus on collision detectors that are complete and eventually accurate:

Definition 8.1.15. *We say that a collision detector CD is in the class $\Diamond\boldsymbol{\mathcal{A}}$-$\boldsymbol{\mathcal{C}}$ if every rule CD-rule $\in$ CD is complete and eventually accurate.*

8.1.4 Contention Managers

We model randomized backoff protocols as playing the role of a *contention manager*, which provides advice to each broadcast port as to whether it should be active—that is, broadcast—in a round. In each round, each contention manager receives input from each broadcast port as to whether it wants to perform a broadcast, that is, whether it wants to contend for the channel. The goal of a contention manager is to guarantee that if some port contends for long enough, then some port is given exclusive access to the channel. Some contention managers might guarantee fairness properties: if some port contends for long enough, then *that specific* port gains access to the channel.

We assume that the system is equipped with a set of contention managers, each of which may be responsible for some aspect of the protocol, some subset of the nodes, or some specific region of the network. We assume that *CM-names* is a non-empty set of contention manager names, and that *CM-names* does not include the special symbol $\perp$.

Formally, a contention manager is defined as follows:

Definition 8.1.16. *A* ***contention manager*** *with name $d \in$ CM-names is a progressive, timed I/O automaton A, along with a set of liveness properties, satisfying the following restrictions:*

1. *Automaton A has one type of input action, $\mathsf{bcast}(m, cm)_i$, where m is a message or $\perp$, $cm \in$ CM-names$_\perp$, and $i \in$ bcast-ports.*

2. *Automaton A has one type of output action: $\mathsf{cm\text{-}advice}(i, a)_d$, where port $i \in$ bcast-ports and $a \in \{\mathsf{active}, \mathsf{passive}\}$.*

3. *For any execution α of automaton A, each* $\mathsf{cm\text{-}advice}(i,\cdot)_d$ *event in α is preceded by a* $\mathsf{bcast}(\cdot,\cdot)_i$ *event, and there is no other intervening* $\mathsf{cm\text{-}advice}(i,\cdot)_d$ *event.*

4. *For any execution α of automaton A, if a* $\mathsf{bcast}(\cdot,\cdot)_i$ *event occurs at time t in α, and $\ell time(\alpha) > t$, then a* $\mathsf{cm\text{-}advice}(i,\cdot)_d$ *event occurs at time t, after the* bcast_i *event.*

The first two restrictions define the signature of a contention manager. When a broadcast port performs a $\mathsf{bcast}(\cdot, cm)_i$ in round r, it indicates to contention manger cm that it wants access to the channel in the following round $r+1$, if $cm \neq \perp$; otherwise, if $cm = \perp$, it indicates that it does not want access to the channel.

The contention manager guarantees—as per Restrictions 3 and 4—that each event $\mathsf{bcast}(\cdot,\cdot)_i$ is followed immediately by a $\mathsf{cm\text{-}advice}(i,\cdot)_d$ event.

Figure 8-3 presents the canonical contention manager automaton, canonicalCM, a simple automaton which, together with the empty set of liveness properties, satisfies the definition of a contention manager. The canonical contention manager places no constraints on what advice is output. In Chapters 9 and 10, we will consider contention managers that consists of the *canonicalCM* automaton, combined with a set of liveness properties that restrict the set of executions.

8.1.5 Synchronous Broadcast Service

The mobile nodes communicate using a synchronous broadcast service. In this section, we describe the broadcast service, discuss its parameters, and present a TIOA model for its operation. In Section 8.2, we assume that such a broadcast service exists in the underlying mobile network; in Chapter 10, we show how to emulate such a thesis as part of a virtual infrastructure system.

The broadcast service, when combined with a set of remapped processes, guarantees a round-based synchronous execution.

Definition 8.1.17. *For all $r \in \mathbb{N}$, we define* ***round*** *r as the closed interval of real time $[(r-1) \cdot RndLength, r \cdot RndLength]$.*

For example, if $RndLength = 1$, then round 1 is the interval $[0,1]$, round 2 is the interval $[1,2]$, etc. See Figure 8-6 for an illustration of how an execution is divided into rounds of length $RndLength$.

From the perspective of a process—which has no continuous clock to measure rounds—the first round begins at time 0 when the broadcast service delivers an initial (dummy) recv event. At this point, the process knows that the first round has begun. For each process, then, each round consists of two parts:

1. (*Broadcast*) The process broadcasts a message m in $msgs$ or the symbol $\perp$, indicating no message. The broadcast event, $\mathsf{bcast}(m, cm)$, includes an extra contention management parameter: $cm \in CM\text{-}names_\perp$, which indicates whether the process wants to broadcast in the following round, and if so, with which contention manager it wants to contend. As per Restriction 4 on processes (Definition 8.1.2), the broadcast occurs in response to the recv from the previous round.

Figure 8-3: Automaton *canonicalCM*: Canonical Contention Manager

State:
  discrete *waiting* $\subseteq$ *bcast-ports*, initially $\emptyset$

Signature:

Input:
  bcast(m, cm)$_i$, $m \in$ *msgs*, $cm \in$ *CM-names*, $i \in$ *bcast-ports*

Output:
  cm-advice(i, cm), $i \in$ *bcast-ports*, $cm \in$ {active, passive}

Trajectories:
  stops when:
    $waiting \neq \emptyset$

Input bcast(m, cm)$_i$
Effect:
  $waiting \leftarrow i$

Output cm-advice(i, cm)
Precondition:
  $i \in waiting$
Effect:
  $waiting \leftarrow waiting - \{i\}$

Figure 8-4: Automaton $\mathsf{Bcast}(R, R', RndLength, bcast\text{-}ports, part, CD)$, State, Signature, and Trajectories

State:
  System State:
    analog $time \in \mathbb{R}$, initially 0
    analog $location[\,]$, an array of $\mathbb{R} \times \mathbb{R}$ with one entry per *bcast-port*, initially the initial location of each port
    discrete $failed \subseteq bcast\text{-}ports$, initially $\emptyset$
    constant function $CD\text{-}rule : bcast\text{-}ports \times \mathbb{R} \times \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \{\pm, \mathsf{null}\}$,
      fixed initially to an arbitrary collision detector execution from the collision detector CD

  Broadcast State:
    discrete $round \in \mathbb{N}$, initially 0
    discrete $M[\,]$, array indexed by round number $\in \mathbb{N}_0$,
      where $M[r] \subseteq msgs_\perp \times bcast\text{-}ports_\perp \times (\mathbb{R} \times \mathbb{R})$, *initially* $[\langle \emptyset, \perp, \langle 0,0 \rangle \rangle, \langle \emptyset, \perp, \langle 0,0 \rangle \rangle, \ldots]$
    discrete $beginRound[\,]$, an array of $\mathbb{R} \times \mathbb{R}$ with one entry per *bcast-port*, initially $[\langle 0,0 \rangle, \langle 0,0 \rangle, \ldots]$
    discrete $doneP \subseteq bcast\text{-}ports$, initially $\emptyset$
    discrete $advice[\,]$, map from *bcast-ports* to $\{\mathsf{active}, \mathsf{passive}\}$, initially $\forall p \in bcast\text{-}ports, advice[p] = \mathsf{passive}$

Derived Variables:
  discrete $M[r].msgs = \{m : \exists\ i \in bcast\text{-}ports, \ell \in \langle \mathbb{R} \times \mathbb{R} \rangle : \langle m, i, \ell \rangle \in M\text{-}locs[r]\}$
array indexed by round number $\in \mathbb{N}_0$, where $M[r].msgs \subseteq msgs_\perp$

Signature:
Input:
  $\mathsf{bcast}(m, cm)_i$, $m \in msgs_\perp$, $cm \in CM\text{-}names_\perp$, $i \in bcast\text{-}ports$
  $\mathsf{cm\text{-}advice}(i, cm)_d$, $i \in bcast\text{-}ports$, $cm \in \{\mathsf{active}, \mathsf{passive}\}$, $d \in CM\text{-}names$
Output:
  $\mathsf{fail}()_i$, $i \in bcast\text{-}ports$
  $\mathsf{recover}()_i$, $i \in bcast\text{-}ports$
  $\mathsf{recv}(some\text{-}msgs, cd, cm, loc)_i$, $some\text{-}msgs \subseteq msgs$, $cd \in \{\pm, \mathsf{null}\}$, $cm \in \{\mathsf{active}, \mathsf{passive}\}$, $loc \in \mathbb{R} \times \mathbb{R}$, $i \in bcast\text{-}ports$
Internal:
  $\mathsf{next\text{-}round}()$

Trajectories:
stops when
  $time = round \cdot RndLength$
  or
  $\exists\ j,k \in bcast\text{-}ports, \exists\ i \in I : j$ and k are associated with node i and $\langle j, time \rangle \in failed$ and $\langle k, time \rangle \notin failed$
evolves:
  $d(time) = 1$
  $|d(location[i])| \leq v_{\max}, \forall i \in bcast\text{-}ports$

Figure 8-5: Automaton Bcast($R, R', RndLength, bcast\text{-}ports, part, CD$), Transitions

Input bcast(m, cm)$_i$
Local State:
  $r \in \mathbb{N}$
Effect:
  $r \leftarrow \lfloor time/RndLength \rfloor + 1$
  $M[r] \leftarrow M[r] \cup \langle m, i, beginRound[i] \rangle$

Input cm-advice(i, cm)$_d$
Effect:
  if (cm = active) **then**
    $advice[i] \leftarrow cm$

Internal next-round()
Precondition:
  $doneP \cup failed = bcast\text{-}ports$
  $time = round \cdot RndLength$
Effect:
  $round \leftarrow round + 1$
  $doneP \leftarrow \emptyset$
  $\forall i \in bcast\text{-}ports$: $beginRound[i] \leftarrow location[i]$

Output fail()$_i$
Precondition:
  $\nexists\ t \in \mathbb{N}$: $\langle i,t \rangle \in failed$
Effect:
  $advice[i] \leftarrow$ passive
  $failed \leftarrow failed \cup \langle i, time \rangle$

Output recover()$_i$
Precondition:
  $\exists\ t$ such that $\langle i, t \rangle \in failed$
  $time \geq t + RndLength$
Effect:
  $failed \leftarrow failed - \langle i, t \rangle$

Output recv($some\text{-}msgs$, cd, cm, loc)$_i$
Local State:
  $sentM \subseteq msgs$
  $rcvedM \subseteq msgs$
  $sentMinterfere \subseteq msgs$
  $rcvedMinterfere \subseteq msgs$
Precondition:
  $time = round \cdot RndLength$
  $some\text{-}msgs \subseteq M[round].msgs - \{\perp\}$
  if $\langle m, i, \cdot \rangle \in M[round]$ **then**
    $m \in some\text{-}msgs$
  $i \notin doneP$
  $i \notin failed$
  let $sentM = \{m \neq \perp$: $\exists\ j \in bcast\text{-}ports$, $\langle m, j, \cdot \rangle \in M[round]$, $|beginRound[j] - beginRound[i]| \leq R\}$
  let $rcvedM = sentM \cap some\text{-}msgs$
  let $sentMinterfere = \{m \neq \perp$: $\exists\ j \in bcast\text{-}ports$, $\langle m, j, \cdot \rangle \in M[round]$, $|beginRound[j] - beginRound[i]| \leq R'\}$
  let $rcvedMinterfere = sentMinterfere \cap some\text{-}msgs$
  $cd = CD\text{-}rule(i, round, |sentM|, |rcvedM|, |sentMinterfere|, |rcvedMinterfere|)$
  $cm = advice[i]$
  $loc = location[i]$
Effect:
  $advice[i] \leftarrow$ passive
  $doneP \leftarrow doneP \cup i$

2. (*Receive*) Messages are delivered to the process via the action $\mathsf{recv}(M, cd, cm, loc)$, where $M \subseteq msgs$ is a set of messages; $cd \in \{\pm, \mathsf{null}\}$ and $cm \in \{\mathsf{active}, \mathsf{passive}\}$ are related to collision detection and contention management, respectively, and are discussed later in the section; $loc \in (\mathbb{R} \times \mathbb{R})$ is the location of node i.

Along with coordinating the synchronous round structure, the broadcast service here also models portions of the environment; specifically, it is responsible for maintaining general execution information, such as the real time and the location of the mobile nodes. It is convenient to model this global state as part of the broadcast service[4]. Thus, the broadcast service notifies nodes when they fail, as well as simply delivering messages.

We now discuss the parameters that define the broadcast service. The formal TIOA specification of the broadcast service is presented in Figures 8-4 and 8-5. We then describe the state maintained by the broadcast service, and present an overview of the operation of the broadcast service. Finally, we argue that the broadcast service is a progressive timed I/O automaton.

In Section 8.1.7, we present some additional liveness properties that may be associated with the Bcast automaton.

Broadcast Service Parameters

The broadcast service is defined by six parameters:

1. $R \in \mathbb{R}^{>0}$, a broadcast radius governing the range of the wireless broadcast.

2. $R' \in \mathbb{R}^{>0}$, $R' \geq R$, an interference radius governing how far away broadcasting nodes must be to guarantee non-interference. When $R = R' = 1$, this describes the model commonly referred to as the "unit-disc model." When $R < R' = 1$, this describes the model commonly referred to as the "quasi-unit-disc model."

3. *RndLength* $\in \mathbb{R}^{>0}$, the length of a round. The round length should be long enough to allow a single device to broadcast a single message. For the basic broadcast service, *RndLength* $= 1$. The virtual broadcast service will have a larger *RndLength*.

4. *bcast-ports*, the set of broadcast ports.

5. *part*, a partition of broadcast ports into non-intersecting sets. Conceptually, broadcasts by ports in different sets of the partition should not interfere with each other. One could imagine different sets in the partition representing different wireless frequencies, or different slots in a pre-allocated time-division schedule.

6. *CD*, a collision detector for *bcast-ports*, as described in Section 8.1.3).

[4]In Part I of this thesis, we referred to the combined broadcast service and GeoSensor as the "RealWorld."

Broadcast Service State

Figure 8-4 presents the signature and state for the reference TIOA specification for the broadcast service. In this section, we describe the state maintained by the broadcast service.

We divide the state variables into two categories: (1) system state, related to the general operation of the entire system, and (2) broadcast state, related to the specific operation of the broadcast service.

First, we discuss the system state (lines 2–7, Figure 8-4). The broadcast service maintains information about the ongoing execution, including the current time, the location of the nodes, the failure status of the nodes, and the collision-detector rule:

- *time* is the current real time. The evolution clause $d(time) = 1$ (line 37, Figure 8-4) indicates that it progresses at exactly the rate of real time.

- *location*[] is an array that maintains the exact geographical (2-space) location of each port in the system. These locations evolve non-deterministically, subject only to the restriction that the nodes may not move too fast, i.e., they have maximum velocity v_{max} (line 38, Figure 8-4). Notice that the location of each node changes continuously; sudden jumps in the location are disallowed.

- *failed* is the set of nodes that are currently failed, along with the time at which they failed. Conceptually, a node in the system fails "atomically," in the sense that all the automata associated with the same node should receive fail and recover events at the same real time. The broadcast service, however, interacts with processes on a per-port basis. The *stops when* condition in line 35, Figure 8-4, ensures that all the ports associated with a single node fail and recover together. Specifically, time cannot pass until either (1) all the ports associated with a given node are in the set *failed*, or (2) none of the ports associated with a given node are in the set *failed*.

- *CD-rule* is a specific collision detector rule chosen from the collision detector *CD*, a set of collision detector rules. The *CD-rule* component of the state is constant, in that it does not change during an execution. At the beginning of an execution of the broadcast service, a specific execution of the collision detector is fixed, non-deterministically. This modelling choice provides a simple way to allow non-determinism in the collision detector.

The second part of the state consists of the broadcast state (lines 9–9, Figure 8-4), which maintains the current round, the set of messages broadcast in each round, the set of ports that have received messages in each round, and the contention manager advice delivered to each port. The broadcast state consists of the following components:

- The variable *round* maintains the current round of the execution. Round 1 begins at time 0, and ends at time 1; in general, round r begins at time $(r-1) \cdot RndLength$ and ends at time $r \cdot RndLength$. The internal action next-round

increments the round number when the previous round is complete. Rounds are discussed further in Section 8.1.6.

- $M[]$ is an array with one entry for each round of the execution. The entry $M[r]$ stores the set of messages that are broadcast in round r. Each element in the array consists of three components: (1) a message $m \in msgs_\perp$, (2) a port $i \in bcast\text{-}ports$, indicating which port broadcast message m, and (3) a location $\ell \in \mathbb{R} \times \mathbb{R}$ indicating the location of the port (as per $location[]$ in the system state) which broadcast the message at the time that the message was sent. These three components provide all the necessary information about the messages sent in round r. The message array M is updated only when a bcast action occurs. (See lines 1–6, Figure 8-4.)

  We define the derived variable $M[r].msgs \subseteq msgs_\perp$ to be the set of messages (without the associated port and location information) sent in round r. That is, $M[r].msgs$ is the set $\{m : \exists i \in bcast\text{-}ports, \ell \in \langle \mathbb{R} \times \mathbb{R} \rangle : \langle m, i, \ell \rangle \in M[r]\}$.

- $beginRound[]$, is an array that maintains the exact location of each port in the system at the beginning of the current round. It is useful throughout the round to remember where the nodes were when the round began.

- $doneP$ is a bookkeeping variable that records the set of ports that have already received a message in the current round. The $doneP$ set is updated whenever a recv event occurs, delivering a message to some port. When every port has either received a message or failed, then the round is complete. That is, when $doneP \cup failed = bcast\text{-}ports$ and time has reached the end of the current round, then a next-round action can occur. (See lines 13–20, Figure 8-4.)

- $advice$ maintains the current information on which nodes should be active, according to the contention managers. Each contention manager provides input to the broadcast service via cm-advice$(i, cm)_d$ events, where $i \in bcast\text{-}ports$ and $cm \in \{\text{active}, \text{passive}\}$. (See lines 8–11, Figure 8-4.) Each such event indicates that port i should either be active or passive in the following round. This advice is then passed to the port itself during a recv event. (See line 54, Figure 8-4.)

Broadcast Service Transitions Overview

In this section, we provide an overview of the broadcast service transitions in Figure 8-5. We begin with the two input actions, bcast and cm-advice, followed by the internal action next-round, and concluding with the output actions fail, recover, and recv.

Input Actions:

- The primary purpose of the broadcast service is to allow a node on some port $i \in bcast\text{-}ports$ to broadcast a message: bcast$(m, cm)_i$ (lines 1–6, Figure 8-5), where m is a message in $msgs_\perp$ and $cm \in CM\text{-}names_\perp$ is a request to a contention manager. First, the broadcast service calculates

the round r in which the message should be delivered (line 5, Figure 8-5), and then adds that message to the set of messages $M[r]$. The round structure is described further in Section 8.1.6.

- The input $\mathsf{cm\text{-}advice}(i, cm)_d$ (lines 8–11, Figure 8-5) allows the contention manager CM_d to pass information to the broadcast service. Specifically, it provides advice $cm \in \{\mathsf{active}, \mathsf{passive}\}$ with respect to port $i \in$ *bcast-ports*. If any contention manager advises a node to be active in round r, then $advice[i]$ is set to active; otherwise, it remains set to passive

Internal Actions:

- The internal action next-round (lines 13–20, Figure 8-5) advances the round when two criteria are met: first, the broadcast service has delivered messages for the previous round to every non-failed node, as checked by ensuring that every port is represented in the set $doneP \cup failed$, and second, it is the correct real time for the next round to begin. This action increments the round number, resets $doneP$ to $\emptyset$, and records the locations of all the ports. It is useful throughout the round to keep track of where the nodes were at the beginning of the round.

Output Actions:

- The output actions fail_i (lines 22–27, Figure 8-5) and $\mathsf{recover}_i$ (lines 29–34, Figure 8-5) cause port i to fail and recover, respectively. Any port that is not failed—as determined by the *failed* set and checked by the precondition of the fail action—may fail at any time. Similarly, any port that has failed more than time *RndLength* ago may recover. When a fail_i event occurs, the $advice[i]$ is reset to passive, preparing for the possibility that node i will recover in a later round.

  Notice that both fail and recover events can only happen with finite frequency: each port can only fail and recover once in every round. Also, notice that fail and recover outputs occur at ports, while in the real world, one might expect all the ports associated with a given node should fail and recover at the same time. This property is enforced by the *stops when* condition on line 35 of Figure 8-4, which ensures that two ports associated with the same node share the same failure/recovery status.

- Finally we describe the receive output action which delivers messages, along with other information, at the end of each round. Specifically, the receive action $\mathsf{recv}(\mathit{some\text{-}msgs}, cd, cm, loc)_i$ (lines 36–58, Figure 8-5) delivers messages and other information to port $i \in$ *bcast-ports*. The receive event happens precisely at the end of each round (see Section 8.1.6). The recv action has the following properties, as a result of its preconditions:
  - The recv event occurs exactly at the end of the round (line 43, Figure 8-5).

– *some-msgs* is some subset of the messages that should be delivered in the current round (line 44, Figure 8-5).
– If port i sends a message in round r, then port i receives its own message in round r (lines 45–46, Figure 8-5).
– Messages are delivered only once in a round (line 47, Figure 8-5), and only if port i has not failed (line 48, Figure 8-5).
– $cd \in \{\pm, \mathsf{null}\}$ indicates whether there is a collision, as determined by the given collision detector execution (lines 49–53, Figure 8-5). The local variable *sentM* calculates the set of messages broadcast in round r by a node that was nearby at the beginning of the round. Specifically, for a recv_i event, it includes all messages sent by a node j where $|location[i] - location[j]| \leq R$ at the beginning of round r. The local variable *sentMinterfere* calculates the set of messages broadcast in round r by a node in the interference radius at the beginning of the round, that is, all messages sent by a node j where $|location[i] - location[j]| \leq R'$ at the beginning of round r. The local variable *rcvedM* calculates the subset of *sentM* that is delivered by the recv event, while *rcvedMinterfere* calculated the subset of *sentMinterfere* that is delivered by the recv event.
– $cm \in \{\mathsf{active}, \mathsf{passive}\}$ passes the appropriate contention management advice to the process (line 54, Figure 8-5). Recall that the variable $advice[i]$ stores the most recent advice from a contention manager.
– $loc \in (\mathbb{R} \times \mathbb{R})$ is the current location of the port when the recv event occurs (line 55, Figure 8-5).

When a recv_i action occurs, two state variables are modified: first, the $advice[i]$ is reset to passive, in preparation for the next round advice; second, the port i is added to the set *doneP*, which ensures that only one recv_i event can occur in the round.

In addition, we will assume that the broadcast service guarantees eventual collision freedom, which can roughly be interpreted as guaranteeing that eventually, if only one message is sent by all neighbors of port i, then port i receives that message. For more information on eventual collision freedom, see Section 8.1.9.

Progressive Timed Automaton

Before proceeding, we argue that the broadcast service is a progressive timed I/O automaton (see Definition A.2.8):

Lemma 8.1.18. *Automaton* $\mathsf{Bcast}(R, R', RndLength, bcast\text{-}ports, part, CD, update\text{-}loc)$ *is a progressive TIOA.*

Proof. First, we argue that the broadcast automaton is a TIOA. Specifically, we have to show that it is time-passage enabled (see Definition A.2.5). (It is immediate from

the TIOA syntax that it is input enabled.) There are two trajectories that result in time stopping. In each case, we show that some action is enabled.

First, consider the case when: $\mathit{time} = \mathit{round} \cdot \mathit{RndLength}$ (line 33, Figure 8-4). In this case, either $\mathit{doneP} \cup \mathit{failed} = \mathit{bcast\text{-}ports}$, in which case next-round() is enabled, or recv_i is enabled for $i \notin \mathit{doneP} \cup \mathit{failed}$. Specifically, notice that the action recv_i is enabled when $\mathit{some\text{-}msgs} = M[\mathit{round}] - \{\perp\}$, and cd and cm are set as per *CD-rule* and $\mathit{advice}[i]$.

Second, consider the case when:

$$\exists j, k \in \mathit{bcast\text{-}ports}, \exists i \in I: \quad \begin{array}{l} j \text{ and } k \text{ are associated with node } i \\ \quad \text{and } \langle j, \mathit{time}\rangle \in \mathit{failed} \\ \quad \text{and } \langle k, \mathit{time}\rangle \notin \mathit{failed} \end{array}$$

(line 35, Figure 8-4). Specifically, the following invariant holds in every execution: $\forall t < \mathit{time}$, if j and k are ports associated with some node $i \in I$, then $\langle j, t\rangle \in \mathit{failed}$ if and only if $\langle k, t\rangle \in \mathit{failed}$. This invariant follows immediately from the stops when condition: consider the last time at which the invariant holds in a prefix of the execution, and notice that $\mathit{time} = t$ until the invariant is true. Thus, assume that j and k are ports associated with some node $i \in I$, and $\langle j, \mathit{time}\rangle \in \mathit{failed}$ and $\langle k, \mathit{time}\rangle \notin \mathit{failed}$. We know that for all $t' < \mathit{time}$, $\langle k, t'\rangle \in \mathit{failed}$ if and only if $\langle j, t'\rangle \in \mathit{failed}$. Since $\langle j, \mathit{time}\rangle \in \mathit{failed}$, we know that $\langle j, t'\rangle \notin \mathit{failed}$, and hence $k, t' \notin \mathit{failed}$. Thus, the event fail_k is enabled.

Next, we argue that the automaton is progressive (Definition A.2.8). Consider some finite interval of time $[t_1, t_2]$. Notice that the recv_i action can only happen once for each time when $\mathit{time} = \mathit{round} \cdot \mathit{RndLength}$. Thus there are at most $\lceil (t_2 - t_1)/\mathit{RndLength} \rceil + 1$ receive outputs for each $i \in \mathit{bcast\text{-}ports}$. Since $\mathit{bcast\text{-}ports} = I \times \mathit{process\text{-}ports}$, and the sets I and *process-ports* are finite, *bcast-ports* is also finite, resulting in a finite number of receive actions. Similarly, the next-round action can occur only once for every interval of time of length *RndLength*. Finally, notice that for each port, the number of fail events can be at most one greater than the number of receive events; moreover each port can only recover once every round: the precondition ensures that if a port failed at time t, it can only recover after time $t + \mathit{RndLength}$. Hence there are at most a finite number of fail and recover events. □

8.1.6 General System Definition

In this section we formally define the general system as a composition of three types of automata: processes, a broadcast service, and a set of contention managers:

Definition 8.1.19. *The **general system***

$$G(I, \mathit{process\text{-}ports}, \mathit{msgs}, \mathit{CM\text{-}names}, P, A, R, R', \mathit{RndLength}, \mathit{part}, \mathit{CD}, \mathit{CM}) \ ,$$

consists of the composition of the following automata:

1. *$P(A, I)$, the set of remapped processes for $\langle A, I\rangle$,*

2. $\mathcal{B}(R, R', msgs, RndLength, bcast\text{-}ports, part, CD)$, *the broadcast service, where* $bcast\text{-}ports = I \times process\text{-}ports$.

3. $\{CM_d \in CM | d \in CM\text{-}names\}$, *the set of contention managers.*

Throughout the rest of this section, we occasionally refer to a general system $\mathcal{G}$, omitting the list of parameters that has already been specified above. We begin by noting that a general system as a whole is progressive:

Lemma 8.1.20. *Every general system $\mathcal{G}$ is a progressive timed I/O automaton.*

Proof. First, notice that each individual component is a TIOA: the broadcast service, the (remapped) processes, and the contention managers. Next, notice that the broadcast service and the contention manager are progressive. The composed set of (remapped) processes is internally progressive, since they share no input/output actions. Thus we can conclude that the composition of the (remapped) processes, the broadcast service, and the contention managers is internally progressive by Theorem A.2.11. However, the final composed automaton has no input actions, since each has been matched with an output action during the composition. Thus the resulting automaton is progessive. □

Synchronous Round Structure

A general system implements a round-based synchronous model. In this section we discuss the precise resulting round structure, associating each event in an execution of general system $\mathcal{G}$ with a round number. Throughout this section, recall that a round r is defined as the closed interval of time $[(r-1) \cdot RndLength, r \cdot RndLength]$, as per Definition 8.1.17. Recall also that if α is an execution of a timed I/O automaton, we say that an event e occurs at time t in α if $\ell time(\alpha') = t$, where α' is the prefix of α including all the events and trajectories preceding event e.

Broadcast and Receive Events. The broadcast service delivers messages precisely at the end of each round:

Definition 8.1.21. *For execution α of a general system $\mathcal{G}$, we say that a* $\mathsf{recv}(\ldots)_i$ *event at port $i \in bcast\text{-}ports$ is a* ***round*** *r* ***receive event*** *if it occurs in α at time $r \cdot RndLength$.*

Since the broadcast service delivers messages only at the end of a round, every receive event is thus assigned a round number. The processes broadcast messages at the beginning of each round:

Definition 8.1.22. *For execution α of a general system $\mathcal{G}$, we say that a* $\mathsf{bcast}(\ldots)_i$ *event at port $i \in bcast\text{-}ports$ is a* ***round*** *r* ***broadcast event*** *if it occurs in α at time $(r-1) \cdot RndLength$.*

See Figure 8-6 for an illustration of how bcast and recv events are assigned to rounds. Since each broadcast happens in response to a preceding receive event (Restriction 4 of processes), every broadcast event is thus assigned a round number. Notice that the round $r+1$ broadcast events occur at the same time as the round r receive events.

Contention Manager Events. We proceed to classify the cm-advice events by round:

Definition 8.1.23. *For execution α of a general system $\mathcal{G}$, for $r \geq 2$, we say that a* cm-advice$(i, cm)_d$ *event is a* ***round*** *r* ***advice event*** *if it occurs in α at time $(r-2) \cdot RndLength$.*

Notice that round r advice actually arrives at the beginning of round $r-1$. This event is the only type of event that does not occur in the round to which it is assigned. However, a round r cm-advice event provides advice for behavior in round r, and therefore it is assigned to round r, despite occurring in round $r-1$. See Figure 8-6 for an illustration of how cm-advice events are assigned to rounds.

Next Round Events. We next classify the next-round events:

Definition 8.1.24. *For execution α of a general system $\mathcal{G}$, we say that a* next-round *event is a* ***round*** *r* ***next-round event*** *if it occurs in α at time $(r-1) \cdot RndLength$.*

Thus, the round r next-round event occurs at the beginning of round r. See Figure 8-6 for an illustration of how next-round events are assigned to rounds.

Failure and Recovery Events. We next assign the failure and recover events to rounds:

Definition 8.1.25. *For execution α of a general system $\mathcal{G}$, we say that a* fail$_i$ *event is a* ***round*** *r* ***fail event*** *if r is the largest round such that it occurs after a round r* next-round *event. If it occurs prior to the first* next-round *event, i.e., the round 1* next-round *event, we refer to it as a round 0* fail *event.*

Definition 8.1.26. *For execution α of a general system $\mathcal{G}$, we say that a* recover$_i$ *event is a* ***round*** *r* ***recover event*** *if r is the largest round such that it occurs after a round r* next-round *event. If it occurs prior to the first* next-round *event, i.e., the round 1* next-round *event, we refer to it as a round 0* recover *event.*

See Figure 8-6 for an illustration of how fail and recover events are assigned to rounds.

Other Events. It remains to assign the internal events of processes to rounds:

Definition 8.1.27. *For execution α of a general system $\mathcal{G}$, let e be an event in α that is an action in the signature of some automaton p in $P(A, I)$, and not classified by Definitions 8.1.21–8.1.26. Let $i \in I$ be the mobile node identifier associated with automaton p.*

Let r be the largest integer > 0 such that for some port $j \in$ process-ports, e occurs either after a round r broadcast or after a round r recover event in α on port $\langle i, j \rangle$. If no such r exists, let $r = 0$. We say that e is a ***round*** *r* ***process event****.*

We use the broadcast events, rather than the next-round events, to classify the events that are local to a process so that it remains possible to assign rounds to events even when examining a an execution restricted only to a single process automaton.

The following straightforward lemma claims that every event is assigned to some round:

Lemma 8.1.28. *Let α be an execution of a general system $\mathcal{G}$. Let e be an event in α. Then there is some round $r \in \mathbb{N}$ such that e is a round r event.*

Proof. If e is a $\mathsf{recv}(\ldots)_i$ event, for some port i, then e is classified by Definition 8.1.21, since all receive events occur at some time $r \cdot \mathit{RndLength}$, by the precondition in line 43 of Figure 8-5.

If e is a $\mathsf{bcast}(\ldots)_i$ event, for some port i, then there is some preceding $\mathsf{recv}(\ldots)_i$ event—with no intervening broadcast event—according to Restriction 4(a) on processes. This preceding receive event must occur at some time $r \cdot \mathit{RndLength}$, as just discussed. There are two cases: (1) $\ell\mathit{time}(\alpha) = r \cdot \mathit{RndLength}$, in which case the broadcast event must occur at time $r \cdot \mathit{RndLength}$; (2) $\ell\mathit{time}(\alpha) > r \cdot \mathit{RndLength}$, in which case Restriction 4(b) on processes indicates that the broadcast event must occur at time $r \cdot \mathit{RndLength}$. In either case, the broadcast event is assigned to round $r + 1$.

If e is a $\mathsf{cm\text{-}advice}_d$ event, it is assigned to a round by Definition 8.1.23, since all $\mathsf{cm\text{-}advice}_d$ events occur immediately after bcast events, with no time passage, as per Restrictions (3) and (4) on contention managers. That is, the preceding broadcast event specified by Restriction (3) must occur at some time $r \cdot \mathit{RndLength}$, as just discussed. There are two cases: (1) $\ell\mathit{time}(\alpha) = r \cdot \mathit{RndLength}$, in which case the advice event occurs at time $r \cdot \mathit{RndLength}$; (2) $\ell\mathit{time}(\alpha) > r \cdot \mathit{RndLength}$, in which case Restriction (4) on contention managers indicates that the broadcast event must occur at time $r \cdot \mathit{RndLength}$. In either case, the advice event is assigned to round $r+2$ by Definition 8.1.23.

If e is a next-round event, it must occur at time $r \cdot \mathit{RndLength}$, for some r, as ensured by the precondition in line 16 of Figure 8-5.

If e is a fail_i or a $\mathsf{recover}_i$ event, it must either occur after some next-round event, or prior to all next-round events, and hence is assigned a round by Definition 8.1.25 or Definition 8.1.26.

If e is not one of the preceding events, it must be an event in the signature of a remapped process, and not covered by the previous cases. Thus it is assigned a round by Definition 8.1.27. □

Finite and Infinite Executions

We will often be interested in executions containing at least r rounds, for some $r \geq 1$. Thus, we say that α is an "r-round execution" when the broadcast service has completed round r and advances to the next round:

Definition 8.1.29. *For execution α of general system $\mathcal{G}$, for $r \in \mathbb{N}$, we say that α is an r-**round execution** if a $\mathsf{next\text{-}round}(r+1)$ event occurs in α.*

Notice that an r-round execution may contain events for rounds $> r$; when we say that an execution α is an r-round execution, it indicates that α includes *at least* r rounds.

We also consider infinite executions of a general system $\mathcal{G}$:

Definition 8.1.30. *We say that α is an **infinite execution** of a general system $\mathcal{G}$ if it is an execution of $\mathcal{G}$ and also an infinite sequence of trajectories and events.*

Since $\mathcal{G}$ is progressive, this implies that $\ell time(\alpha) = \infty$, and also that the number of rounds in α is unbounded.

Lemma 8.1.31. *Execution α is an infinite execution of a general system $\mathcal{G}$ if and only if it is an execution of $\mathcal{G}$ and $\ell time(\alpha) = \infty$.*

Proof. First, assume α is an infinite execution, i.e., an unbounded sequence of trajectories and actions. Since a general system is progressive, by Lemma 8.1.20, the execution must have infinite limit time.

Next, assume α has infinite limit time. Since the round length is finite and a next-round event occurs every round, there must be an infinite number of next-round events, and hence an infinite number of events and trajectories in the sequence α. □

Failure and Recovery

We can now refer to the round in which a process fails:

Definition 8.1.32. *For any $r \in \mathbb{N}$, for an r-round execution α of general system $\mathcal{G}$, we say that node $i \in I$ **fails in round** r if there is a round r fail_i event in α.*

(Recall that a round r fail event for node $i \in I$ occurs between the round r next-round and round $r+1$ next-round events for node i.) Similarly, a node can recover in a given round:

Definition 8.1.33. *For any $r \in \mathbb{N}$, for an r-round execution α of general system $\mathcal{G}$, we say that node $i \in I$ **recovers in round** r if there is a round r $\mathsf{recover}_i$ event in α.*

Thus, for any given round, a node is either failed or non-failed:

Definition 8.1.34. *For any $r \in \mathbb{N}$, for any r-round execution α of general system $\mathcal{G}$, we say that node $i \in I$ is **failed in round** r if, for some $r_{fail} \leq r$, the following two conditions hold:*

1. *Node i fails in round r_{fail}.*
2. *For all rounds r' such that $r_{fail} < r' < r$, node i does not recover in round r'.*

For example, in Figure 8-6, we say that i is failed in rounds 4 and 5.

There are two facts to note. First, if node i fails in some round r_{fail}, it cannot recover until round $r_{fail} + 1$, due to the restriction that a node can only recover time $> RndLength$ after it fails (line 32, Figure 8-5). Second, notice that even if node i recovers during round r, we consider node i to be failed in round r. This is because even if node i recovers in round r, it does not really participate in round r as it does not broadcast a round r message. (In the example in Figure 8-6, this corresponds to the fact that there is no round 5 bcast_i event.)

8.1.7 Message Delivery Guarantees

A *reliable* synchronous broadcast service should guarantee that a message is delivered in round r if and only if it was broadcast in round r. In a wireless network, the broadcast service may not be reliable: sometimes a message from round r may be lost. In fact, communication is prone to collisions when two nearby processes broadcast at the same time. As a result of a collision, each node can lose an arbitrary subset of messages that were broadcast in a round. Moreover, collisions may affect nodes in a non-uniform way: when a node broadcasts a message, some nodes may receive the message while others may not. To this point, we have formally described a general system and specified TIOA automata to model the various general system components. In this section, we discuss the precise guarantees related to which messages are delivered.

We now discuss the basic message delivery guarantees provided by the broadcast service: integrity, self-delivery, and per-round delivery. Then we discuss the collision detection properties that a general system guarantees, and prove two lemmas relating the completeness and accuracy of message delivery to the completeness and accuracy of the collision detector.

Integrity, Self-Delivery, and Per-Round Delivery

First, the broadcast service provides a basic integrity guarantee:

Lemma 8.1.35 (Integrity)**.** *For ever $r \in \mathbb{N}$, for every r-round execution α of a general system $\mathcal{G}$, for every port $i \in$ bcast-ports, if $\mathsf{recv}(M, \ldots)_i$ is a round r receive event in α, then for every $m \in M$, a round r $\mathsf{bcast}(m, \cdot)_j$ event precedes it for some port $j \in$ bcast-ports.*

Proof. This property is guaranteed by line 44, which ensures that each message delivered in round r has been stored in $M[r]$; line 5 ensures that each message stored in $M[r]$ was broadcast in round r. □

The broadcast service also guarantees that if node i broadcasts a message in round r, it receives its own message in round r (lines 45–46).

Lemma 8.1.36 (Self-delivery)**.** *For every $r \in \mathbb{N}$, for every r-round execution α of a general system $\mathcal{G}$, for every port $i \in$ bcast-ports, if a round r $\mathsf{bcast}(m, \cdot)_i$ event and a round r $\mathsf{recv}(M, \ldots)_i$ event occurs in α, then $m \in M$.*

Finally, as is typical in a synchronous broadcast service, the broadcast service delivers messages only once in each round to each non-failed port. Thus, it should not perform two $\mathsf{recv}(\ldots)_i$ events in the same round.

Lemma 8.1.37 (Per-round delivery)**.** *For any $r \in \mathbb{N}$, for any r-round execution α of general system $\mathcal{G}$, there is at most one round r $\mathsf{recv}(\ldots)_i$ event for each port $i \in$ bcast-ports; there is exactly one if i is non-failed in round r.*

Proof. First, we argue that there is at most one such receive event. The set *doneP* tracks the set of nodes that have already been delivered messages. In line 47, the broadcast service checks that a node has not already received messages for the current round. In line 58, the broadcast service adds i to *doneP*. Finally, in line 19, the broadcast service resets *doneP* to empty in preparation for the next round, and increments the round number. Therefore there is at most one round r $\mathsf{recv}(\ldots)_i$ event for each port $i \in$ *bcast-ports*.

Next, we argue that there is at least one such recv event: notice that the next-round action cannot execute until $i \in \textit{doneP} \cup \textit{failed}$, as per the precondition in line 15. Thus, if i is non-failed, the round is not incremented until a round r recv occurs. Since α is an r-round execution, there is a $r+1$ next-round event, and thus there is some round r $\mathsf{recv}(\ldots)_i$ event, for each port $i \in$ *bcast-ports*. □

In combination with Restriction 4 of processes, Lemma 8.1.37 implies that each non-failed port performs one broadcast in each round:

Lemma 8.1.38. *For every $r \in \mathbb{N}$, for every r-round execution α of a general system $\mathcal{G}$, for each port $i \in$ bcast-ports, there is at most one round r $\mathsf{bcast}(\ldots)_i$ for port i. Moreover, if the node associated with port i does not fail in or prior to round r, then there is exactly one such broadcast event.*

If the node associated with port i is correct, we can therefore refer to the unique round r bcast on port i and the unique round r recv on port i.

Collision Detector Properties

In this section, we relate the completeness and accuracy of the broadcast service to the completeness and accuracy of the collision detector. We begin by defining what it means for a process to detect a collision in a given round, and then present two basic lemmas regarding when a process *must* detect a collision, and when a process *may* detect a collision.

Detecting a Collision. Collision detection information is delivered by the broadcast service to each mobile node as part of the recv event:

Definition 8.1.39. *For every $r \in \mathbb{N}$, for every r-round execution α of a general system $\mathcal{G}$, we say that port $i \in$ bcast-ports* ***detects a collision in round*** *r when α contains a round r $\mathsf{recv}(\cdot, \pm, \cdot, \cdot)_i$ event.*

A collision detection in round r provides an indication that some message that node i should have received in round r was lost. It does not provide any information with respect to the number of lost messages or the identities of their senders.

Completeness. The broadcast service provides the following collision detection guarantees, which reflect the completeness and eventual accuracy guarantees of the collision detector:

Lemma 8.1.40 (Completeness)**.** *Assume that CD* $\in \Diamond\mathcal{A}$-$\mathcal{C}$*. For every* $r \in \mathbb{N}$*, for every* r*-round execution* α *of a general system* $\mathcal{G}$*, for* $i, j \in$ *bcast-ports be two broadcast ports:*

If there exists a round r $\mathsf{bcast}(m, \ldots)_j$ *event for* j *in* α *and a round* r $\mathsf{recv}(M, cd, \ldots)_i$ *event for* i *in* α*, and* i *and* j *are within distance* R *at the beginning of round* r[5] *and* j *does not fail prior to the end of round* r*, then either (1)* $m \in M$*, or (2)* $cd = \pm$*.*

Proof. Assume $m \notin M$. Consider lines 49 and 53 of the recv action: notice that $m \in sentM$, but $m \notin rcvedM$, hence $|sentM| < |rcvedM|$. Since *CD-rule* is complete, this implies that $cd = \pm$. □

That is, in each round, if two nodes are within the broadcast radius at the beginning of the round and one of them broadcasts a message, then either the other node will receive that message or it will detect a collision.

This assumption of "perfect" completeness is quite strong. Previously [23, 79], we have examined weaker notions of completeness, such as "majority completeness" and "zero completeness." It seems likely that all the algorithms described in this thesis can be adapted for a majority complete collision detector[6]. Lower bounds in [23, 79] suggest that it is likely impossible to adapt the algorithms in this thesis for zero complete collision detectors. Specifically, we know that with only a zero-complete, eventually accurate collision detector, it is impossible to solve consensus efficiently; every consensus protocol requires at least a logarithmic number of rounds. We can also conclude that it would be impossible to adapt the algorithms in this thesis for an "eventually complete" collision detector, as it is impossible to solve consensus with such a collision detector.

Accuracy. For eventual accuracy, we define the stabilization round $r_{acc}(\alpha)$, which is the round at which the collision detector becomes stable. From $r_{acc}(\alpha)$ onwards, the collision detector reports a collision only when a message was, in fact lost.

Definition 8.1.41. *For every infinite execution* α *of a general system* $\mathcal{G}$*, define* $\boldsymbol{r_{acc}(\alpha)}$ *to be the smallest round such that for every* $r \geq r_{acc}(\alpha)$*:*

- *If there exists a round* r $\mathsf{recv}(M, \pm, \ldots)_i$ *event in* α *for some* $i \in$ *bcast-ports, then there exists a port* $j \in$ *bcast-ports and* $m \in msgs$ *such that:*
  1. *There exists a round* r $\mathsf{bcast}(m, \ldots)_j$ *event in* α*.*
  2. *Port* j *is within distance* R' *of* i *at the beginning of round* r*.*
  3. *Message* $m \notin M$*.*

If no such $r_{acc}(\alpha)$ *exists, then we define* $r_{acc} = \infty$*.*

[5] *Recall that the velocity of a node is bounded by* v_{max}*, and hence this limits their distance from each other at the end of the round. In Section 8.2, we relate this maximum velocity to the broadcast radius.*

[6] In [23, 79] we have shown that the problem of consensus can be solved efficiently (i.e., in $O(1)$ time) if a system is equipped with a majority-complete, eventually accurate collision detector.

It then follows from the definition of eventual accuracy that such a stabilization round exists:

Lemma 8.1.42 (Eventual Accuracy)**.** *Assume that CD* $\in \Diamond\mathcal{A}$-$\mathcal{C}$*. For every infinite execution* α *of a general system* $\mathcal{G}$*,* $r_{acc}(\alpha)$ *is finite.*

Proof. According to the definition of accuracy, there is some r_{acc} such that for all $r \geq r_{acc}$, *CD-rule*$(\cdot, r, \cdot, \cdot, p, p) = \mathsf{null}$, for all $p \in \mathbb{N}_0$. Consider some round $r \geq r_{acc}$, and assume there exists a round r $\mathsf{recv}(M, \pm, \ldots)_i$ event in α for some $i \in$ *bcast-ports*. From the way in which *cd* is set in lines 51 and 53, we can conclude that $|sentMinterfere| < |rcvedMinterfere|$ in round r. Thus there exists some message that was broadcast in round r by some node j that is within distance R' of i at the beginning of round r that is not received by node i in round r, as required. Thus $r_{acc}(\alpha) = r_{acc}$. □

This lemma implies that, eventually, if a node detect a collision, then it did not receive some message sent by an interfering node.

8.1.8 Contention Manager Properties

The goal of the contention manager is to advise certain broadcast ports to be active, and other broadcast ports to be passive. The canonical contention manager—with the empty set of additional properties—defined in Section 8.1.4 simply guarantees that some advice is given in each round. We will be interested in this thesis in contention managers that provide further guarantees as to which ports are advised to be active. In this section, we begin by defining what it means for a port to contend in a given round, and then discuss desirable properties of contention managers.

Contending

When a port requests that a particular contention manager advise it to be active, we say that that port is "contending:"

Definition 8.1.43. *For round* $r \in \mathbb{N}$*, for an* r*-round execution* α *of general system* $\mathcal{G}$*, we say that port* $i \in$ *bcast-ports* ***contends for*** CM_d ***in round*** r *of* α *if* α *contains a round* $r - 1$ *broadcast event* $\mathsf{bcast}(\cdot, d)_i$*.*

Definition 8.1.44. *For round* $r \in \mathbb{N}$*, for an* r*-round execution* α *of general system* $\mathcal{G}$*, we say that port* $i \in$ *bcast-ports* ***contends in round*** r *of* α *if it contends for some* CM_d*,* $d \in$ *CM-names.*

The contention manager observes the broadcast events of the processes, and can thus determine which ports are contending in a given round.

Advising

Recall that the contention manager outputs advice as to whether a port i should be active or passive via $\mathsf{cm\text{-}advice}(i, cm)_d$ output events. This information is provided to the broadcast service, which then passes it to the broadcast ports along with the messages from the current round (line 54). We have already associated each cm-advice event with a particular round (see Definition 8.1.23). For example, a cm-advice event at time $r \cdot \mathit{RndLength}$ is providing advice for round $r + 2$ (which begins at time $(r + 1) \cdot \mathit{RndLength}$). When a contention manager produces a *cm-advice* event, we say that it is advising a port to be active or passive:

Definition 8.1.45. *For round $r \in \mathbb{N}$, for an r-round execution α of general system $\mathcal{G}$:*

- *We say that CM_d **advises** i **to be active** in round r when there is a round r $\mathsf{cm\text{-}advice}(i, cm)_d$ event where $cm = \mathsf{active}$.*
- *We say that CM_d **advises** i **to be passive** in round r when there is a round r $\mathsf{cm\text{-}advice}(i, cm)_d$ event where $cm = \mathsf{passive}$.*

If any contention manager advises a broadcast port to be active in a round, we say that the port is advised to be active:

Definition 8.1.46. *For round $r \in \mathbb{N}$, for an r-round execution α of general system $\mathcal{G}$:*

- *We say that broadcast port $i \in$ bcast-ports is **advised to be active** in round r if for some $d \in$ CM-names, CM_d advises i to be active in round r.*
- *Otherwise, we say that port i is **advised to be passive** in round r.*

Interference and Nearby Nodes

When discussing contention managers, it is often useful to consider the set of ports that may be near to a given location during a given round:

Definition 8.1.47. *For any $\ell \in (\mathbb{R} \times \mathbb{R})$, $r \in \mathbb{N}$, and $S \in$ part, the set $\mathbf{near}(\boldsymbol{\ell}, \boldsymbol{r}, \boldsymbol{S})$ is the set of ports $k \in S$ such that $|\mathit{location}[k] - \ell| \leq R/4$ at the beginning of round r.*

When discussing a range of rounds, we are interested in ports that are nearby in *all* the relevant rounds:

Definition 8.1.48. *For any $\ell \in (\mathbb{R} \times \mathbb{R})$, $r_1, r_2 \in \mathbb{N}$, and $S \in$ part, the set $\mathbf{near}(\boldsymbol{\ell}, [\boldsymbol{r_1}, \boldsymbol{r_2}], \boldsymbol{S})$ is the set of ports:*

$$\bigcap_{r \in [r_1, r_2]} \mathsf{near}(\ell, r, S) \ .$$

Notice these definitions refer to ports that are nearby at the beginning of one or more rounds. Recall that a port's location is potentially changing continuously, and maintained by the broadcast service in the *location* variable. However, the velocity

of a mobile node is bounded by v_{max}, and hence by assuming that two nodes are close at the beginning of a round, we can conclude that they are not too far apart at the end of a round.

We may also be interested in the nodes close enough to "interfere" with a given port's broadcasts.

Definition 8.1.49. *For any $\ell \in (\mathbb{R}\times\mathbb{R})$, $r \in \mathbb{N}$, and $S \in part$, the set* **interfere(ℓ, r, S)** *is the set of ports $k \in S$ such that $|location[k] - \ell| \leq 2R'$ at the beginning of round r.*

Non-Interfering and Fair Contention Managers

We begin by describing a powerful—and useful—class of contention managers that guarantee two strong properties: eventual non-interference and eventual fairness. Intuitively, a non-interfering contention manager guarantees that it will advise only one port in a region to be active; a fair contention guarantees that each port will get repeated turns to be active.

Non-Interference. Ideally, only broadcast ports that are sufficiently far apart should be activated by a contention manager. Specifically, a contention manager should ensure that, eventually, it never advises two ports withing distance $2R'$ to be active in the same round. With a non-interfering contention manager, all collisions can, eventually, be avoided simply by following the advice of the contention manager; this follows as a result of eventual collision freedom, which is discussed in Section 8.1.9.

Definition 8.1.50. *Assume $S \in part$, a set of ports, and that CM is a contention manager in some general system $\mathcal{G}$. We say that a contention manager* CM *is* ***eventually non-interfering*** *for S if it guarantees the following property:*

- *For every infinite execution α of $\mathcal{G}$, there exists some round r_{cm} such that for every round $r \geq r_{cm}$:*
  - *For every pair of broadcast ports $i, j \in S$, if i and j are advised by CM to be active in round r, then $i \notin \mathsf{interfere}(location[j], r, S)$.*

Fairness. A second desirable property of a contention manager is that it should eventually guarantee that each broadcast port is occasionally advised to be active. Consider the following notion of an "eventually fair" contention manager that gives each port repeated turns to broadcast:

Definition 8.1.51. *Assume $d \in \mathbb{N}$, $S \in part$, and that CM is a contention manager in some general system $\mathcal{G}$. We say that CM is* ***eventually fair*** *for S with delay d if it guarantees the following property:*

- *For every infinite execution α of $\mathcal{G}$, there exists some round r_{cm} such that for every round $r \geq r_{cm}$:*

- *If port $i \in S$ is not failed in rounds $[r, r+d]$, then there exists a round $r' \in [r, r+d]$ such that contention manager CM advises port i to be active in round r' in α.*

In this case, each non-failed port—that remains non-failed for sufficiently long—is guaranteed that every so often it will be designated as active. Notice that this contention manager does not distinguish between contending and non-contending ports; it simply gives each port turns. If the contention manager *CM* is also non-interfering, it guarantees that when the port is advised to be active, no other interfering port will be activated.

Contention Fairness. A weaker notion of fairness ensures that if a broadcast port contends for long enough, it will get a turn, i.e., it will get several consecutive rounds in which to broadcast. We therefore define a fairness property that is parameterized by two integers $\langle a, b \rangle$: each port that contends for a consecutive rounds (without failing) will be activated for b consecutive rounds. (Typically, b is much smaller than a.)

Definition 8.1.52. *Assume that CM is a contention manager in some general system $\mathcal{G}$, that $a, b \in \mathbb{N}$, $a > b$, and that $R_{distance} > 0 \in \mathbb{R}$ is a radius. We say that CM is* ***eventually*** *(a, b)-**contention fair** for set $S \in part$ with radius $R_{distance}$ if it guarantees the following property:*

- *For every infinite execution α of $\mathcal{G}$, there exists some round r_{cm} such that for all rounds $r \geq r_{cm}$:*
  - *If $j \in$ bcast-ports contends for CM in rounds $[r, r+a-1]$ and j is non-failed in rounds $[r, r+a-1]$, then there exists some round $r' \in [r, r+a-b-1]$ such that CM advises j to be active in rounds $[r', r'+b-1]$, and advises every non-failed port within distance $R_{distance}$ of j to be passive in rounds $[r', r'+b-1]$.*

Notice that in the preceding definition, since $r' \in [r, r+a-b-1]$, we can conclude that $r'+b \in [r, r+a-1]$, and hence that port j is contending and non-failed throughout the interval $[r', r'+b-1]$.

Regional Contention Manager

In this section, we consider another contention manager that guarantees only a "regional" notion of eventual fairness. A regional contention manager is responsible only for ports in a certain geographic region of the network. This contention manager appears easier to implement in real systems than a fair, or (a, b)-far contention manager, in that it only resolves contention among nodes in a single specific region, that is, nodes that can communicate with each other.

Let ℓ be a location in the plane. We begin by defining what it means for an execution to satisfy "ℓ-restricted contention." Conceptually, a regional contention manager is designed to reduce contention in some specific area of the network. Thus,

it will yield useful advice only when the contending ports are in that particular area of the network. Thus, we say that an execution satisfies "ℓ-restricted contention" when only ports near location ℓ contend:

Definition 8.1.53. *For some $\ell \in \mathbb{R} \times \mathbb{R}$, we say that an execution α of a general system $\mathcal{G}$ with contention manager CM satisfies **ℓ-restricted contention** for CM if for every round $r > 0$, for every set $S \in part$, for every port $j \in S$, port j contends for CM in round r only if $j \in \mathsf{near}(\ell, r-1, S)$.*

We now define a "regional" contention manager, which (informally) guarantees that eventually, if an execution satisfies ℓ-restricted contention, then whenever a port contends in a region, some port is advised to be active:

Definition 8.1.54. *Assume $\ell \in (\mathbb{R} \times \mathbb{R})$ and $S \in part$, and that CM is a contention manager in some general system $\mathcal{G}$. We say that CM is an **eventual ℓ-regionally fair** contention manager for S if it guarantees the following property:*

- *For every infinite execution α of general system $\mathcal{G}$, if α satisfies ℓ-restricted contention, then there exists some round r_{cm} such that for every round $r \geq r_{cm}$:*
  1. *At most one port is advised by CM to be active in round r.*
  2. *If some port $i \in \mathsf{near}(\ell, [r, r+1], S)$ contends for CM in round r, and port i does not fail prior to the beginning of round $r+1$, i.e., through the bcast_i event in round $r+1$, then there exists a port $j \in S$ such that:*
     (a) *Port j contends for CM in round r.*
     (b) *Port j does not fail prior to the beginning of round $r+1$, i.e., prior to the bcast_j event in round $r+1$.*
     (c) *Port $j \in \mathsf{near}(\ell, [r, r+1], S)$.*
     (d) *Contention manager CM advises port j to be active in rounds r and $r+1$.*

Thus, an ℓ-regionally fair contention manager guarantees that if an execution satisfies ℓ-restricted contention, then eventually it satisfies the following property: if some port near location ℓ contends for some round r, and does not fail for d subsequent rounds, then some contending port that is near to ℓ (though not necessarily the same one), and that does not fail for d rounds, is advised to be active in round r. At the same time, it guarantees that all other potentially interfering ports are advised to be passive in round r.

Notice that this *almost* guarantees that the contention manager is eventually non-interfering; however, it guarantees non-interference only when the execution satisfies ℓ-restricted contention, and only when some appropriate port is contending. As a result, an ℓ-regional contention manager reduces contention only among ports near to a location ℓ; some other method must be used to reduce contention among ports that are farther away.

Conservative Contention Managers

Finally, we will sometimes want to ensure that the contention managers advise a broadcast port to be active only if that port wants to be active. Notice that a fair contention manager is not "conservative" in that it activates ports in some partition S regardless of whether they contend or not.

Definition 8.1.55. *Assume that CM is a contention manager in some general system $\mathcal{G}$. We say that a contention manager CM is* ***conservative*** *if for every $r > 0$, for every r-round execution α of $\mathcal{G}$, CM advises a port to be active in round r only if it contends for CM in round r.*

8.1.9 Eventual Collision Freedom

To this point, we have no reliability guarantees on the broadcast service. In this section, we discuss a limited reliability guarantee: eventually, if only one message is sent by a neighbor of port i or i itself, then i receives that message. In fact, the guarantee is somewhat stronger in that it ensures that ports in different sets in the partition *part* can broadcast at the same time without interfering with each other.

We now define the basic eventual collision freedom property. This property says that if a port j broadcasts a message in round r, and no interfering port broadcasts a message in round r, then all nearby ports will receive the message:

Definition 8.1.56. *We say that an infinite execution α of a general system $\mathcal{G}$ satisfies* ***eventual collision freedom*** *if there exists a round r_{cf} such that for every round $r \geq r_{cf}$:*

For all $i, j \in$ bcast-ports, $S \in$ part, $m \in$ msgs:

- *If the following conditions hold:*
  1. Port j broadcasts a message: *A round r $\mathsf{bcast}(m, \ldots)_j$ occurs in α.*
  2. Port j is a neighbor of port i: $|location[j] - location[i]| \leq R$ *at the beginning of round r.*
  3. No interfering port broadcasts a message: *For every port $k \in \mathsf{interfere}(location[i], r, S)$, $k \neq j$, a round r $\mathsf{bcast}(\perp, \ldots)_k$ occurs in α.*
- *Then, if a round r $\mathsf{recv}(M, \ldots)_i$ occurs in α, $m \in M$.*

Notice that we only focus on the location of a node at the beginning of a round; since the maximum velocity of each node is bounded, this is sufficient to limit its location throughout the round.

We also introduce a slightly weaker version of eventual collision freedom, called "Eventual Collision Freedom with Good Advice" (ECF[GA]). The only difference is that ECF[GA] guarantees successful message delivery only when the broadcasting node complies with the advice of the contention manager; when the broadcasting node ignores the contention manager advice, there is no reliability guarantee.

Definition 8.1.57. *We say that an infinite execution α of general system $\mathcal{G}$ satisfies* ***eventual collision freedom with good advice*** *if there exists a round r_{cf} such that for every round $r \geq r_{cf}$:*

For all $i, j \in$ bcast-ports, $S \in$ part, $m \in$ msgs:

- *If the following conditions hold:*
  1. Port j broadcasts a message: *A round r $\mathsf{bcast}(m, \ldots)_j$ occurs in α.*
  2. Port j is a neighbor of port i: $j \in \mathsf{near}(location[i], r, S)$
  3. No interfering port broadcasts a message: *For every port $k \in \mathsf{interfere}(location[i], r, S)$, $k \neq j$, a round r $\mathsf{bcast}(\perp, \ldots)_k$ occurs in α.*
  4. Port j is advised to be active: *Port j is advised to be active in round r.*
- *Then, if a round r $\mathsf{recv}(M, \ldots)_i$ occurs in α, $m \in M$.*

We will focus our attention on executions of the basic system that guarantee eventual collision freedom. In Chapter 9, we will discuss a virtual infrastructure system in which every execution guarantees ECF[GA].

Notice that the eventual collision freedom property is not guaranteed by the BCast automaton. Since it is a property that holds "eventually," rather than perpetually, we restrict the set of executions of Bcast to satisfy Definition 8.1.56 or Definition 8.1.57.

8.1.10 A Note on Motion-controlled Devices

Throughout this section, we have defined a class of general systems in which the devices have no control over their own motion. In some practical situations, however, a mobile device may in fact be able to control its motion. For example, a mobile robot can direct its own motion in a suitable fashion; a vehicle may choose its own velocity; machines in a factory may have some autonomous control over their actions.

Moreover, some of the most compelling uses for the virtual infrastructure paradigm presented in this thesis are those in which devices may have some control over their motion. For example, consider a battlefield scenario where a set of tanks wish to coordinate their motion according to some predetermined formation; in this case, the tanks might use virtual nodes to coordinate their motion. (See [69] for an example of using virtual infrastructure to solve a basic coordination problem among mobile robots.)

The model as presented in Chapter 8, however, provides no mechanism for a process to influence the location of a node, as maintained by the broadcast service. It is relatively straightforward to augment the general system model presented here to include motion control. We briefly list the places in the formal model that must be modified.

- *Process definition:* First, it is necessary to augment the definition of a process (Definition 8.1.2) to include an output action, such as $\mathsf{move}(\ldots)$, which produces motion control signals. This information may be in the form of a velocity

vector, a target waypoint, or any other desired form of motion control signal. In practice, the most common such signal is acceleration.

- *Broadcast service signature:* Next, the broadcast service, which maintains the location information, must be modified to receive the move events as inputs, and update the location continuously in response. For example, if the move action outputs a velocity vector, then the broadcast service should maintain the current velocity of each node, and update the location correctly as a function of time.

- *Broadcast service behavior:* A final complication is that the moving entity is a mobile *node*, while the broadcast service maintains the location of *ports*; thus it is necessary to translate mobile node motion to port locations.

Everything else remains unchanged. Moreover, since the current model allows for any arbitrary motion, any algorithm for the general systems described herein should work correctly in a model containing motion control.

8.2 Basic Systems

In this section we define the basic wireless model used throughout Part II of this thesis, which we refer to as the "Basic System." The basic system models networks of physical devices communicating by wireless radio broadcast. Each device executes a process, the basic unit of computation. The nodes coordinate their communication using contention managers, and receive feedback on the success or failure of communication from collision detectors. An example instantiation of the basic system is illustrated in Figure 8-7.

Communication in the basic system proceeds in synchronous rounds. In each round, each node chooses whether to broadcast a message. This decision is based on the current state of the process, which may include information received from the contention managers in the previous round. Eventually, when contention is sufficiently low, communication is reliable: if only one message is sent by a neighboring node, then that message is received. When messages are lost, a collision detector provides (potentially unreliable) feedback that a problem occurred: the collision detector may indicate that a message that should have received was lost.

The basic system $BS(\mathit{process\text{-}ports}_B, \mathit{msgs}_B, \mathit{CM\text{-}names}_B, P_B, A_B, \mathit{CM}_B)$ is parameterized by six values: a set of ports, a message alphabet, a set of contention manager names, a set of processes, an algorithm, and a set of contention managers. (As a general convention, the subscript B refers to the basic system.) In this section, we define the basic system as a specific type of general system in which some of the parameters are fixed.

8.2.1 Parameter Definitions

We now fix for the remainder of Part II the following values: I_B, R_B, R'_B, and $\mathit{RndLength}_B$.

- The set of nodes:

  Definition 8.2.1. *Fix* $\boldsymbol{I_B}$ *as a finite, non-empty alphabet of names for the physical mobile nodes.*

- The broadcast and interference radii:

  Definition 8.2.2. *Fix* $\boldsymbol{R_B}$ *and* $\boldsymbol{R'_B}$ *where* $R_B \leq R'_B$ *and* $R_B \geq 20v_{max}$.

  Notice in particular the assumption that nodes cannot move too fast. Specifically, we assume a relationship between their maximum velocity and the broadcast radius R_B. Thus, in five rounds, for example, a node can most a distance of at most $R_B/4$.

- The round length:

  Definition 8.2.3. *Fix* $\boldsymbol{RndLength_B} = 1$.

  This normalizes the round length to 1.

- The collision detector:

  Definition 8.2.4. *Fix* $\boldsymbol{CD_B}$ *to be an eventually accurate, complete collision detector, that is, one in the class* $\Diamond\mathcal{A}$-$\mathcal{C}$.

8.2.2 Basic System Definition

In this section, we formally define the basic system as a specific type of general system:

Definition 8.2.5. *Given the following parameters:*

- $\textit{process-ports}_B$, *a non-empty set of port identifiers,*
- $\textit{msgs}_B$, *a non-empty set of messages,*
- $\textit{CM-names}_B$, *a finite, non-empty set of contention manager names,*
- P_B, *a finite, non-empty set of non-recoverable processes for:*

$$\langle I_B, \textit{process-ports}_B, \textit{msgs}_B, \textit{CM-names}_B \rangle \ ,$$

- A_B, *an algorithm for* $\langle I_B, P_B \rangle$,
- $CM_B = \{CM_d : d \in \textit{CM-names}\}_B$, *a finite, non-empty set of contention managers.*

Define the following terms:

- *Let* $\textit{bcast-ports}_B = I_B \times \textit{process-ports}_B$.

- *Let* $\boldsymbol{part_B} = \{bcast\text{-}ports_B\}$*, set containing a single set of all the broadcast ports in the basic system.*

The ***basic system*** $BS(process\text{-}ports_B, msgs_B, CM\text{-}names_B, P_B, A_B, CM_B)$ *is defined to be general system:*

$$G(I_B, process\text{-}ports_B, msgs_B, CM\text{-}names_B, P_B, A_B, R_B, R'_B, RndLength_B, part_B, CD_B, CM_B) .$$

Since all the processes in P_B are non-recoverable, we restrict our attention in this thesis to executions α in which for all $i \in I_B$, there is at most one fail_i event in α. Clearly, this is without loss of generality.

We also restrict our attention to executions of the basic system that satisfy eventual collision freedom (Definition 8.1.56).

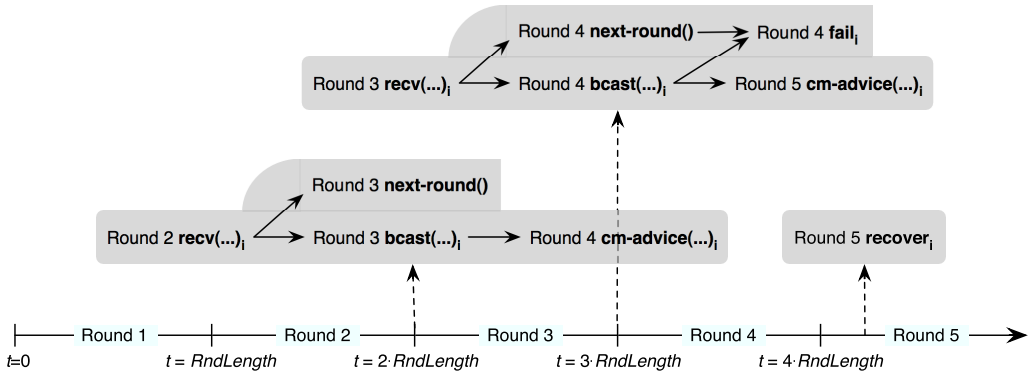

Figure 8-6: Example illustrating the timing of events in a round. The diagram depicts the first five rounds of an execution, and the dotted arrows indicate events that occur at three particular points in the execution: the end of round 2, the end of round 3, and during round 5. (Other intervening events are omitted.) The index i refers to an arbitrary port in *bcast-ports*, and the solid arrows indicate a partial ordering on events that occur at the same instant in time. For example, the round 2 recv event for node i necessarily precedes the round 3 bcast event for node i. Notice that round r lasts from time $(r-1) \cdot \mathit{RndLength}$ through time $r \cdot \mathit{RndLength}$. The bcast, recv, and cm-advice events occur at the round boundaries. For example, at the end of round 2 (i.e., at time $2 \cdot \mathit{RndLength}$), the following sequence of events occurs for each port $i \in \mathit{bcast\text{-}ports}$: a round 2 recv event for i, a round 3 bcast event for i, and a round 4 cm-advice event for i. The events occurs in this order (though not necessarily consecutively). The next-round events occur at the beginning of their respective rounds, and necessarily follow all the recv events of the preceding round. The fail$_i$ event at time $3 \cdot \mathit{RndLength}$ is classified as a round 4 fail event as it follows the round 3 next-round event. The recover$_i$ event in round 5 is classified as a round 5 recover event as it follows the round 5 next-round event (not depicted). Notice also that the round 4 bcast$_i$ event necessarily precedes the round 4 fail$_i$ event, as a failed process does not perform a bcast event.

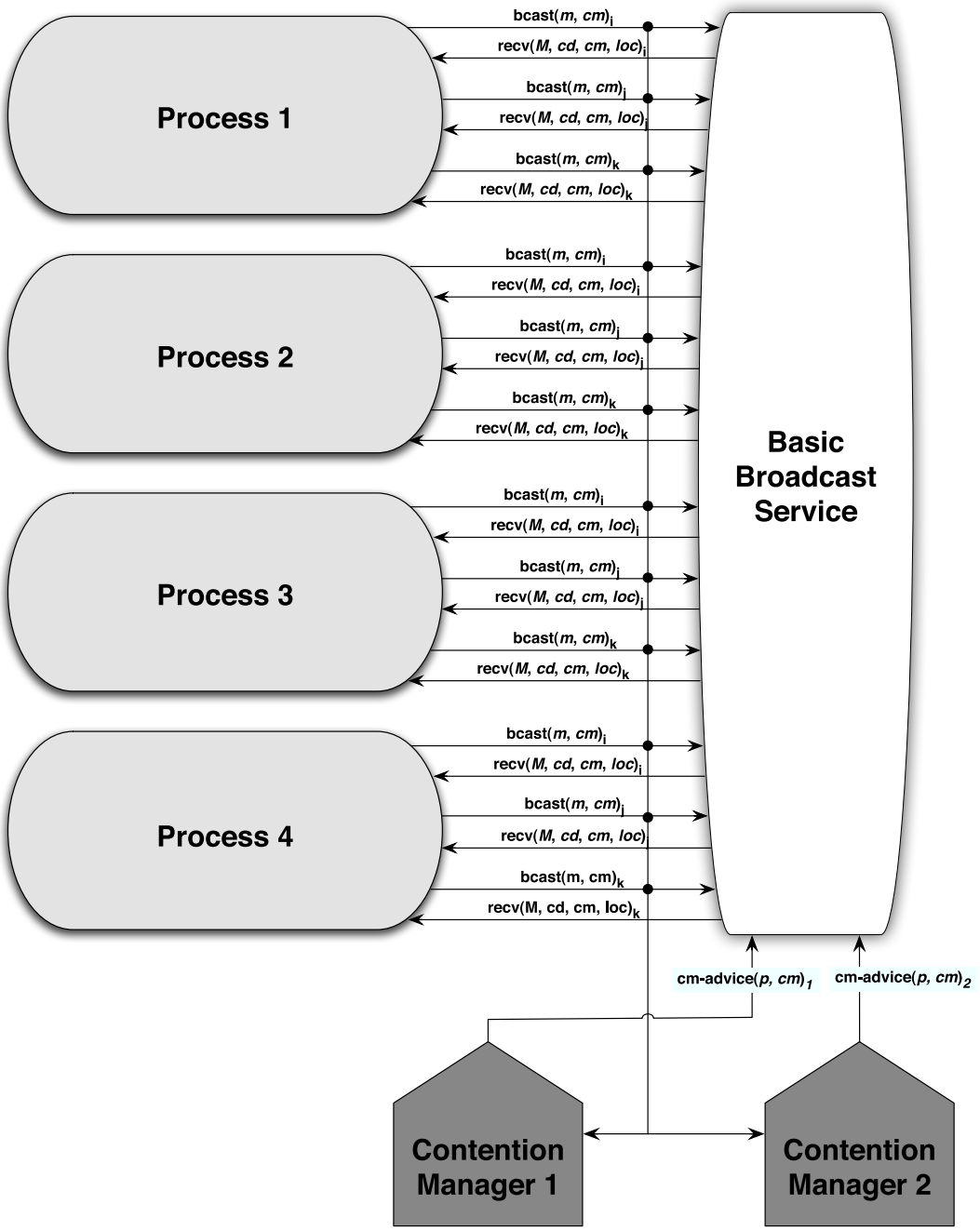

Figure 8-7: An example instantiation of the basic system, our model for wireless ad hoc networks. This example system consists of four processes, that is, $I = \{1, 2, 3, 4\}$. Each process accesses the basic broadcast service via a set of process ports. In this case, each process has three ports, that is, $\textit{process-ports}_B = \{1, 2, 3\}$. The basic system has two contention managers, that is, $\textit{CM-names} = \{1, 2\}$.

Chapter 9

Virtual Infrastructure Systems

In this chapter we describe a **virtual infrastructure system**, the virtual infrastructure abstraction being discussed in Part II of this thesis. A virtual infrastructure system consists of clients interacting with virtual nodes via a (virtual) broadcast channel. Virtual nodes are different from clients in two main ways. First, they are static: they remain in a fixed location throughout the execution. Second, virtual nodes are also recoverable: if they fail, they can resume operation at a later point, if circumstances allow. Our goal is that overall, virtual nodes should be more reliable and predictable.

In many ways, however, a virtual infrastructure system is quite similar to a basic system: each client and virtual node executes a process; the clients and virtual nodes communicate using a wireless broadcast service; the virtual nodes and clients receive advice from a contention manager on how to use the broadcast channel efficiently[1]; the clients and virtual nodes receive feedback from a collision detector when messages are lost.

An example virtual infrastructure system containing four clients and two virtual nodes is illustrated in Figure 9-1. It is particularly instructive to compare the system in Figure 9-1 to the basic system illustrated in Figure 8-7; notice that the resulting structure is quite similar, with the primary difference being the existence of virtual nodes.

We begin in Section 9.1 by defining a virtual infrastructure system. In Section 9.2, we present an example of how to design an application in the virtual infrastructure model. (In this case, we use the example of implementing a simple tracking service.) Finally, in Section 9.3, we discuss what it means for a protocol to implement virtual infrastructure.

[1]Notice that the contention managers in the virtual system can be used much like the contention managers in the basic system to decide when a node should broadcast and when a node should remain silent.

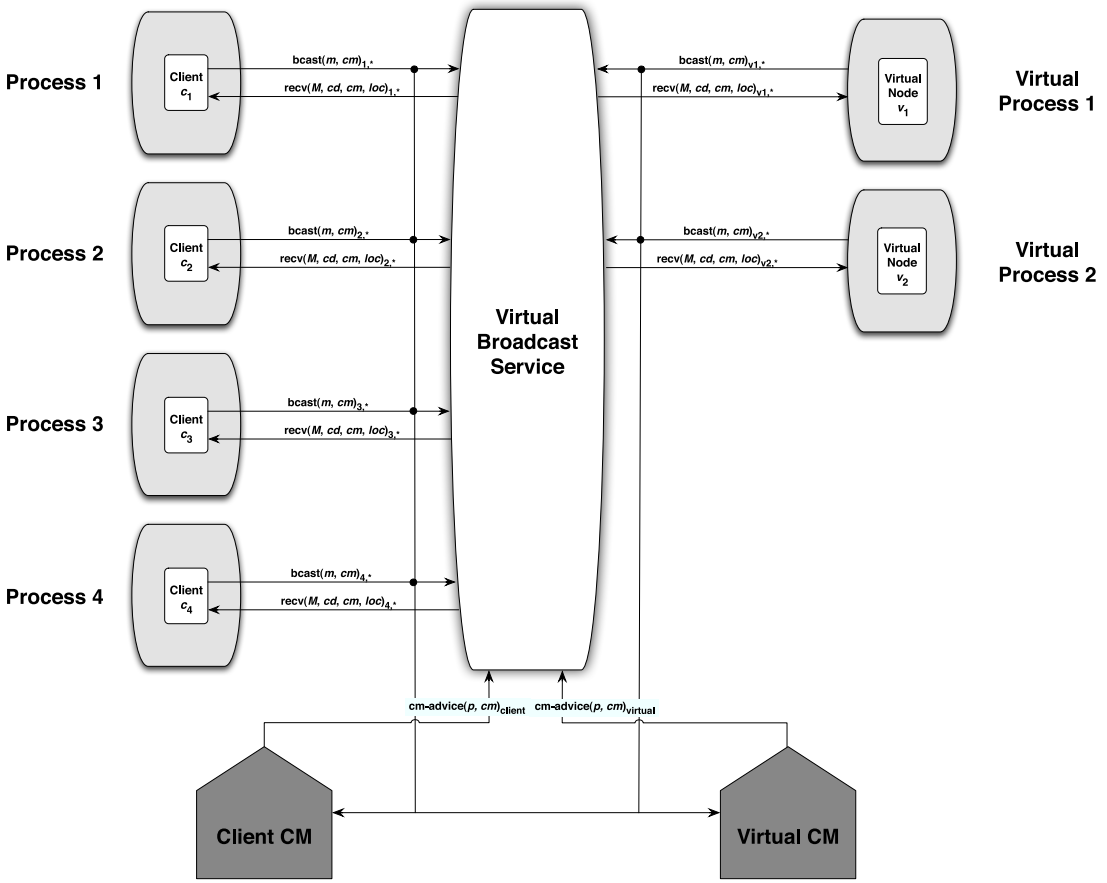

Figure 9-1: An example of a virtual infrastructure system. This example consists of four clients and two virtual nodes. In this case, $I_V = \{v1, v2\}$. Notice that each process has one port with which to access the broadcast service, that is, $\textit{process-ports}_V = \{*\}$. Also notice that there are two contention managers, that is, $\textit{CM-names}_V = \{\mathsf{client}, \mathsf{virtual}\}$. These contention managers give advice to the clients and to the virtual nodes.

9.1 Defining Virtual Infrastructure

The virtual infrastructure system $VS(I_V, msgs_V, P_V, A_V, loc_V)$ is parameterized by five values: a finite set of virtual node identifiers, a message alphabet, a set of processes, an algorithm, and a mapping designating the static location of each virtual node. (As a general convention, the subscript V refers to the virtual infrastructure system parameters.) In this section, we formally define the virtual infrastructure to be a type of general system in which some of the parameters are fixed.

9.1.1 Parameter Definitions

Recall that we have already fixed certain constants in Section 8.2: I_B, R_B, R'_B, and $RndLength_B$. In this section, we fix an additional set of values, those associated with the virtual infrastructure system, specifically, R_V, R'_V, $process\text{-}ports_V$, CD_V, $CM\text{-}names_V$, and CM_V.

- The broadcast and interference radii:

  Definition 9.1.1. *Fix $\boldsymbol{R_V}$ and $\boldsymbol{R'_V}$ where $R_V \leq R'_V$ such that:*

  - $R_V = R_B/2$
  - $R'_V = R_B/2 + R'_B$.

- The ports available to each node:

  Definition 9.1.2. *We fix $\boldsymbol{process\text{-}ports_V} = \{*\}$, that is, the set containing the single element $*$.*

- The collision detector CD_V:

  Definition 9.1.3. *Fix $\boldsymbol{CD_V}$ to be the set of all eventually accurate, complete collision detector rules.*

  Notice that, by definition, CD_V is in the class $\Diamond\mathcal{A}$-$\mathcal{C}$.

- The contention manager names:

  Definition 9.1.4. *Fix $\boldsymbol{CM\text{-}names_V} = \{\mathsf{virtual}, \mathsf{client}\}$.*

  Thus, the virtual infrastructure system contains two contention managers: one contention manager for the set of virtual nodes and one contention manager for the set of client nodes.

- The contention managers:

  Definition 9.1.5. *Fix $\boldsymbol{CM_{virtual}}$ to be the canonical contention manager automaton (see Figure 8-3) with two liveness properties: eventual non-interference, and eventual fairness for the set of ports $I_V \times \{*\}$.*

Definition 9.1.6. *Fix* $\boldsymbol{CM_{client}}$ *to be the canonical contention manager automaton (see Figure 8-3) with no additional liveness properties.*

Notice that we do not *a priori* assume any liveness guarantees for the client contention manager. There may, however, be a subset of executions in which the client contention manager satisfies additional liveness properties.

The virtual infrastructure implementation discussed in Chapter 10 has the property that if the basic system's contention managers satisfy a stronger liveness property, then the client contention manager also satisfies a stronger liveness property. As an example, we show that if a basic system contains a "global" contention manager guaranteeing (a, b)-contention fairness, then our implementation of virtual infrastructure guarantees that the client contention manager satisfies (a', b')-contention fairness for port set $I_B \times \{*\}$, where $a' = \lceil a/RndLength_V \rceil$ and $b' = \lfloor b/RndLength_V \rfloor$. More generally, we will show that if the basic system contains a "global" contention manager, then the client contention manager is a "sampling" of that contention manager, as per Definition 11.10.9 in Chapter 11.

9.1.2 Scheduling the Virtual Nodes

One of the advantages provided by virtual infrastructure is that virtual nodes are more predictable than real nodes; specifically, in Part II of this thesis, they are static, remaining at a fixed location throughout the execution. One use for this fact is that we can schedule the virtual nodes *a priori* so as to avoid collisions between nearby virtual nodes. (This schedule will be useful, for example, in constructing the virtual contention manager.) Notice that only virtual nodes are included in the schedule; clients, which move unpredictably, might be hard to schedule efficiently.

The length of the schedule will depend on the density of the virtual node locations: if there are many virtual nodes in a small area, then the schedule will need to be longer so that each virtual node gets a turn to broadcast.

Let $schedule[0..s-1]$ be an array in which each entry in the array is a set of virtual nodes in some arbitrary set I_V, and let loc be a function mapping virtual nodes in I_V to a location in $\mathbb{R} \times \mathbb{R}$. The integer s is the number of entries in the array. The schedule is "non-conflicting" with respect to I_V and loc if no two "neighboring" virtual nodes are scheduled to broadcast at the same time:

Definition 9.1.7. *We say that* $schedule[0..s-1]$ *is* ***non-conflicting*** *with respect to* I_V *and* loc *if it satisfies the following property:* $\forall i \in [0, s-1], \forall v, v' \in I_V$*, if* $v, v' \in schedule[i]$*, then* $|loc(v)_V - loc(v')_V| > 2R'_V$.

That is, if virtual nodes v and v' are in $schedule[i]$, for some i, then virtual nodes v and v' are not too close together. We say that the $schedule$ is "complete" with respect to I_V and loc if it includes each virtual node exactly once:

Definition 9.1.8. *We say that* $schedule[0..s-1]$ *is* ***complete*** *with respect to* I_V *if it satisfies the following property:* $\forall v \in I_V$*, there exists a unique* $i \in [0, s-1]$ *such that* $v \in schedule[i]$.

That is, every virtual node appears exactly once somewhere in the schedule. When discussing a schedule within the context of some particular virtual infrastructure in which I_V and *loc* are fixed parameters, we simply refer to the schedule as non-conflicting or complete, where I_V and *loc* are clear from context.

Definition 9.1.9. *We say that the* ***size*** *of schedule*$[0..s-1]$ *is* s*.*

Finding a complete, non-conflicting schedule is relatively straightforward: consider the graph that contains nodes I_V and an edge between two nodes v and v' if v and v' are within distance $2R'_V$; any coloring of this graph is exactly a non-conflicting, complete schedule. Assume, for example, that we have a coloring of the graph with s colors. Then we define a schedule as follows: for $i \in [0, s-1]$, *schedule*$[i]$ contains all the nodes $v \in I_V$ of color i. If the degree of the graph is Δ, then finding a $(\Delta+1)$-coloring is straightforward. Such colorings can be found in a distributed manner using, for example, techniques found in [3, 67].

9.1.3 Formal Definition

In this section, we formally define the virtual infrastructure system as a specific type of general system:

Definition 9.1.10. *Given the following parameters:*

- I_V*, a non-empty set of identifiers,*
- $msgs_V$*, a message alphabet,*
- P_V*, a non-empty set of processes for* $\langle I_B \cup I_V, process\text{-}ports_V, msgs_V, CM\text{-}names_V \rangle$*, where:*
  - *at least one process in* P_V *is recoverable and deterministic,*
  - *at least one process in* P_V *is non-recoverable,*
- A_V*, an algorithm for* $\langle I_B \cup I_V, P_V \rangle$ *where:*
  - *for all* $i \in I_B$*, process* $A_V(i)$ *is non-recoverable,*
  - *for all* $v \in I_V$*, process* $A_V(v)$ *is recoverable and deterministic,*
- $loc_V : I_V \to \mathbb{R} \times \mathbb{R}$*, a mapping from virtual nodes to locations.*

Define the following terms:

- *Let* $S_1 = I_B \times \{*\}$*.*
- *Let* $S_2 = I_V \times \{*\}$*.*
- *Let* $\boldsymbol{part_V} = \{S_1, S_2\}$*.*
- *Let schedule be a schedule that is complete with respect to* I_V *and non-conflicting with respect to* I_V *and* loc_V*. Let* SMAX *be the size of the schedule.*

- *Let* $\boldsymbol{RndLength_V} = SMAX + 10$.

The ***virtual infrastructure system*** $VS(I_V, msgs_V, P_V, A_V, loc_V)$ *is defined to be general system:*

$$G(I_B \cup I_V, process\text{-}ports_V, msgs_V, CM\text{-}names_V, P_V, A_V, R_V, R'_V, RndLength_V, part_V, CD_V, CM_V) ,$$

subject to the following additional assumptions:

- *The locations of the virtual nodes are constant: for* $v \in I_V$*, the location of* I_V *is equal to* $loc(v)_V$.

Notice that there are two specific restrictions made on virtual nodes that are not required of nodes in I_B: (1) Their location is static, unchanging throughout the execution, and (2) they are deterministic. Thus, given the current state of a virtual node, there is exactly one message that it is enabled to broadcast. We claim that this is not a severe restriction, as a non-deterministic process can be made deterministic by specifying a schedule that determines which transitions to make in which states.

Notice also that the virtual round length $RndLength_V$ is some positive integer value that depends on SMAX, and thus on the density of the virtual node locations: if there are many virtual nodes in a small area, then the round length will be longer; if the virtual nodes are sparsely distributed, then the round length will be shorter.

9.2 Example Virtual Infrastructure Application

In this section, we give a simple example of how to design an application for the virtual infrastructure system. We focus on the problem of tracking the locations of the clients (i.e., mobile nodes). Note that this example is only for didactic purposes—the result is not particularly efficient, nor does it guarantee desirable properties like locality of update. For a more careful location tracking system built on a (somewhat different) virtual infrastructure platform, see [35, 83][2]. The main goal of this example is simplicity: it demonstrates that a relatively powerful application can be designed with a minimum of effort.

The tracking service provides two actions with which it interacts with clients: query(j), an input which initiates a query for node $j \in I_B$, and respond(j, ℓ), an output which reports the most recent known location ℓ of node j.

The server component, running on the virtual nodes, supports two functions: a query, indicating that a client has received a query for the location of node $j \in I_B$, and update(j, ℓ), which notifies the service of the new location ℓ of node $j \in I_B$. The service fulfills these requests on a best-effort basis: when the location of a node is known, and if this location is not too out of date, it returns the value; when the location of a node is unknown, then it returns nothing. Under certain assumptions, we can conclude that when the system stabilizes, each query will receive a response.

[2]Specifically, they focused on a virtual infrastructure layer similar to that presented in Chapter 5, except that the virtual nodes are timed, rather than asynchronous. They also focus on self-stabilization as an important property for an infrastructural layer.

Figure 9-2: Virtual Infrastructure Example: Automaton VI

Signature:
Input:
recv(M, cd, cm, loc), $M \subseteq msgs_V$, $cd \in \{\pm, \text{null}\}$, $cm \in \{\text{active}, \text{passive}\}$, $loc \in (\mathbb{R} \times \mathbb{R})$

Output:
bcast($\langle$gossip, cl, $ct\rangle$, cm), cl an array of of locations, ct an array of integers, $cm \in \{\text{client}, \text{virtual}\}$
bcast($\langle$response, j, $\ell\rangle$, cm), $j \in I_B$, $\ell \in (\mathbb{R} \times \mathbb{R})$, $cm \in \{\text{client}, \text{virtual}\}$
bcast($\langle$query-error$\rangle$, cm), $cm \in \{\text{client}, \text{virtual}\}$
bcast($\perp$, cm), $cm \in \{\text{client}, \text{virtual}\}$

State:
query-set $\subseteq I_B$, the current set of active queries
client-loc, an array indexed by I_B of locations in $\mathbb{R} \times \mathbb{R}$
client-time, an array indexed by I_B of integers in $\mathbb{N}_0$, initially $[0, 0, \ldots, 0]$
cm-advice $\in \{\text{active}, \text{passive}\}$, the current advice of the contention manager
query-error $\in \{\textbf{true}, \textbf{false}\}$
turn $\in \{\text{gossip}, \text{query-response}\}$, the next action by the virtual node, initially, gossip
do-bcast $\in \{\textbf{true}, \textbf{false}\}$, the broadcast flag

Constants:
vn-timeout $\in \mathbb{N}$, the virtual node update timeout

Input recv(M, cd, cm, loc)
Effect:
  if $(M = \{\langle \text{update}, \ell, j\rangle\})$ **then**
    client-loc$[j] \leftarrow \ell$
    client-time$[j] \leftarrow$ *vn-timeout*
  else if $(M = \{\langle \text{query}, j\rangle\})$ **then**
    query-set $\leftarrow$ *query-set* $\cup\ j$
  else if $(M = \{\langle \text{gossip}, cl, ct\rangle\})$ **then**
    for *every* $j \in I_B$ **such that** *ct*$[j] >$ *client-time*$[j]$ *do*
      client-loc$[j] \leftarrow cl[j]$
      client-time$[j] \leftarrow ct[j]$
  for all $j \in I_B$ *do*
    if $(\textit{client-time}[j] > 0)$ **then**
      client-time$[j] \leftarrow$ *client-time*$[j] - 1$
  if $(cd = \pm)$ **then**
    query-error $\leftarrow$ **true**
  cm-advice $\leftarrow$ *cm*
  do-bcast $\leftarrow$ **true**

Output bcast($\perp$, virtual)
Precondition:
  turn $=$ query-resposne
  cm $=$ active
  query-error $=$ **false**
  $(\textit{query-set} = \emptyset)$ **or** $(\forall j \in \textit{query-set} : \textit{client-time}[j] = 0)$
  do-bcast $=$ **true**
Effect:
  turn $\leftarrow$ gossip
  do-bcast $\leftarrow$ **false**

Output bcast($\perp$, virtual)
Precondition:
  $(cm = \text{passive})$
  do-bcast $=$ **true**
Effect:
  do-bcast $\leftarrow$ **false**

Trajectories:
  stops when: $(\textit{do-bcast} = \textbf{true})$

Output bcast($\langle$gossip, cl, $ct\rangle$, virtual)
Precondition:
  turn $=$ gossip
  cm $=$ active
  cl $=$ *client-loc*
  ct $=$ *client-time*
  do-bcast $=$ **true**
Effect:
  turn $\leftarrow$ query-response
  do-bcast $\leftarrow$ **false**

Output bcast($\langle$response, j, $\ell\rangle$, virtual)
Precondition:
  turn $=$ query-response
  cm $=$ active
  $j \in$ *query-set*
  client-time$[j] > 0$
  $\ell =$ *client-loc*$[j]$
  do-bcast $=$ **true**
Effect:
  query-set $\leftarrow$ *query-set* $- \{j\}$
  turn $\leftarrow$ gossip
  do-bcast $\leftarrow$ **false**

Output bcast($\langle$query-error$\rangle$, virtual)
Precondition:
  turn $=$ query-response
  cm $=$ active
  query-error $=$ **true**
  do-bcast $=$ **true**
Effect:
  query-error $\leftarrow$ **false**
  do-bcast $\leftarrow$ **false**

Figure 9-3: Virtual Infrastructure Example: Automaton client(i), where $i \in I_B$

Signture:
Input:
  recv(M, cd, cm, loc)$_i$, $M \subseteq msgs_V$, $cd \in \{\pm, \text{null}\}$, $cm \in \{\text{active}, \text{passive}\}$, $loc \in (\mathbb{R} \times \mathbb{R})$

Input:
  query(j)$_i$, $j \in I_B$

Output:
  bcast($\langle$update, *current-location*, $j\rangle$, cm)$_i$, *current-location* $\in (\mathbb{R} \times \mathbb{R})$, $j \in I_B$, $cm \in \{\text{client},\text{virtual}\}$
  bcast($\langle$query, $j\rangle$, cm)$_i$, $j \in I_B$, $cm \in \{\text{client},\text{virtual}\}$
  bcast($\perp$, cm)$_i$, $cm \in \{\text{client}, \text{virtual}\}$
  response(j, ℓ)$_i$, $j \in I_B$, $\ell \in (\mathbb{R} \times \mathbb{R})$

State:
  sleep $\in \mathbb{N}_0$, how long until the next update should be sent.
  query $\in I_B \cup \perp$, a client whose location is being queried, or $\perp$.
  waiting $\in \{$**true**, **false**$\}$, an indication that the query is waiting for a response.
  cm-advice $\in \{\text{active}, \text{passive}\}$, the current advice of the contention manager.
  current-location $\in \mathbb{R} \times \mathbb{R}$, the current location of i.
  client-loc, an array indexed by I_B of locations in $\mathbb{R} \times \mathbb{R}$.
  do-bcast $\in \{$**true**, **false**$\}$, the broadcast flag.

Constants:
  client-timeout $\in \mathbb{N}$, the client update timeout

Input recv(M, cd, cm, loc)$_i$
Effect:
  if $(M = \{\langle \text{response}, query, \ell\rangle\})$ **then**
    client-loc[*query*] $\leftarrow \ell$
    waiting $\leftarrow$ **false**
  else if $(M = \{\langle \text{query-error}\rangle\})$ **then**
    waiting $\leftarrow$ **false**
  if $(sleep > 0)$ **then**
    sleep $\leftarrow sleep - 1$
  if $((cd = \pm)$ **and** $(waiting =$ **true**$))$ **then**
    waiting $\leftarrow$ **false**
  cm-advice $\leftarrow cm$
  current-location $\leftarrow loc$
  do-bcast $\leftarrow$ **true**

Input query(j)$_i$
Effect:
  query $\leftarrow j$
  client-loc[*query*] $\leftarrow \perp$
  waiting $\leftarrow$ **false**

Output bcast($\langle$update, *current-location*, $i\rangle$, cm)$_i$
Precondition:
  $sleep = 0$
  cm-advice $=$ active
  if $((query \neq \perp)$ **and** $(waiting =$ **false**$))$ **then**
    $cm =$ client
  else
    $cm = \perp$
  do-bcast $=$ **true**
Effect:
  sleep $\leftarrow$ *client-timeout*
  do-bcast $\leftarrow$ **false**

Trajectories:
  stops when: (*do-bcast* $=$ **true**)

Output bcast($\langle$query, $j\rangle$, $\perp$) $_i$
Precondition:
  $sleep \neq 0$
  cm-advice $=$ active
  $query = j$
  waiting $=$ **false**
  do-bcast $=$ **true**
Effect:
  waiting $\leftarrow$ **true**
  do-bcast $\leftarrow$ **false**

Output bcast($\perp$, cm)$_i$
Precondition:
  (cm-*advice* $=$ passive)
    or
  $((sleep \neq 0)$ **and** $((query = \perp)$ **or** $(waiting =$ **true**$)))$
  if $(waiting =$ **true**$)$ **or** $(sleep = 0)$ **then**
    $cm =$ client
  else
    $cm = \perp$
  do-bcast $=$ **true**
Effect:
  do-bcast $\leftarrow$ **false**

Output respond(j, ℓ)$_i$
Precondition:
  $j = query$
  client-loc[j] $= \ell$
Effect:
  client-loc[j] $\leftarrow \perp$
  waiting $\setminus ets$ **false**
  query $\leftarrow \perp$

Specifying the Virtual Infrastructure System

Before proceeding to discuss the protocol in more detail, we must specify the parameters for the virtual infrastructure system. We assume that the virtual nodes are deployed to form a grid. Assume, for ease of discussion in this section, that $R_V = 1$, that is, normalize the units of distance such that the broadcast radius in the virtual infrastructure system is 1. We then assume that $I_V = \{\langle x, y\rangle : x, y \in \mathbb{N}_0\}$, and that $loc(\langle x, y\rangle)_V = (x, y)$. That is, we assume a virtual node at each integer grid-point in the plane.

The processes are defined in Figures 9-2 and 9-3. The first, Figure 9-2, contains the pseudocode for the process that executes on each virtual node; the second, Figure 9-3, contains the pseudocode for the process that executes on each client. In this case, the clients are not anonymous: each has a unique identifier from the set I_B. Figure 9-3 defines a set of processes: $\{\mathsf{client}(i) : i \in I_B\}$. By contrast, the virtual nodes all execute the same (identical) process. We define the set of processes:

$$P_V = \{\mathsf{client}(i) : i \in I_B\} \cup \{\mathsf{VI}\} \ .$$

We define the algorithm $A(i)_V$, for $i \in I_B \cup I_V$, as follows:

$$A(i)_V = \begin{cases} client(i) & i \in I_B, \\ VI & i \in I_V \ . \end{cases}$$

Finally, we define the message alphabet $msgs_V$ implicitly as the set of messages sent by the automata in Figures 9-2 and 9-3.

Overview of the Tracking Service

The basic idea behind this simple protocol is to use the virtual nodes to form on overlay network over which the current location of each node is distributed. Every so often, each client sends an update of its current location to the nearest virtual node. The information is then flooded through the virtual node overlay network, which keeps track of how recently the information has been updated. When a virtual node receives a query, it sends back the most recent known location of the specified client, or nothing, if no recent information is known.

The Virtual Node Process. In more detail, we first examine Figure 9-2, the process executed by each virtual node. Each virtual node maintains the following state: *query-set*, a set of queries awaiting responses; *client-loc*, an array tracking the most recent known location for each $j \in I_B$; an array *client-time* that tracks how many rounds remain before the location information for a node is too out of date; *cm-advice*, the last advice received from the contention manager; *query-error*, a flag that indicates whether there has been a collision; *turn*, a flag indicating whether to use the next round for flooding gossip through the overlay, or for responding to a query; and *do-bcast*, a broadcast flag the enforces the "immediate response" property of a process, that is, enforcing that each recv event is followed immediately by a bcast

event.

We now examine the behavior of the recv transition. A virtual node may receive three types of messages: update messages, query messages, and gossip messages.

- The update messages contain the most recent location of some node $j \in I_B$. The new location is stored in *client-loc*$[j]$, and the round counter in *client-time*$[j]$ is reset to *vn-timeout*, the length of time the virtual node will consider the location information to be "recent" enough.

- Each query message contains a request for the location of some node $j \in I_B$. In this case, j is added to the query set.

- Finally, gossip messages contains the *client-loc* and *client-time* arrays of a neighboring virtual node. The virtual node integrates this information into its own *client-loc* and *client-time* arrays, taking for each node $j \in I_B$ the value that has a later deadline, i.e., a longer remaining expiration time.

Next, the recv transition decrements the time remaining in *client-time* for each client whose location is known. If a collision is detected, then the *query-error* flag is set, indicating that some query might have been lost. The contention manager advice is stored in *cm-advice*, and the broadcast flag *do-bcast* is set, leading to a bcast transition immediately following the completion of the recv transition.

We now describe the bcast transitions. The virtual node broadcasts three types of messages: gossip messages, response messages, and query-error messages. The virtual node alternates rounds in which it sends gossip messages and rounds in which it sends response/query-error messages, as controlled by the *turn* variable.

- Each gossip message includes the client-loc and the client-time arrays, which include the location and expiration times for some subset of the nodes.

- The response messages include the location of some particular node $j \in$ *query-set*, after which j is removed from the *query-set*. When the virtual node detects a collision, as recorded in the *query-error*, it sends a query-error message instead of the usual response message, indicating that there was a problem with the most recent query. This notifies the clients to rebroadcast any queries that were lost.

The remaining broadcast transition (lines 53–58) is enabled only when the other bcast transitions are disabled, and handles the cases where the virtual node chooses not to send a gossip or a response message.

The Client Processes. We next discuss the processes running on the clients (Figure 9-3) which provide an example of clients interacting with the tracking service, making requests and receiving responses. Clients maintain the following state: *sleep*, a count determining how long until the next location update should be sent to the tracking service; *query*, the ongoing query to the tracking service; *waiting*, a flag indicating that the client is awaiting a response from the service; *cm-advice*, the most recent advice from the contention manager; *current-location*, the current location of

node i; *client-loc* remembers the locations of the virtual nodes, as per the most recent queries; and *do-bcast*, a broadcast flag the enforces the "immediate response" property of a process, that is, enforcing that each recv event is followed immediately by a bcast event.

We now describe the recv transition. A client receives two types of messages from the server process running on the virtual nodes: response messages and query-error messages.

- The response messages indicate that a query is complete, returning the result of a location query. In this example, these responses are stored in the client-loc array.

- The query-error messages indicate that the virtual node has detected a collision; thus, if there is an ongoing query, it may not have received the query. Thus, in this case, the query is resent, as is indicated by setting *waiting* to false.

Next, the clients update the *sleep* counter, decrementing it by one. (The *sleep* counter controls the frequency with which the clients update the virtual nodes with their location.) Next, clients use their collision detectors to determine whether a response from a virtual node has been lost. When a client detects a collision, it resends the query to the virtual node, in case the virtual node sent a response that was lost due to a collision.

We now describe the bcast transitions. Clients broadcast two types of messages: update messages and query messages.

- Periodically, clients send update messages to the tracking service. When the variable *sleep* reaches zero, then it is time to send a new update. The maximum value of *sleep* is *client-timeout*, which determines the frequency of the periodic updates. Notice that messages are sent only when the contention manager designates the node as active.

- Every so often, a client may receive a query input, which sets *query* to some arbitrary node $j \in I_B$, and sets *waiting* to false, thus initiating a query. At this point, the client sends a query message to the server component of the tracking service. Notice that the query message is sent only in a round in which $sleep \neq 0$, that is, in a round that is not already designated for an update message, and also only if the contention manager designates the node as active.

The remaining bcast transition (lines 71–82) is enabled only when the other bcast transitions are disabled, and handles the cases where the client chooses not to send a query or an update message.

Performance of the Simple Tracking Application

Under certain assumptions, the tracking application guarantees that eventually, once the network stabilizes, each location query gets a response containing the approximate location of the queried node. We first enumerate the relevant assumptions. Notice

that to a large extent these numbers are chosen arbitrarily. It matters only that the *vn-timeout* is large enough with respect to the *client-timeout*, the diameter of the network, and the delays of the two contention managers.

- First, assume that the size of the network is bounded by some D (for diameter); specifically, assume that there are no mobile nodes outside of the square in the real plane defined by $\langle (0,0),(D,D)\rangle$. Thus, each message can travel from one end of the network to the other in time $\leq 2D$.
- Assume that the client contention manager guarantees eventual $(50D, 1)$-contention fairness (see Definition 8.1.52).
- Assume that $R'_V = 4$.
- Assume that the virtual contention manager is eventually fair with delay 25 (see Definition 8.1.51). (Notice that it is easy to construct a schedule for the virtual nodes of size 25, since the virtual node network is a grid, and $R'_V = 4$.)
- Assume that *client-timeout* $\leq 50D$.
- Assume that *vn-timeout* $> 200D$.
- Assume that none of the virtual nodes fail.

In this case, we can conclude that eventually, each query receives a response. First, notice that once the network stabilizes, we can conclude that *some* virtual node receives a location update from each node every $100D$ rounds. Specifically, the *client-timeout* ensures that after $50D$ rounds, the client begins to request access to the broadcast channel; since the channel is $(50D, 1)$-contention fair, we can conclude that within a further $50D$ rounds, the client receives authorization to broadcast with no interference. By eventual collision freedom, we can conclude that the virtual node receives this messages. Thus every $100D$ rounds, some virtual node receives a location update for each client.

Next, we notice that it takes a further $100D$ rounds for the information to disseminate through the network. Each virtual node broadcasts a gossip message at least once every 50 rounds: it receives a turn from the fair contention manager every 25 rounds, and it broadcasts a gossip message every two turns. Again, by eventual collision freedom we can conclude that this gossip message is received. Within $2D$ hops, the gossip message propagates throughout the network. Since each hop takes at most 50 rounds, we can conclude that within $100D$ rounds, the gossip has spread throughout the network.

Thus we conclude that every $200D$ rounds, every virtual node receives an update for each client's location. Since the *vn-timeout* $> 200D$, this is sufficient to ensure that each virtual node maintains a valid estimate of each client's location.

The overall delay for a query of the tracking service depends on how many concurrent queries occur: recall that the virtual node only responds to one query in a round in which it sends a response. (A simple optimization, of course, would allow

the virtual node to send multiple query responses in one round.) Assume there are no concurrent queries; a query will then complete after stabilization in $50D + 50$ rounds: within $50D$ rounds, the client is authorized by the contention manager to broadcast its query; within a further 50 rounds, the virtual node sends a response. Assuming there are at most k concurrent queries, the delay is $50D + 50k$ rounds.

Finally, notice that given a bound on the velocity of a client, we can calculate the accuracy of the location information. Specifically, we know that $|location[i] - query(i)| \leq 200D \cdot v_{max}$.

9.3 Simulating Virtual Infrastructure

Our goal in building virtual infrastructure is to emulate a virtual system using a basic system. Thus, given values for the parameters which specify a virtual infrastructure system—$I_V, P_V, A_V, loc_V, msgs_V$—we want to construct a basic system—$process\text{-}ports_B, msgs_B, CM\text{-}names_B, P_B, A_B, CM_B$—that is, in a sense, indistinguishable from the virtual infrastructure system. For the remainder of this thesis, we fix the set of virtual infrastructure parameters: I_V, $msgs_V$, P_V, A_V, loc_V, and we fix a specific instantiation of the virtual infrastructure system $VS(I_V, msgs_V, P_V, A_V, loc_V)$. In this section, we discuss what it means to correctly simulate the virtual infrastructure system in a basic system.

First, recall that, informally, a virtual infrastructure system consists of clients and virtual nodes communicating using a virtual broadcast channel. Formally, we refer to the set of processes running on nodes in I_B as "clients":

Definition 9.3.1. *Given a virtual infrastructure system, the set of* ***client processes*** *is* $\{A_V(i) : i \in I_B\}$.

We refer to the set of processes running on virtual nodes as "virtual processes:"

Definition 9.3.2. *Given a virtual infrastructure system, the set of* ***virtual processes*** *is* $\{A_V(w) : w \in I_V\}$.

Our goal in simulating virtual infrastructure is to ensure that client processes believe that they are operating in a virtual infrastructure system. Each client broadcasts and receives messages as if the system contains actual virtual nodes; an emulator acts as an intermediary between the client and the basic broadcast service, providing the illusion that virtual nodes really exist. Figure 9-4 depicts the situation in which each client interacts with an emulator (depicted as A_i in the figure), and together the clients and emulators form processes that communicate using the basic broadcast service. In Sections 10.1 and 10.2, we present an "emulator" of this sort.

The emulator should guarantee a "trace inclusion" property for the clients: for every "trace" of the clients in the basic system, there should exist an execution of the virtual infrastructure system with the same "trace."

We now discuss the notion of trace inclusion in more detail. Given an execution of a virtual infrastructure system $\mathcal{G}_V$, consider the events observable by the clients with the real times at which they occur. In other words, consider a timed execution of

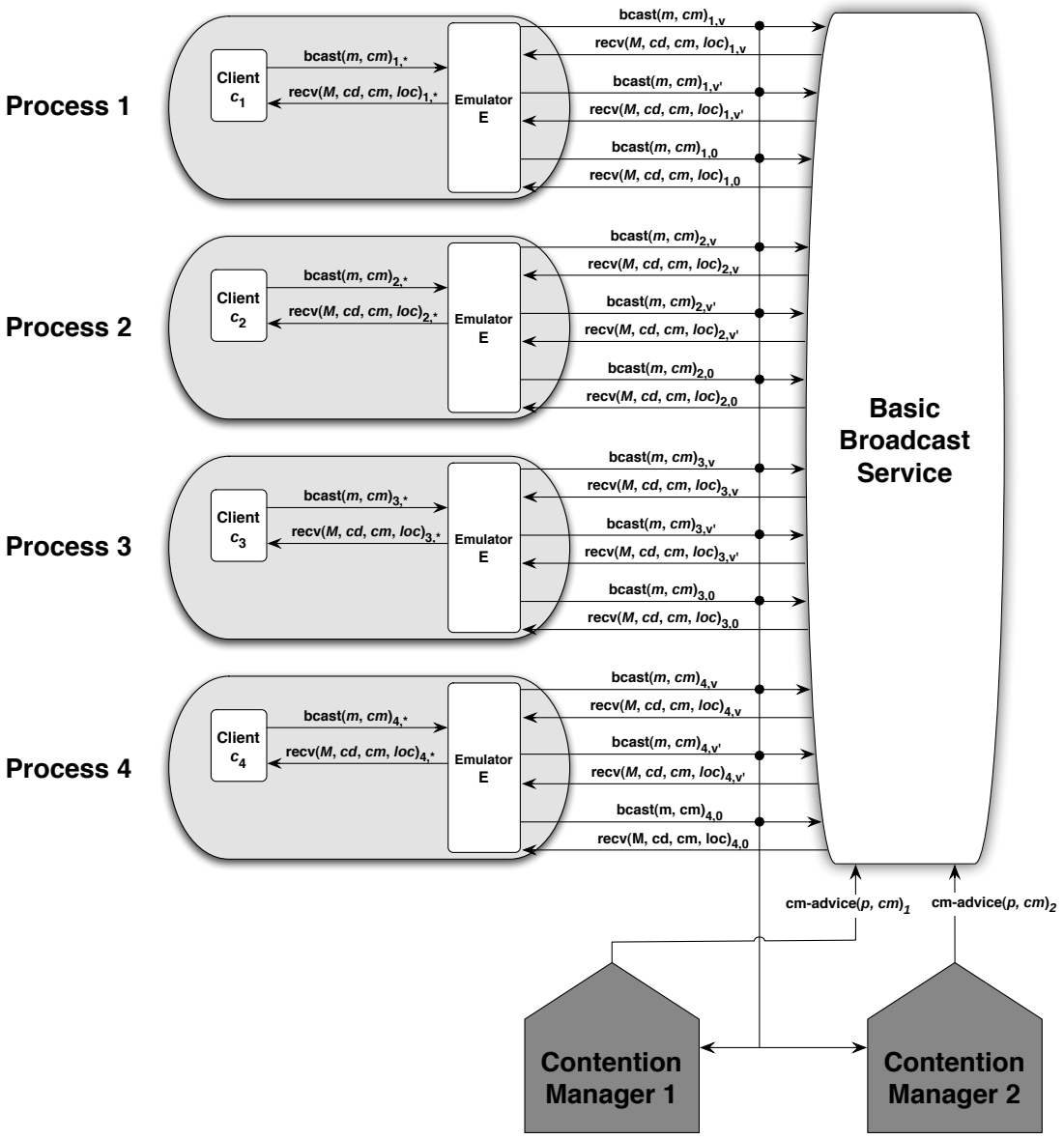

Figure 9-4: An overview of the emulator system. Emulator E is an algorithm that implements the virtual infrastructure system (depicted in Figure 9-1) on top of the basic system (depicted in Figure 8-7). Each client is composed with emulator E to form a process, and these processes are executed in the basic system. We say that emulator E implements the specified virtual infrastructure system if the clients cannot distinguish whether they are executing in the basic system, as in the figure above, or in the virtual infrastructure system (Figure 9-1). That is, emulator E implements the virtual infrastructure system when every trace of the clients in the basic system is a trace of the clients in the virtual infrastructure system.

the virtual system, consisting of clients, virtual nodes, the broadcast service, and the contention managers, in which all the events except the client broadcast and receive events are hidden. We define the set of hidden actions as follows:

$$H_V = \mathcal{G}_V.actions - \{\mathsf{bcast}_{i,*} : i \in I_B\} - \{\mathsf{recv}_{i,*} : i \in I_B\} \ .$$

Notationally, we refer to the set of externally-visible, timed client traces as the following: $traces(hide_{H_V}(\mathcal{G}_V))$. Thus, the only actions visible in the trace are $\mathsf{bcast}_{i,*} : i \in I_B$ and $\mathsf{recv}_{i,*} : i \in I_B$.

Similarly, given an execution of the basic system $\mathcal{G}_B$, consider the events observable by the clients with the real times at which they occur. In this case, we are considering a timed execution of the clients, the emulator automata, the broadcast service, and the contention managers, with all events except the client broadcast and receive events hidden. We define the set of hidden actions as follows:

$$H_B = \mathcal{G}_B.actions - \{\mathsf{bcast}_{i,*} : i \in I_B\} - \{\mathsf{recv}_{i,*} : i \in I_B\} \ .$$

Notationally, we refer to the set of externally-visible, timed client traces as the following: $traces(hide_{H_B}(\mathcal{G}_B)$.

When we say that the clients cannot distinguish between the basic system and the virtual infrastructure system, we mean that every event observed in the basic system is consistent with an execution of the virtual infrastructure system. Formally, then, we want to show that:

$$traces(hide_{H_B}(\mathcal{G}_B) \subseteq traces(hide_{H_V}(\mathcal{G}_V)) \ .$$

Finally, we want our emulator to have certain good properties. For example, a legal behavior in the virtual infrastructure system is for all the virtual nodes to fail at time 0 and never recover. We will show in Chapter 11 under what conditions our emulator guarantees virtual nodes to be alive. We also want the emulator to send small messages and introduce limited communication overhead. We discuss these desiderata in more detail when presenting the emulator algorithm.

Chapter 10

Emulating Virtual Infrastructure

In this section we present an algorithm that implements a virtual infrastructure system. We begin in Section 10.1 with an overview of the basic algorithm. In Section 10.2, then, we present a detailed description of the emulator protocol. The pseudocode for the algorithm is presented in Figure 10-2–13.

10.1 Algorithm Overview

In order to emulate a single virtual node at location ℓ, we replicate the virtual node at every device within distance $R_B/4$ of location ℓ.[1] By replicating the data at each of the participating nodes, we guarantee robustness in a dynamic, fault-prone environment. As mobile nodes join and leave the system, and as they enter and leave the relevant region near to ℓ, they participate—and cease participating—in the emulation. We refer to mobile nodes that are active in the simulation of some virtual node v—along with the emulator processes that execute on them—as *replicas* for v.

The key difficulty in emulating some particular virtual node v is maintaining consistency among the replicas of v. In an ideal world, the replicas responsible for emulating virtual node v would maintain identical copies of the virtual node's state. When the execution begins, this is trivially accomplished: each replica is initialized with the initial state of virtual node v. As the execution progresses and the virtual node sends and receives messages, the replicas must remain consistent.

Recall that in each (virtual) round, clients broadcast messages to the virtual node, and the virtual node broadcasts messages to the clients. If all the replicas receive the same set of messages in a (virtual) round, then they can maintain identical copies of the virtual node's state: since the virtual node is deterministic, if all the replicas receive the same messages in a round, each replica can process the messages and consistently update the replicated state. (In Part I of this thesis, we presented a strategy for implementing virtual infrastructure that operates along these lines.) Unfortunately, as has been discussed in Part II of this thesis, the wireless broadcast

[1] In the terminology from Part I, we would refer to the circle of radius $R_B/4$ around location ℓ as the focal point region.

channel is unreliable. As a result, we cannot be sure that replicas receive the same messages in a round: some replicas may receive a message, others may not.

10.1.1 Consensus and the Impossibility of Perfect Consistency

One way to maintain consistency among the replicas of a virtual node would be to run a *consensus* protocol for each round of the virtual node; the replicas could then agree on which messages the virtual node receives (and sends) in each (virtual) round. In the same way, they could agree on the other parameters that are part of a recv action, i.e., the collision detection information *cd* and contention management information *cm* for the virtual node. By repeatedly running consensus, the replicas could emulate a sequence of rounds. Moreover, as was described in [23, 79][2], there is a very efficient algorithm for consensus when collision detectors guarantee (perfect) completeness and eventual accuracy. In fact, each instance of consensus terminates in 3 rounds once the network has stabilized, that is, after eventual collision freedom (round r_{cf}), eventual accuracy (round r_{acc}), and eventually good contention management (round r_{cm}).

Unfortunately, the consensus protocol is efficient *only* after the network has stabilized. Thus, the virtual infrastructure will behave poorly prior to stabilization since the length of a virtual round depends on the length of the consensus protocol. Specifically, the length of a (virtual) round would vary unpredictably. (Recall in Chapter 9, we present a virtual infrastructure in which virtual rounds have a constant length.)

This unpredictability would have many negative consequences. First, it would complicate the synchronization among virtual nodes in the virtual infrastructure layer. If the (virtual) rounds have a fixed length, it is easy to ensure that all virtual nodes are executing the same (virtual) round at the same time. On the other hand, if virtual rounds have an unpredictable length, then some virtual nodes may execute much more rapidly than others, leading to added complexity in programming the virtual infrastructure. Instead of emulating a synchronous abstraction, the result would be an asynchronous (virtual) network, possibly with some variety of eventual synchrony guarantees.

A second negative consequence would be the difficulty in implementing timing-dependent algorithms. Many applications for virtual infrastructure involve interactions with the real world: for example, coordinating mobile nodes, collecting sensor data, or controlling tiny actuators. These real-world tasks are highly-timing dependent, and often have real-time deadlines. If (virtual) round lengths are unpredictable, it is much more difficult (and sometimes impossible) to implement such algorithms. For example, if a virtual node is attempting to collect data from a sensor, and needs to correlate the stream of sensor data with other virtual nodes, it is important that the sensor readings be taken at appropriate intervals of time. If some sensor readings cannot be measured (due to collisions and other network instability), it is often better to record the fact that the measurement failed, rather than sampling the sensor at

[2] In [23], see Algorithm 1, and a performance evaluation in Figure 1(a). In [79], see Algorithm 1, Section 7.1.

the wrong time. Similarly, in the case of coordinating actuators, it is often better to respond to instability (perhaps by performing some reasonable default action), than to simply wait for the network to stabilize.

A natural question, then, is whether some improved consensus algorithm could guarantee constant-round termination even during the unstable parts of the execution. Unfortunately, as was shown in [23, 79], it is impossible in the wireless setting to achieve consensus in a constant number of communication rounds while collisions continue to occur. Yet in order to implement a virtual infrastructure as described in Chapter 9, we require every simulated round to terminate in a constant number of communication rounds. That is, we want the $RndLength_V$ for the virtual broadcast service to be constant. Thus, using consensus as a black-box building block is infeasible. Moreover, the protocol presented in this chapter implements each virtual round in a constant number of basic rounds (specifically, $\mathsf{SMAX} + 10$ basic rounds); thus we can conclude that the protocol will not be able to guarantee that the replicas agree on the state of a virtual node at the end of each virtual round.

10.1.2 Agreeing on Virtual Node Executions

To overcome this seeming impossibility, we consider a weaker notion of consistency. Instead of trying to ensure consistency at the end of each virtual round, we allow some disagreement among the replicas, particularly during periods when the network is unreliable. We guarantee consistency only when the network is stable. This weakened guarantee is sufficient to build a virtual infrastructure that appears consistent from the clients' perspective, as the clients receive messages from the virtual nodes only when the network is stable. This realization is one of the key insights in circumventing the seeming impossibility.

In Section 10.1.1, we discussed a strategy in which the replicas attempt in each round to agree on a set of messages to receive in that round. The algorithm presented in this chapter uses a slightly different approach. Instead of agreeing only on a set of messages in each round, the replicas for each virtual node attempt to agree on a (finite) execution of the virtual node. (Recall from Appendix A that an execution is simply a sequence of trajectories and actions, satisfying appropriate properties.) Moreover, our goal is to design an agreement protocol that uses only a bounded number of rounds in the emulation of each virtual round. At the end of the agreement protocol—i.e., after some bounded number of rounds—a replica may be in one of three states: it may be certain that agreement has been achieved; it may be certain that agreement has *not* been achieved; or it may be *uncertain* as to whether or not agreement has been achieved. The key requirement is that the executions agreed upon in each round—when agreement occurs—converge to a single execution. (This property is captured by Lemma 11.7.19 in Chapter 11.)

There is one further aspect of the agreement protocol worth noting at a high level. Unlike a traditional consensus protocol which requires that some node broadcast a complete proposal, the agreement protocol does not require the replicas to send a proposal containing the entire execution. A proposal of this sort would, obviously, be much too large. Instead, the agreement protocol broadcasts information only on the

current (virtual) round, and a small amount of summary information regarding prior virtual rounds. This compressed proposal is an important technical tool which makes this protocol feasible, and the constant-sized protocol messages are a key contribution of this protocol.

Thus, the main technical challenge solved by the protocol in this paper is to agree sequentially on the outcome of each (virtual) round, that is, to agree on an execution, by performing a series of constant-round agreement instances, one for each round, while sending only constant-sized messages, and yet guaranteeing that eventually the replicas converge to a single execution that is consistent for all the replicas.

10.1.3 Outline of the Protocol

The emulation protocol for a single virtual node consists of eleven phases. Ten of these phases are implemented using a single basic round each; one of these phases is implemented using SMAX basic rounds, where SMAX is the size of a complete, non-conflicting schedule for the virtual nodes (see Definitions 9.1.7–9.1.9; recall that the schedule is only for virtual nodes). This results in a virtual round length of $RndLength_V = 10 + \mathsf{SMAX}$, as specified in Chapter 9.

We now briefly describe the eleven phases. For the purpose of this description, consider the case where the replicas are emulating some virtual round r for some virtual node v. In this case, the agreement protocol is trying to agree on an execution up until the end of round r, and at the same time is trying to ensure that all the replicas have enough information about round r for the use of future instances of the agreement protocol. The protocol divides into four sections: the message protocol (phases 1–2), the scheduled agreement instance (phases 3–5), the unscheduled agreement instance (phases 6–8), and the join protocol (phases 9–11):

The Message Protocol:

1. client: In the client phase, each client broadcasts its message for the current virtual round r.

2. vn: In the vn phase, emulators for virtual node v decide whether v should broadcast a message in virtual round r, and under certain conditions, broadcast that message. This decision depends on the outcome of the prior agreement protocols, the contention manager, etc.

The Scheduled Agreement Instance:

3. scheduled-ballot: This phase begins the first of two agreement instances that occur during virtual round r. Each replica participates in only one of the two agreement instances, depending on the *schedule*: if $v \in schedule[r \mod \mathsf{SMAX}]$[3],

[3]The attentive reader may notice that, since rounds start with 1, while the schedule begins with an index of 0, that the first round actually uses the second slot in the schedule. This anomaly introduces no problems, and we leave it for simplicity of notation.

then v's replicas participate in the scheduled-ballot instance; otherwise they participate in the unscheduled-ballot instance. In this phase, the replicas participating in the first agreement instance may broadcast a "ballot," which contains information about round r, along with some summary information about prior rounds.

4. scheduled-veto-1: This phase is the second round in the first agreement instance. In this phase, replicas for v broadcast veto messages if the agreement protocol appears to be failing.

5. scheduled-veto-2: This phase is the last round in the first agreement instance. In this phase, replicas for v again broadcast veto messages if the agreement protocol appears to be failing.

The Unscheduled Agreement Instance:

6. unscheduled-ballot: This phase begins the second of two agreement instances that occur during virtual round r. Replicas that did not participate in the scheduled agreement instance participate in the unscheduled agreement instance. That is, if $v \notin \mathit{schedule}[r \mod \mathsf{SMAX}]$, then v's replicas participate in the unscheduled instance. In this phase, the replicas participating in this agreement instance may broadcast a ballot. This phase is implemented by SMAX basic rounds. The replicas for virtual node v participate in only one of these basic rounds, however. Specifically, fix $r_b \in [0, \mathsf{SMAX} - 1$, where r_b identifies the current basic round within the unscheduled-ballot phase; if $v \in \mathit{schedule}[r_b]$, and if $v \notin \mathit{schedule}[r \mod \mathsf{SMAX} - 1]$, then the replicas for virtual node v participate in the r_b^{th} basic round in the unscheduled-ballot phase. During the other basic rounds in the unscheduled-ballot phase, the replicas for virtual node v do nothing.

7. unscheduled-veto-1: This phase is the second round in the second agreement instance. In this phase, replicas for v broadcast veto messages if the agreement protocol appears to be failing.

8. unscheduled-veto-2: This phase is the last round in the second agreement instance. In this phase, replicas for v again broadcast veto messages if the agreement protocol appears to be failing.

The Join Protocol:

9. join: This phase begins the join protocol. A replica for virtual node v may broadcast in this round if it wants to join the emulation.

10. join-ack: This phase is the second round of the join protocol. A replica that has already joined virtual node v may send a join response in this round, providing critical information to joining nodes.

11. join-veto: This phase is the last round of the join protocol. In this phase, a replica for virtual node v can determine whether the virtual node should be reset. A virtual node should be reset only when there are no replicas that have already completed the join protocol, which indicates that the virtual node has failed.

In Sections 10.1.5–10.1.7, we describe each of these phases in more detail. First, however, we discuss a critical aspect of the system: the implementation of the virtual contention managers.

10.1.4 The Virtual Contention Managers

Along with delivering messages and collision information, the virtual broadcast service also delivers contention management information to the clients and virtual nodes. Recall (from Chapter 9) that the virtual infrastructure system has two contention managers: a global contention manager for the clients, and a virtual contention manager for the virtual nodes. Implementing the client contention manager is straightforward, and will be discussed later: the broadcast service simply delivers to the clients the advice from one of the contention managers in the basic system (the "client contention manager"), sampling the advice from the basic contention manager once per virtual round—during the client phase.

The virtual contention manager, however, is emulated by the protocol presented in this chapter. Recall that the virtual contention manager should be fair and non-interfering: each virtual node should get a turn every so often to be active, and no two nearby virtual nodes should be active at the same time.

We build this contention manager using the *schedule* that we have already assumed: since the virtual nodes reside at known, pre-determined locations, we can explicitly schedule them in advance. Since the *schedule* is non-conflicting, the resulting contention manager is non-interfering. Since the *schedule* is complete, the resulting contention manager is fair.

Given such a schedule, we fix an ideal and fair contention manager as follows: virtual node v is advised to be active in virtual round r if and only if $v \in schedule[r \bmod \mathsf{SMAX}]$. It is easy to verify that the resulting contention manager has the desired properties. Moreover, it is easy for the emulator at each replica to calculate exactly what the advice should be for the virtual node, independent of the success—or failure—of communication with the other replicas.

The schedule determines which agreement instance replicas participate in: if v is "scheduled", then virtual node v is advised to be active in round r and the replicas for v use the scheduled agreement instance; if v is "unscheduled", then virtual node v is advised to be passive in round r and the replicas for v use the unscheduled agreement instance.

10.1.5 The Message Protocol

The purpose of the first two phases is for the clients and the replicas to broadcast the messages that are intended to be broadcast in a given virtual round, according

to the process automata. In the client phase of virtual round r (i.e., the first phase), each client broadcasts its message for round r, or nothing ($\perp$), if it has no round r message.

In the vn phase, each emulator for each virtual node v decides whether v should broadcast a message in round r. Throughout the execution, each emulator for v maintains a preferred execution for v. During good periods, when agreement holds, the replicas will all have the same preferred execution; during intervals when the network is unstable, the preferred executions may differ.

In order to decide whether to broadcast in the vn phase, a replica examines its preferred execution, and determines whether the virtual node is enabled to broadcast in the last state of that execution, and if so, what message it is enabled to broadcast. (Since the virtual node is deterministic, this is a deterministic choice based only on the state of the virtual node at the end of the preferred execution.)

A replica also examines two other factors related to contention management: (1) Is the replica itself advised by the basic system contention manager to be active in the current basic round? (2) Is the virtual node advised to be active in the current virtual round r?

If virtual node v is scheduled, a replica for v broadcasts a message m during the vn phase under the following conditions:

- The replica is advised to be active by a contention manager.
- The state of the virtual node, as calculated according to the replica's "preferred execution," is such that the virtual node is enabled to broadcast the message m.

If virtual node v is not scheduled, however, then the replica's decision as to whether to broadcast does not depend on the advice of the contention managers. That is, if the state of the virtual node, as calculated according to the replica's preferred execution," is such that the virtual node is enabled to broadcast the message m, then the replica broadcasts message m in the vn phase.

This rule may seem counterintuitive, as it ignores the contention manager when the virtual node is unscheduled[4]. This is almost guaranteed to result in a collision during the vn phase, which may seem undesirable and inefficient. However, this behavior is consistent with the ECF[GA] guarantee that the emulator is providing in its implementation of the virtual broadcast service: since the virtual node is enabled to broadcast a message, and since the virtual node is deterministic, the virtual node will broadcast a message in the current virtual round; since this violates the advice of the contention manager, the broadcast service is not required to deliver the message. Specifically, recall that the virtual broadcast service guarantees to deliver messages reliably only when virtual nodes comply with the advice of the contention manager.

[4]An alternate and seemingly simple solution would be to require nodes to follow the advice of the contention manager. However, it is often quite useful for protocols to sometimes ignore the contention manager. For example, in the protocol presented in this chapter, the contention manager is sometimes ignored—specifically in the veto rounds.

10.1.6 The Scheduled and Unscheduled Agreement Instances

The emulation protocol consists of two agreement instances: the scheduled and unscheduled agreement instances. As mentioned previously, the emulator for virtual node v participates in the scheduled agreement instance if v is scheduled for round r, and in the unscheduled agreement instance otherwise. In both instances of the agreement protocol, the primary purpose is to agree on which messages to receive, and whether or not to detect a collision. In the case of the scheduled agreement instance, an additional outcome is a decision as to whether or not the virtual node should broadcast a message. Finally, when agreement is reached, all incomplete agreement instances from previous rounds are resolved.

The agreement protocol is a modified version of the consensus protocol in [23, 79] that provides more information on the status of other replicas, and is amenable to smaller message sizes. The basic wireless consensus protocol in [23, 79] uses two rounds: a data round, in which the proposal is broadcast, and a veto round in which a negative acknowledgment is broadcast if anything went wrong in the data round. The agreement protocol here uses three rounds: a data round and two veto rounds.

In some ways, the agreement protocol can be seen as an implementation of "three-phase commit" (3PC), a classic protocol for atomic commit first introduced in [95,96]. In the language of three-phase commit, in each instance of the agreement protocol the nodes try to "commit" to that virtual round, or "abort" the round. When the protocol "commits" or "aborts" successfully, every participating node is aware of the result. On the other hand, in some instances, due to lost messages, the protocol may not "commit" or "abort." The result is that some nodes end the round in an intermediate state, unsure of whether the round will eventually be accepted or rejected. If there were only two phases (i.e., one veto phase), then it would not be possible to later reconstruct a consistent commit/abort decision for the round. The third phase allows a later "recovery" protocol to retroactively determine whether the round was accepted or rejected.

This analogy is not exact for several reasons. The main difference is that the agreement problem is somewhat different from classical atomic commit. First, the goal is to agree on some set of messages (and other information) for the round, rather than to simply commit or abort a transaction; in that sense, the agreement problem is more similar to consensus. Second, the goal is to agree on the behavior in a sequence of virtual rounds. Unlike in traditional three-phase commit, where the system is suspended to run the recovery protocol, in the agreement protocol presented here, the "recovery" protocol is integrated into the main protocol itself: virtual rounds continue to be emulated, even as prior virtual rounds are still being resolved. This leads to additional difficulties, since the result of earlier virtual round affects the outcome of later virtual rounds.

We now continue to describe the protocol in more detail. Each agreement instance begins with a ballot phase in which zero, one, or more, replicas broadcast a "proposal." A replica's proposal is in fact a compact representation of its preferred execution; we discuss later the details of this representation.

To gain additional information about the success or failure of the agreement in-

stance, the virtual node emulator uses *two* veto rounds, instead of one. (Thus, there are three phases total.) The status of each virtual round is determined by the success (or failure) of these veto rounds. Each replica designates the status of each virtual round by one of four colors: green, yellow, orange, or red.

- *Green:* A green round indicates that agreement has been achieved, and that the replicas have agreed on an execution of the virtual node. In this case, the virtual broadcast service may deliver messages to the clients. If an emulator designates a round as green, it can be certain that the round is "successful." (This is analogous to a "commit" in the case of atomic commit.)

- *Yellow:* A yellow round indicates that every replica at least received the current proposal, which is guaranteed to be unique, but replicas may still disagree on the execution. Even so, it is acceptable for a replica to include messages from the current round in its preferred execution (which may become the proposal in future rounds). If an emulator designates a round as yellow, it cannot determine whether or not the round is "successful."

- *Orange:* An orange round indicates that some replicas may have received the current proposal, but some may not have. When a replica designates a round as orange, it may not include messages from that round in its preferred execution, as some other replicas may not have received the proposal and thus may not have those messages. If these messages were used in a future proposal, it would be impossible to represent the proposal compactly. On the other hand, if a replica designates a round as orange, then it remains possible that some other replica may in the future propose an execution including messages from this round. If an emulator designates a round as orange, it cannot determine whether or not the round is "successful."

- *Red:* Finally, a red round indicates that the current agreement instance failed, and no messages from this round will be included in any future execution. In particular, some replicas—perhaps all—may not have received the current proposal. If an emulator designates a round as red, it can be certain the round is not "successful." (This is analogous to an "abort" in the case of atomic commit.)

Thus, the round color reflects each replica's local knowledge about *the other replicas'* knowledge of the emulated virtual rounds. The two veto phases, along with the ballot phase, determine the color designation of the round. If a replica receives exactly one ballot, no veto messages, and there are no collisions in any of the rounds, then the replica designates the virtual round as green. Conversely, if a replica does not receive exactly one ballot or detects a collision in the ballot round, then the replica designates the virtual round as red. Intermediate levels of success result in the yellow and orange designation: if there is a collision or veto in the first veto round, then the replica designates the round as orange; if there is a collision or veto in the second veto round, then the replica designates the round as yellow. This is summarized in Figure 10-1.

ballot	veto-1	veto-2	**Replica Color**
√	√	√	Green
√	√	X	Yellow
√	X	X	Orange
X	X	X	Red

Figure 10-1: Table indicating how a node responds to collisions in the different phases of the algorithm: ballot/veto-1/veto-2. A check (√) indicates that the node receives a message in that round, and no collisions or veto indications are received. An **X** indicates that the node does not correctly receive the message in that round. The *Color* column indicates the designated color for a round. Green/yellow rounds indicate (tentatively) that a message may be processed, while orange/red rounds indicate (tentatively) that the virtual node has detected a collision.

One of the key invariants of our protocol is that any two replicas disagree on the color designation by at most one shade. For example, if replica i designates a round as green, then every other replica must designate the round as green or yellow. Similarly, if replica i designates a round as red, then every other replica must designate the round as orange or red. This limited form of disagreement is key to the correctness of the emulation protocol.

A final issue related to the agreement instances is the composition of the ballots: one of the key properties of the protocol is that the ballot does not have to be too big, even though it is proposing agreement on some execution history. The ballot, then, includes two pieces of information: (1) information about the current virtual round, including any messages received in the client and vn phases, and any collision detected in the client and vn phases, and (2) a pointer to the most recent seemingly "good" round, that is, the most recent virtual round designated by the replica as green or yellow.[5]

When a set of replicas executes a sequence of such agreement instances, one for each virtual round, they can use this information to construct a preferred execution of the virtual node. In rounds that are "good," the preferred execution includes a recv event containing the messages specified by the ballot; in rounds that are "bad," the preferred execution includes a recv event containing an empty set of messages, and a collision. The replicas use the previous-round pointer in the ballots to determine which rounds are "good" and which rounds are "bad." These pointers create a "linked-list" of pointers in the ballot history, and when two replicas agree on this linked-list of pointers, they agree on their preferred execution.

[5]When considering analogies to three-phase commit, there are some similarities between the additional information included in the ballot, and the additional information introduced by Dolev and Keidar in their "Enhanced Three Phase Commit" protocol [47]. In their protocol, each node maintains a variable *last-elected*, which keeps track of the most recent recovery attempt that the node coordination, and *last-attempt*, the most recent recovery attempt in which the node attempted to commit or abort.

Scheduled Agreement Instance. We have already discussed above the overview of the agreement protocol. The scheduled agreement instance is a straightforward instantiation of this protocol: the scheduled-ballot phase implements the ballot phase, and the scheduled-veto-1 and scheduled-veto-2 phases implement the veto phases. An emulator for virtual node v participates in the scheduled agreement instance if v is scheduled for the current virtual round. It includes in its proposal the following elements: (1) all the messages received during the client and vn phases; (2) a flag indicating whether or not a collision is detected during the client phase; similarly, a flag indicating whether or not a collision is detected during the vn phase; and, (3) the previous-round pointer.

Moreover, since the schedule ensures that no two nearby virtual nodes are scheduled in the same virtual round, we can be sure that no nearby virtual nodes will interfere with the scheduled agreement instance. Thus, the only contention during the scheduled agreement instance arises from replicas of the virtual node in question, and these replicas use a regionally-fair contention manager to reduce their contention.

Emulators for unscheduled virtual nodes, however, listen passively to the scheduled agreement instance. Consider, for example, the case where v is an unscheduled virtual node, and v' is a scheduled virtual node. If v' decides to broadcast a message in the current virtual round, then (perhaps) some replica for v' may choose to broadcast that message during the vn phase and the scheduled-ballot phase. If virtual node v' is successful in broadcasting this message, then virtual node v, the unscheduled virtual node, should receive this message. Thus the replicas for v must listen attentively during the scheduled agreement instance to determine which virtual nodes are sending messages, and which virtual nodes are successful.

Thus, an emulator for an unscheduled virtual node v listens passively during the scheduled agreement instances, and maintains its own view on the color of the virtual round. It does not broadcast veto messages, however; it simply listens. If the emulator for the (unscheduled) virtual node v believes that the round is a green round for the (scheduled) virtual node v', it includes that message in its set of messages to receive in the round; otherwise, it omits it. By the same invariant discussed above regarding the color of the rounds, if the emulator for v believes that the round is green for v', then each emulator for v' has designated the round either green or yellow.

Unscheduled Agreement Instance. The unscheduled agreement instance uses SMAX+2 basic rounds. The unscheduled ballot phase uses SMAX rounds and is the only phase to use more than one basic round. This use of multiple rounds is necessary to avoid contention: unlike in the case of the scheduled agreement instance, without the use of the *schedule*, there is no guarantee that two nearby virtual nodes will not interfere with each other.

Of the SMAX basic rounds used to implement the ballot phase, each virtual node is assigned exactly one by the schedule. For basic round $r_b \in [0, \mathsf{SMAX} - 1]$ within the unscheduled-ballot phase, emulators for virtual node v use basic round r_b if $v \in schedule[r_b]$. Since the schedule is non-interfering, each virtual node can be assured that nearby virtual nodes will not interfere with its ballot phase.

All the virtual nodes share the same two veto phases. Thus when a virtual node fails to reach agreement, it may also interfere with its neighboring virtual nodes.

The ballot and veto phases then proceed as discussed above in the scheduled agreement instance. The ballot includes all the messages received and collisions detected in the client phase, along with the messages received in the vn phase. (In fact, the set of messages received is modified by the outcome of the scheduled ballot instance.) That is, as in the scheduled ballot instance, the replicas are agreeing on the set of messages to receive and whether or not to detect a collision.

10.1.7 The Join and Restart Protocols

The last three phases are dedicated to the join protocol, which allows new replicas to join in the emulation, and also allows replicas to reset the virtual node when it has failed.

In order to join a virtual node emulation, a new replica must synchronize with the existing replicas. A node chooses to join the virtual node emulation when it is within distance $R_B/4$ of the virtual node location. (In practice, a hysteresis zone is recommended, though not necessary, as it prevents frequent entrance and departure by nodes near the edge of a region.) In the join phase of the protocol, a node broadcasts a request to join the emulation. If a replica that has already joined receives a join request, and if the contention manager advises that replica to be active, then it broadcasts its current protocol state in the join-ack phase, thus giving the joining node an up-to-date snapshot of the current status of the emulation.

As before, we must contend with the interference between different replicas for nearby virtual nodes. Imagine that two physical nodes try to join two different virtual nodes in the same region. These two nodes may interfere with each other. We therefore use the schedule to mediate between the different virtual nodes, so that emulators for neighboring virtual nodes do not disrupt each other's join protocols, ensuring that each emulator receives an opportunity to join its virtual node.

Occasionally, a virtual node may fail when all the replicas leave the region. We want the virtual node to restart in a default initial state when some mobile nodes reenter the region. Moreover, we want to ensure that the replicas maintain a consistent state, despite the occasional restart.

The key difficulty here is ensuring that the virtual node really is uninhabited; if some replica is active, and a new mobile node attempts to "restart" the virtual node, then there is a risk that the virtual node's state is unnecessarily lost, and the possibility of inconsistency. By resetting a virtual node only if it is abandoned (i.e., failed), it is relatively easy to maintain a consistent state among the replicas.

We use the join-veto phase to detect a failed virtual node. Specifically, in the join-veto phase, each node that has already joined broadcasts a message. (This is in fact likely to cause a collision, if there is more than one participating replica.) A new joining node is then aware that it should not reset the virtual node. (One might consider an optimized version of the protocol that reduces the number of join-veto messages by requiring replicas to broadcast in the join-veto phase only when some node wants to perform a reset.)

One additional detail is needed: a reset can occur only in a round in which virtual node v is scheduled. This means that each virtual node has a chance to execute its join-veto round without interference from nearby virtual nodes and nearby join-veto messages. Thus, a functioning virtual node will not prevent a neighboring virtual node from being reset.

10.2 Virtual Infrastructure Emulator

In this section, we discuss the detailed implementation of the virtual infrastructure emulator. We first discuss the basic structure of the emulator, and its interface with clients and the broadcast service (see Section 10.2.1). We then discuss the state maintained by the virtual node emulator (see Section 10.2.2). Next, we detail the main algorithm itself (see Section 10.2.3). Then, we present some helper functions used by the algorithm (see Section 10.2.4), and an auxiliary component, the virtual node multiplexer (see Section 10.2.5). We conclude (Section 10.2.6) with the full specification of the emulator basic system, that is, we fix the parameters P_B, $\mathit{process\text{-}ports}_B$, msgs_B, A_B, $\mathit{CM\text{-}names}_B$, CM_B.

10.2.1 Virtual Infrastructure Emulator: Structure and Signature

Our goal is to build an emulator that acts as an intermediary between the clients and the broadcast service, and emulates a virtual infrastructure system. To this end, the main purpose of this section is to describe such an emulator automaton E.

In order to build a basic system that emulates a virtual infrastructure system, each client process is composed with emulator E, forming a new process that is executed in the basic system. (See Figure 9-4 for an illustration of the emulator automaton composed with a client to form a process.) Thus, the emulator provides a single broadcast/receive port to the client, and interacts with a set of broadcast/receive ports provided by the broadcast service $\mathcal{B}_B$. Recall that the port set of the virtual infrastructure system, $\mathit{process\text{-}ports}_V = \{*\}$; thus, each client has a broadcast port with the subscript $*$. The emulator's port to the client, then, has the same subscript, $*$.

The emulator itself is a composition of two types of a components: individual "virtual node emulators", each of which emulates one virtual node, and a "multiplexer" which connects the virtual node emulators to the client. (See Figure 10-3 for an illustration of the emulator components, their ports, and their interactions with the client.) We refer to the first automaton as $E(v)$, for $v \in I_V$, and the second as *Multiplexer*.

The main body of this section is presenting the emulator process(es). (If the virtual system being emulated is anonymous, there will be only one process.) Recall that a system is instantiated by remapping processes, disambiguating the automata based on the nodes on which they are running. Thus, the process specification does not include any reference to a particular node $i \in I_B$.

Figure 10-2: Automaton $E(v)$: Virtual Node Emulator — Signature and Data Structures

Signature:
Input:
recv($allM$, cd, cm, $loc)_v$, $allM \subseteq msgs_B$, $cd \in \{\pm, \mathsf{null}\}$, $cm \in \{\mathbf{true}, \mathbf{false}\}$, $loc \in \mathbb{R} \times \mathbb{R}$
fail()

Output:
vn-client-output($vnm\text{-}out$, $vnCD\text{-}out)_v$, $vnm\text{-}out \in \{\langle \mathsf{vn}, m, \ell \rangle : m \in msgs_V, \ell \in \mathbb{R} \times \mathbb{R}\}_\perp$, $vnCD \in \{\pm, \mathsf{null}\}$
bcast(m, $cm)_v$, $m \in msgs_B$, $cm \in CM\text{-}names_B \cup \{\perp\}$

State:
Emulator Data Structures:
$rnd \in \mathbb{N}_0$, initially 0
$phase \in \{$client, vn, scheduled-ballot, scheduled-veto-1, scheduled-veto-2, unscheduled-ballot, unscheduled-veto-1, unscheduled-veto-2, join, join-ack, join-veto$\}$, initially client
$ballot[\,]$, an array indexed by $\mathbb{N}$ of $\perp$ or records with the following fields:
$prev\text{-}rnd \in (\mathbb{N}_0)_\perp$, initially $\perp$
$clientM \subseteq msgs_B$, initially $\emptyset$
$clientCD \in \{\pm, \mathsf{null}\}$, initially null
$vnM \subseteq msgs_B$, initially $\emptyset$
$vnCD \in \{\pm, \mathsf{null}\}$, initially null
$round\text{-}status[\,]$, array indexed by $\mathbb{N}$ of $\langle$green, yellow, orange, red, $\perp\rangle$, initially $[\perp, \perp, \ldots, \perp]$
$last\text{-}reset \in \mathbb{N}_0$, initially 0
$last\text{-}good\text{-}state \in states$, initially $start_v$, the initial state for virtual node v
$prev\text{-}rnd \in \mathbb{N}_{0\perp}$, initially $\perp$

Round-Specific Data Structures:
$clientM \subseteq msgs_B$, initially $\emptyset$
$vnM \subseteq msgs_B$, initially $\emptyset$
$clientCD \in \{\pm, \mathsf{null}\}$, initially null
$vnCD \in \{\pm, \mathsf{null}\}$, initially null
$roundCM \in \{$active, passive$\}$, initially passive
$vnphaseBcast \in \{$true, false$\}$, initially false
$reset \in \{\mathbf{true}, \mathbf{false}\}$, initially false
$scheduled \in \{\mathbf{true}, \mathbf{false}\}$, initially false
$scheduled\text{-}status \in \{$green, yellow, orange, red, $\perp\}$, initially $\perp$

Control Data Structures:
$outgoing\text{-}msg \in msgs_B \cup \{\perp\}$, initially $\perp$
$do\text{-}net\text{-}bcast \in \{\mathbf{true}, \mathbf{false}\}$, initially false
$do\text{-}client\text{-}recv \in \{\mathbf{true}, \mathbf{false}\}$, initially false
$unscheduled\text{-}ballot\text{-}rnd \in [0, \mathsf{SMAX}]$, initially 0
$failed \in \{\mathbf{true}, \mathbf{false}\}$, initially false

Join Protocol Data Structures:
$joined \in \{\mathbf{true}, \mathbf{false}\}$, initially false
$join\text{-}req \in \{\mathbf{true}, \mathbf{false}\}$, initially false

Constants:
δ_v, transition function for virtual node $v \in I_V$
$loc(v)_V : I_V \to \mathbb{R} \times \mathbb{R}$, location of virtual node $v \in I_V$
$R_B \in \mathbb{R}$, broadcast radius for basic broadcast service
$R_B' \in \mathbb{R}$, interference radius for basic broadcast service
$R_V \in \mathbb{R}$, broadcast radius for virtual broadcast service
$R'_V \in \mathbb{R}$, interference radius for virtual broadcast service
$schedule[0..\mathsf{SMAX}\text{-}1]$, schedule array, where schedule[r] $\subseteq I_V$ and
(1) $\forall r \in [0..\mathsf{SMAX}\text{-}1], \forall v,w \in schedule[r], |loc[v] - loc[w]| \geq 2R'$
(2) $\forall v \in I_V, \exists\ r \in [0..\mathsf{SMAX}\text{-}1]$ **such that** $v \in schedule[r]$

Trajectories:
stops when:
$((do\text{-}net\text{-}bcast = \mathbf{true})$ **or** $(do\text{-}client\text{-}recv = \mathbf{true}))$
and
$(failed = \mathbf{false})$

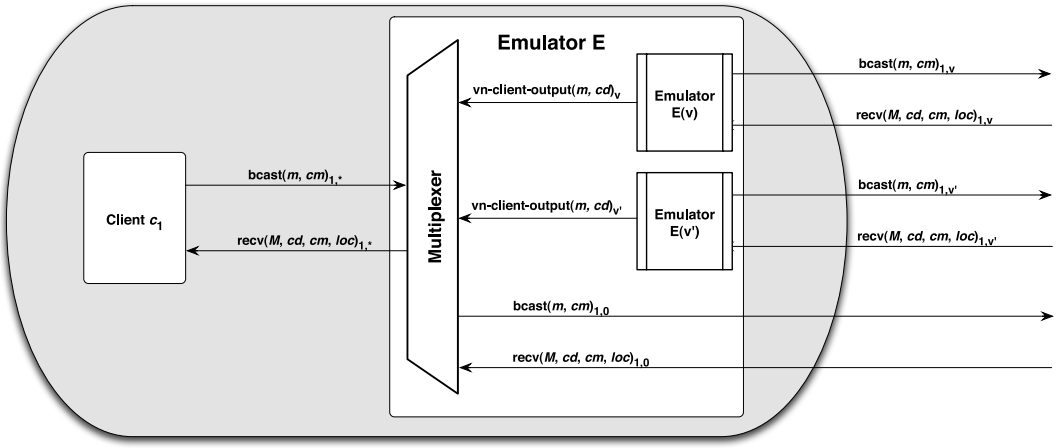

Figure 10-3: Detailed view of a process in the emulator system. The emulator E consists of two types of components: (1) a set of virtual node emulators $E(v)$, each of which is responsible for emulating one virtual node $V \in I_V$, and (2) a multiplexer that mediates communication between the client and the virtual node emulators. The emulator is composed with the client to form the process that is executed in the basic system.

Virtual Node Emulator. Each individual virtual node emulator $E(v)$ is responsible for replicating one particular virtual node $v \in I_V$. See the signature of $E(v)$ on lines 94–101 in Figure 10-2. Each individual virtual node emulator has access to one port on the wireless broadcast service:

- Input $\mathsf{recv}(\ldots)_v$,
- Output $\mathsf{bcast}(\ldots)_v$

Each virtual node emulator also outputs information to the client, acting as a filter between the broadcast channel and the client. Specifically, the virtual node emulator sends information via $\mathsf{vn\text{-}client\text{-}output}(\ldots)_v$ events to the multiplexer, which amalgamates the information from the various virtual node emulators, and passes it on to the client process itself.

An individual virtual node emulator has one other input, fail, which indicates when the mobile node on which the process executes has failed.

Multiplexer. The multiplexer relays the client's broadcasts to the broadcast service, and collects information from the virtual node emulators to pass on to the client. The signature of the multiplexer can be found in Figure 10-13 on lines 637–646. The multiplexer communicates with the client via a broadcast/receive port:

- Input $\mathsf{bcast}(\ldots)_*$
- Output $\mathsf{recv}(\ldots)_*$.

(Recall that the index "$*$" is a special literal, that is, $\mathit{process\text{-}ports}_V = \{*\}$.) Similarly, it communicates directly with the broadcast service via another broadcast/receive port:

- Output $\mathsf{bcast}(\ldots)_0$
- Input $\mathsf{recv}(\ldots)_0$.

Finally, it receives information from the virtual node emulators:

- Input $\mathsf{vn\text{-}client\text{-}output}(\ldots)_v$.

The multiplexer also has an input fail.

In Sections 10.2.2, 10.2.3 and 10.2.4, we discuss the virtual node emulator. In Section 10.2.5, we discuss the multiplexer.

10.2.2 Virtual Node Emulator Data Structures

In this section, we present the various data structures used by the virtual node emulator. The pseudocode can be found in Figure 10-2 on lines 104–150. Throughout this section, we discuss the individual virtual node emulator $E(v)$ for virtual node $v \in I_V$.

Emulator Data Structure:

- *rnd*: The main goal of the virtual node emulator is to determine which messages should be sent and received by the virtual node in a given round. The variable *rnd* keeps track of the current (virtual) round being emulated. (The actual round number in the underlying basic system is larger, as per the correspondence discussed in Section 10.1.) Initially, $rnd = 0$.

- *phase*: The protocol executes in phases, and the variable *phase* keeps track of the current phase. There are eleven different phases:

  1. client,
  2. vn,
  3. scheduled-ballot,
  4. scheduled-veto-1,
  5. scheduled-veto-2,
  6. unscheduled-ballot,
  7. unscheduled-veto-1,
  8. unscheduled-veto-2,
  9. join,
  10. join-ack,
  11. join-veto.

  Initially, *phase* = client. These phases can be divided into four groups: the first two phases are for the clients and replicas to perform their broadcasts for the virtual round; the next three phases are for the scheduled agreement protocol; the next three phases are for the unscheduled agreement protocol; the final three phases are for the join protocol.

- *ballot*: The primary task of the virtual node emulator is to agree on an acceptable execution history. The key data structure in maintaining the different viable execution histories is the *ballot* array (see lines 108–113, Figure 10-2). Each entry in the array holds a record containing information about a single simulated round. Each record contains five fields:

  - *prev-rnd*, a round number,
  - *clientM*, a set of messages,
  - *clientCD*, either $\pm$ or null,
  - *vnM*, a set of messages, and
  - *vnCD*, either $\pm$ or null.

  The record for simulated round r, which we refer to as a ballot, is stored in $ballot[r]$. That is, the array of ballots is indexed by $\mathbb{N}$.

  The ballot specifies which messages the virtual node v should receive from clients in round r in the *clientM* field, and collision detection information related to nodes in the "client" set of the partition for round r in the *clientCD* field. The ballot specifies which messages the virtual node v should receive from other virtual nodes in round r in the *vnM* field, and collision detection information for nodes in the "virtual nodes" set of the partition for round r in the *vnCD* field.

Each ballot also contains some additional information to specify one particular execution history. The pointer $ballot[r].prev\text{-}rnd$ points to the most recent "good" round which precedes r in the virtual node history, if any. That is, it is the most recent round designated as either green or yellow by the replica broadcasting the ballot.

In each simulated round, a replica for a given virtual node v participates in either the scheduled or the unscheduled agreement instance, depending on whether $v \in schedule[rnd \mod \mathsf{SMAX}]$. (Notice that since the first round is 1, the protocol begins with the second slot in the schedule; since the schedule is accessed cyclically, this does not matter.) Thus, in whichever agreement instance is appropriate, either zero, one or more replicas for virtual node v broadcast a ballot. If no ballot is received, or if more than one ballot is received, a replica designates the round as red. Otherwise, the replica stores the ballot, along with other information about the (virtual) round, in the *ballot* array. The ballot contains a proposal as to what should take place during the specific round, as well as a proposed execution history.

- *round-status*: During the simulation of the virtual round, the virtual node emulator associates a color with the round. The status of each simulated round is designated using four colors: *red*, *orange*, *yellow*, and *green*, ordered as $red < orange < yellow < green$. The information about round r's color is kept in the $round\text{-}status[r]$ variable. All the rounds are initially marked as $\perp$. The "default" status for a round is green, and the color is lowered as information is discovered during the execution of the protocol.

- *last-reset*: The variable *last-reset* maintains the most recent round in which the virtual node has been reset. More generally, this variable serves to remember a round in which the replicas are known to be in agreement (if such a round exists); the last round in which the virtual node was reset is a conservative estimate. Initially, $last\text{-}reset = 0$.

- *last-good-state*: The variable *last-good-state* stores the state of the virtual node after the round *last-reset*. Thus, in this case, *last-good-state* is always equal to *start*, the initial state of the virtual node.

- *prev-rnd*: The variable *prev-rnd* maintains the most recent round that the replica designates as either green or yellow. Initially $prev\text{-}rnd = \perp$, indicating that there has been no preceding green or yellow rounds.

Round-Specific Data Structures:

- $clientM, vnM$: The variables *clientM* and *vnM* store messages sent by the clients and replicas during the client and vn phases respectively, for the virtual round currently being simulated. Initially, both sets are empty.

- *clientCD*, *vnCD*: The variables *clientCD* and *vnCD* store the collision detection information related to the messages broadcast in the client and vn phases respectively. Initially, both are set to null.

- *roundCM*: The variable *roundCM* stores the advice given to the emulator. Initially, the advice is passive.

- *vnphaseBcast*: The variable *vnphaseBcast* remembers whether the emulator detects any broadcast in the vn phase of the virtual round. If the emulator receives a message or detects a collision in the vn phase, then *vnphaseBcast* $\leftarrow$ true; otherwise, *vnphaseBcast* $\leftarrow$ false.

- *reset*: The variable *reset* indicates whether the replica is involved in an ongoing reset of the virtual node. Initially, *reset* is false.

- *scheduled*: The flag *scheduled* $\in \{\text{true}, \text{false}\}$ indicates whether the virtual node is scheduled or not, according to the *schedule*, in the current virtual round. This determines whether the replica participates in the scheduled or unscheduled agreement instance; each replica participates in only one of the two agreement instances. Initially, *scheduled* is false.

- *scheduled-status*: The variable *scheduled-status* indicates the success level of the scheduled agreement instance, i.e., its color. Both scheduled and unscheduled virtual nodes observe this value. At the end of the scheduled agreement instance, this is stored in *round-status* if the replica is emulating a scheduled virtual node. Otherwise, it is used to determine whether to receive the message that was broadcast by the scheduled virtual node. Initially, *scheduled-status* is $\perp$.

Control Data Structures:

- *outgoing-msg*: The variable *outgoing-msg* stores the message to send in the next round by the emulator. Whenever the emulator executes a recv action, the next *outgoing-msg* is set. Initially, it is set to $\perp$.

- *do-net-bcast*, *do-client-recv*: The flags *do-net-bcast* and *do-client-recv* indicate when a message should be broadcast over the network or delivered to a client. These flags are used to restrict the trajectories of the emulator: whenever one of these flags is set, the appropriate event must occur immediately with no time passage (see lines 153–157, Figure 10-2). Initially, each of these flags is false.

- *unscheduled-ballot-rnd*: This variable is used to count the SMAX ballot rounds that occur during the unscheduled agreement instance. Unlike all the other phases of the protocol, which take only one basic round, the unscheduled-ballot requires SMAX phases in order that replicas from each virtual node get a chance to broadcast their ballots. This variable is responsible for keeping track of how many basic rounds have been executed during the unscheduled-ballot phase. Initially, this is set to 0.

- *failed*: The flag *failed* indicates whether the mobile node to which the process is assigned has failed. Initially, the *failed* flag is false.

Join Protocol Data Structures:

- *joined*: The flag *joined* tracks whether the join protocol has completed. When the flag is false, it indicates that the emulator has not yet joined the emulation, that is, the replica is not active in replicating the virtual node. Conversely, when the flag is true, the replica is participating actively in the replication. Initially, this flag is false.

- *join-req*: The flag *join-req* indicates that a join has been requested by some emulator process (potentially the same process). Initially, this flag is false.

Constants: The virtual node emulator also uses a set of universally known constants.

- δ_v: The transition function δ_v is the transition function for virtual node v.

- $loc(v)_V$: The location $loc(v)_V$ represents the location of virtual node v, which is fixed statically in advance.

- R_B, R'_B: The constant R_B is the broadcast radius of the basic broadcast service, while R'_B is the interference radius of the basic broadcast service.

- R_V, R'_V: The constant R_V is the broadcast radius of the virtual broadcast service, while R'_V is the interference radius of the virtual broadcast service.

- *schedule*: The constant *schedule* is a non-conflicting, complete schedule of size SMAX.

10.2.3 Virtual Node Emulator

In this section, we provide a detailed description of the emulator automaton $E(v)$, the process component responsible for emulating a single virtual node v. The pseudocode is presented in Figures 10-4–12. The first piece of pseudocode in Figure 10-4 describes how the emulator delivers messages to the multiplexer and the broadcast service. The main logic of the protocol is captured in Figure 10-5–9, which describe how the emulator processes incoming messages from the broadcast service. Figures 10-10 and 10-11 show how the replicas reconstruct an execution history from a sequence of ballot. Finally, Figure 10-12 includes pseudocode for calculating which message a virtual node should send or receive in a given round.

Throughout this discussion, we will refer to virtual node v as the virtual node being emulated. When, for the purpose of exposition, we discuss the behavior of the remapped process, we will refer to node i (or replica i) as the node executing the emulator process.

Figure 10-4: Automaton *E(v)*: Virtual Node Emulator 1 — Delivering Messages to the Network / Client

158 **Output** bcast(m, cm)$_v$
159 **Precondition**:
160 $failed$ = **false**
161 $do\text{-}net\text{-}bcast$ = **true**
162 $do\text{-}client\text{-}recv$ = **false**
163 m = $outgoing\text{-}msg$
164 **if** ($joined$ = **true**) **then**
165 cm = v
166 **else**
167 cm = $\perp$
168 **Effect**:
169 $outgoing\text{-}msg$ $\leftarrow$ $\perp$
170 $phase$ $\leftarrow$ $phase$ +1
171 $do\text{-}net\text{-}bcast$ $\leftarrow$ **false**
172
173 **Output** vn-client-output($vnm\text{-}out$, $vnCD\text{-}out$)$_v$
174 **Precondition**:
175 $failed$ = **false**
176 $do\text{-}client\text{-}recv$ = **true**
177 **if** ($round\text{-}status[rnd]$ = green) **then**
178 **if** ($v \in schedule[rnd \bmod \mathsf{SMAX}]$) **then**
179 **if** ($\exists$ $\langle$vn, v, $m\rangle \in ballot[rnd].vnM : m \in msgs_V$) **then**
180 **Choose** $\langle$vn, v, $m\rangle \in ballot[rnd].vnM$
181 $vnm\text{-}out$ = $\langle$vn, m, $loc(v)_V\rangle$
182 $vnCD\text{-}out$ = null
183 **else if** ($vnphaseBcast$ = **true**) **then**
184 $vnm\text{-}out$ = $\perp$
185 $vnCD\text{-}out$ = $\pm$
186 **else**
187 $vnm\text{-}out$ = $\perp$
188 $vnCD\text{-}out$ = null
189 **else if** ($v \notin schedule[rnd \bmod \mathsf{SMAX}]$) **then**
190 **if** ($vnphaseBcast$ = **true**) **then**
191 $vnm\text{-}out$ = $\perp$
192 $vnCD\text{-}out$ = $\pm$
193 **else if** ($round\text{-}status[rnd\text{-}1] \in$ {green, $\perp$}) **then**
194 $vnm\text{-}out$ = $\perp$
195 $vnCD\text{-}out$ = null
196 **else if** ($ballot[rnd].vnCD = \pm$) **or** ($vnCD = \pm$)**then**
197 $vnm\text{-}out$ = $\perp$
198 $vnCD\text{-}out$ = $\pm$
199 **else**
200 $vnm\text{-}out$ = $\perp$
201 $vnCD\text{-}out$ = null
202 **else if** ($round\text{-}status[rnd] = \perp$) **then**
203 $vnm\text{-}out$ = $\perp$
204 $vnCD\text{-}out$ = null
205 **else**
206 $vnm\text{-}out$ = $\perp$
207 $vnCD\text{-}out$ = $\pm$
208 **Effect**:
209 $do\text{-}client\text{-}recv$ $\leftarrow$ **false**
210
211 **Input** fail
212 **Effect**:
213 $failed$ $\leftarrow$ **true**
214 $scheduled$ $\leftarrow$ **false**

Figure 10-5: Automaton $E(v)$: Virtual Node Emulator 2 — Message Protocol

Input recv($allM$, cd, cm, loc)$_v$ **where** ($phase$ = client)
Local State:
  $next\text{-}vn\text{-}msg \in msgs_V \cup \{\bot\}$
  $out\text{-}req \in CM\text{-}names_V$
  $temp\text{-}state \in states_v$
  $temp\text{-}status[]$, array of ⟨green, yellow, orange, red⟩, initially $[\bot, bot, \ldots, \bot]$
Effect:
  $do\text{-}net\text{-}bcast \leftarrow$ **true**
  if ($|loc - loc(v)_V| > R_B/4$) **then**
    $joined \leftarrow$ **false**
  $outgoing\text{-}msg \leftarrow \bot$
  $phase \leftarrow$ vn
  $clientM \leftarrow \{\langle$client, m, $\ell\rangle \in allM : |\ell - loc(v)_V| \leq R_V\}$
  $clientCD \leftarrow cd$
  $roundCM \leftarrow cm$
  $rnd \leftarrow rnd+1$
  if ($v \in schedule[rnd$ mod SMAX$]$) **then**
    $scheduled \leftarrow$ **true**
  else
    $scheduled \leftarrow$ **false**
  $temp\text{-}status \leftarrow$ calculate-status($rnd - 1$, $prev\text{-}rnd$, $ballot$, $last\text{-}reset$)
  $\langle temp\text{-}state, \cdot\rangle \leftarrow$ calculate-state($rnd - 1$, $temp\text{-}status$, $last\text{-}reset$, $last\text{-}good\text{-}state$)$_v$
  $\langle temp\text{-}state, next\text{-}vn\text{-}msg, out\text{-}req, \cdot\rangle \leftarrow$ do-bcast($temp\text{-}state$)$_v$
  if ($next\text{-}vn\text{-}msg \neq \bot$) **and** ($rnd \neq last\text{-}reset+1$) **and** ($joined$ = **true**) **then**
    if ($roundCM$ = active) **or** ($scheduled$ = **false**) **then**
      $outgoing\text{-}msg \leftarrow \langle$vn, v, $next\text{-}vn\text{-}msg\rangle$

Input recv($allM$, cd, cm, loc)$_v$ **where** ($phase$ = vn)
Local State:
  $nearby\text{-}msgs \subseteq msgs_B$
  b, a record with the following fields:
    $prev\text{-}rnd \in \mathbb{N}_{0\bot}$, initially $\bot$
    $clientM \subseteq msgs_B$, initially $\emptyset$
    $clientCD \subseteq \{\pm,$ null$\}$, initially $\emptyset$
    $vnM \subseteq msgs_B$, initially $\emptyset$
    $vnCD \subseteq \{\pm,$ null$\}$, initially $\emptyset$
Effect:
  $do\text{-}net\text{-}bcast \leftarrow$ **true**
  if ($|loc - loc(v)_V| > R_B/4$) **then**
    $joined \leftarrow$ **false**
  $outgoing\text{-}msg \leftarrow \bot$
  $phase \leftarrow$ scheduled-ballot
  $nearby\text{-}msgs \leftarrow \{\langle$vn, v', $m\rangle \in allM : |loc(v')_V - loc(v)_V| \leq R_V\}$
  $vnM \leftarrow \emptyset$
  $vnCD \leftarrow$ null
  $vnphaseBcast \leftarrow$ **false**
  if ($|loc - loc(v)_V| < R_V + R_B/4$) **then**
    if ($\exists \langle$vn, v, $m\rangle \in nearby\text{-}msgs$) **or** ($cd = \pm$) **then**
      $vnphaseBcast \leftarrow$ **true**
    if ($|nearby\text{-}msgs| > 1$)
        or
      ($cd = \pm$)
        or
      ($(\exists \langle$vn, v', $m\rangle \in nearby\text{-}msgs : v' \notin schedule[rnd$ mod SMAX$], v' \neq v$)
    then
      $vnCD \leftarrow \pm$
    else
      $vnM \leftarrow nearby\text{-}msgs$
  if ($scheduled$ = **true**) **then**
    if ($joined$ = **true**) **and** ($roundCM$ = active) **then**
      $b \leftarrow \langle prev\text{-}rnd, clientM, clientCD, vnM, vnCD\rangle$
      $outgoing\text{-}msg \leftarrow \langle$vn, v, $b\rangle$

Figure 10-6: Automaton $E(v)$: Virtual Node Emulator 3 — Scheduled Agreement Instance

Input recv($allM$, cd, cm, loc)$_v$ **where** ($phase$ = scheduled-ballot)
Local State:
  $M \subseteq msgs_B$
  $my\text{-}ballots \subseteq msgs_B$
  $all\text{-}ballots \subseteq msgs_B$
  $ballot\text{-}msgs \subseteq msgs_B$
  $nearby\text{-}vn\text{-}msgs \subseteq msgs_B$
  $nearby\text{-}msgs \subseteq msgs_B$
  $near\text{-}scheduledV \subseteq I_V$
Effect:
  $do\text{-}net\text{-}bcast \leftarrow$ **true**
  if ($|loc - loc(v)_V| > R_B/4$) **then**
    $joined \leftarrow$ **false**
  $outgoing\text{-}msg \leftarrow \perp$
  $phase \leftarrow$ scheduled-veto-1
  $my\text{-}ballots \leftarrow \{b : \langle \mathsf{vn}, v, b\rangle \in allM\}$
  $all\text{-}ballots \leftarrow \{b : v' \in I_V, \langle \mathsf{vn}, v', b\rangle \in allM\}$
  $ballot\text{-}msgs \leftarrow \{\langle \mathsf{vn}, v', m\rangle \in b.vnM : v' \in I_V, b \in all\text{-}ballots\}$
  $nearby\text{-}vn\text{-}msgs \leftarrow$
    $\{\langle \mathsf{vn}, v', m\rangle \in ballot\text{-}msgs : |loc(v')_V - loc(v)_V| \leq R_V\}$
  $nearby\text{-}msgs \leftarrow$
    $\{\langle \mathsf{vn}, v', m\rangle \in nearby\text{-}vn\text{-}msgs : |loc(v')_V - loc| \leq R_V + R_B/4\}$
  $vnM \leftarrow vnM \cap nearby\text{-}msgs$
  $near\text{-}scheduledV \leftarrow \{v' \in I_V :$
    (1) v' is scheduled,
    (2) $|loc - loc(v')_V| \leq R_V + R_B/4\}$
  $scheduled\text{-}status \leftarrow \perp$
  if ($|loc - loc(v)_V| > R_V + R_B/4$) **then**
    $vnM \leftarrow \emptyset$
  else ($near\text{-}scheduledV = \emptyset$) **then**
    $vnM \leftarrow \emptyset$
  else choose unique $v' \in near\text{-}scheduledV$:
    if ($cd = \pm$)
      or
      ($|\{b : \langle \mathsf{vn}, v', b\rangle \in allM| > 1$)
    then
      $scheduled\text{-}status \leftarrow$ red
      $vnM \leftarrow \emptyset$
    if ($v = v'$) **and** ($scheduled$ = **true**) **and** ($joined$ = **true**) **then**
      if ($\exists\ b \in my\text{-}ballots$) **then**
        $ballot[rnd] \leftarrow b \in my\text{-}ballots$
      else
        $scheduled\text{-}status \leftarrow$ red
      if ($scheduled\text{-}status \neq \perp$) **then**
        $outgoing\text{-}msg \leftarrow \langle \mathsf{vn}, v, \mathsf{veto}\rangle$

Figure 10-7: Automaton $E(v)$: Virtual Node Emulator 3 — Scheduled Agreement Instance

```
322 Input recv(allM, cd, cm, loc)_v where (phase = scheduled-veto-1)
323 Effect:
324     do-net-bcast ← true
325     if (|loc − loc(v)_V| > R_B/4) then
326         joined ← false
327     outgoing-msg ← ⊥
328     phase ← scheduled-veto-2
329     vnM ← vnM −
330         {⟨vn, v′, m⟩ ∈ vnM : v′ ∈ I_V, |loc − loc(v′)_V| > R_V + R_B/4}
331     if (|loc − loc(v)_V| > R_V + R_B/4) then
332         vnM ← ∅
333         ballot[rnd] ← ⟨⊥, ∅, ∅, ∅, ∅⟩
334     if (⟨vn, v′, veto⟩ ∈ allM) or (cd = ±) then
335         if (scheduled-status ≠ red) then
336             scheduled-status ← orange
337     if (scheduled-status ∈ {red,orange})
338             and
339        (joined = true)
340             and
341        (scheduled = true) then
342             outgoing-msg ← ⟨vn, v, veto⟩
343
344 Input recv(allM, cd, cm, loc)_v where (phase = scheduled-veto-2)
345 Effect:
346     do-net-bcast ← true
347     if (|loc − loc(v)_V| > R_B/4) then
348         joined ← false
349     outgoing-msg ← ⊥
350     phase ← unscheduled-ballot
351     vnM ← vnM −
352         {⟨vn, v′, m⟩ ∈ vnM : v′ ∈ I_V, |loc − loc(v′)_V| > R_V + R_B/4}
353     if (|loc − loc(v)_V| > R_V + R_B/4) then
354         vnM ← ∅
355     if (⟨vn, v, veto⟩ ∈ allM) or (cd = ±) then
356         if (scheduled-status ∉ {orange, red}) then
357             scheduled-status ← yellow
358     if (scheduled = true) then
359         if (scheduled-status = ⊥) and (|loc − loc(v)_V| ≤ R_V) then
360             round-status[rnd] ← green
361         else
362             round-status[rnd] ← scheduled-status
363         if (round-status[rnd] ∈ {yellow, green}) then
364             if (joined = true) then
365                 prev-rnd ← rnd
366     else if (scheduled-status ≠ ⊥) then
367         vnM ← ∅
368         vnCD ← ±
369     unscheduled-ballot-rnd ← 0
```

Figure 10-8: Automaton $E(v)$: Virtual Node Emulator 4 — Unscheduled Agreement Instance

```
370 Input recv(allM, cd, cm, loc)_v where (phase = unscheduled-ballot)
371 Local State:
372     my-ballots ⊆ msgs_B
373 Effect:
374     do-net-bcast ← true
375     if (|loc − loc(v)_V| > R_B/4) then
376         joined ← false
377     outgoing-msg ← ⊥
378     if (scheduled = false) and (joined = true) then
379         if (unscheduled-ballot-rnd > 0) and (v ∈ schedule[unscheduled-ballot-rnd − 1]) then
380             my-ballots ← {b : ⟨vn, v, b⟩ ∈ allM}
381             if (my-ballots = ∅) or (|my-ballots| > 1) then
382                 round-status[rnd] ← red
383             if (cd = ±) then
384                 round-status[rnd] ← red
385             if (round-status[rnd] ≠ red) then
386                 ballot[rnd] ← b ∈ my-ballots
387         if (unscheduled-ballot-rnd < SMAX) then
388             if (v ∈ schedule[unscheduled-ballot-rnd]) then
389                 b ← ⟨prev-rnd, clientM, clientCD, vnM, vnCD⟩
390                 if (cm = active) then
391                     outgoing-msg ← ⟨vn, v, b⟩
392         else if (unscheduled-ballot-rnd = SMAX) then
393             if (round-status[rnd] = red) then
394                 outgoing-msg ← ⟨vn, v, veto⟩
395         if (unscheduled-ballot-rnd = SMAX) then
396             phase ← unscheduled-veto-1
397         else unscheduled-ballot-rnd ← unscheduled-ballot-rnd +1
398
399 Input recv(allM, cd, cm, loc)_v where (phase = unscheduled-veto-1)
400 Effect:
401     do-net-bcast ← true
402     if (|loc − loc(v)_V| > R_B/4) then
403         joined ← false
404     outgoing-msg ← ⊥
405     phase ← unscheduled-veto-2
406     if (scheduled = false) and (joined = true) then
407         if (⟨vn, v, veto⟩ ∈ allM) or (cd = ±) then
408             if (round-status[rnd] ≠ red) then
409                 round-status[rnd] ← orange
410         if (round-status[rnd] ∈ {orange,red}) then
411             outgoing-msg ← ⟨vn, v, veto⟩
412
413 Input recv(allM, cd, cm, loc)_v where (phase = unscheduled-veto-2)
414 Effect:
415     do-net-bcast ← true
416     if (|loc − loc(v)_V| > R_B/4) then
417         joined ← false
418     outgoing-msg ← ⊥
419     phase ← join
420     if (scheduled = false) then
421         if (joined = true) then
422             if (⟨vn, v, veto⟩ ∈ allM) or (±∈ cd) then
423                 if (round-status[rnd] ∉ {orange, red}) then
424                     round-status[rnd] ← yellow
425         if (round-status[rnd] = ⊥) and (|loc − loc(v)_V| ≤ R_V) then
426             round-status[rnd] ← green
427         if (joined = true) and (round-status[rnd] ∈ {yellow, green}) then
428             prev-rnd ← rnd
429     if (round-status[rnd] = red) then
430         ballot[rnd] ← ⟨⊥, ∅, ∅, ∅, ∅⟩
431     if (|loc − loc(v)_V| ≤ R_B/4) then
432         if (joined = false) and (v ∈ schedule[rnd mod SMAX])then
433             outgoing-msg ← ⟨vn, v, join⟩
```

Figure 10-9: Automaton *E(v)*: Virtual Node Emulator 5 — Join Protocol

Input recv($allM$, cd, cm, loc)$_v$ **where** ($phase$ = join)
Effect:
  $do\text{-}net\text{-}bcast \leftarrow$ **true**
  if ($|loc - loc(v)_V| > R_B/4$) **then**
    $joined \leftarrow$ **false**
  $outgoing\text{-}msg \leftarrow \perp$
  $phase \leftarrow$ join-ack
  $join\text{-}req \leftarrow$ **false**
  if ($v \in schedule[rnd \bmod \mathsf{SMAX}]$) **then**
    if ($allM \neq \emptyset$) **or** ($cd = \pm$)**then**
      $join\text{-}req \leftarrow$ **true**
      if ($joined$ = **true**) **and** (cm = active) **then**
        $outgoing\text{-}msg \leftarrow \langle$join, v, $\langle ballot, round\text{-}status, prev\text{-}rnd, vnphaseBcast, last\text{-}good\text{-}state, last\text{-}reset\rangle\ \rangle$

Input recv($allM$, cd, cm, loc)$_v$ **where** ($phase$ = join-ack)
Local State:
  $join\text{-}msgs \subseteq msgs_B$
Effect:
  $do\text{-}net\text{-}bcast \leftarrow$ **true**
  if ($|loc - loc(v)_V| > R_B/4$) **then**
    $joined \leftarrow$ **false**
  $outgoing\text{-}msg \leftarrow \perp$
  $phase \leftarrow$ join-veto
  $join\text{-}msgs \leftarrow \{m : \langle$join, v, $m\rangle \in allM\}$
  if ($|loc - loc(v)_V| \leq R_B/4$) **then**
    if ($joined$ = **false**) **then**
      if ($join\text{-}msgs \neq \emptyset$) **then**
        Choose $m \in join\text{-}msgs$.
        $\langle ballot, round\text{-}status, prev\text{-}rnd, vnphaseBcast, last\text{-}good\text{-}state, last\text{-}reset\rangle \leftarrow m$
        $joined \leftarrow$ **true**
      else if ($cd = \pm$)**then**
        $outgoing\text{-}msg \leftarrow \langle$vn, v, veto$\rangle$
  $reset \leftarrow$ **false**
  if ($v \in schedule[rnd \bmod \mathsf{SMAX}]$ **then**
    if ($joined$ = **true**) **then**
      $outgoing\text{-}msg \leftarrow \langle$vn, v, veto$\rangle$
    else if ($|loc - loc(v)_V| \leq R_B/4$) **then**
      $reset \leftarrow$ **true**

Input recv($allM$, cd, cm, loc)$_v$ **where** ($phase$ = join-veto)
Effect:
  $do\text{-}net\text{-}bcast \leftarrow$ **true**
  if ($|loc - loc(v)_V| > R_B/4$) **then**
    $joined \leftarrow$ **false**
  $outgoing\text{-}msg \leftarrow \perp$
  $phase \leftarrow$ client
  if ($|loc - loc(v)_V| \leq R_B/4$) **and** ($v \in schedule[rnd \bmod \mathsf{SMAX}]$) **and** ($reset$ = **true**) **then**
    if ($joined$ = **false**) **and** ($\langle$vn, v, veto$\rangle \notin allM$) **and** ($cd \neq \pm$)**then**
      $joined \leftarrow$ **true**
      $join\text{-}req \leftarrow$ **false**
      $last\text{-}good\text{-}state \leftarrow start_v$, the initial state of virtual node v
      $last\text{-}reset \leftarrow rnd$
      $prev\text{-}rnd \leftarrow rnd$
      for all $r \leq rnd$: $ballot[r] \leftarrow \langle \perp, \emptyset, \emptyset, \emptyset\rangle$
      for all $r \leq rnd$: $round\text{-}status[r] \leftarrow \perp$
  $do\text{-}client\text{-}recv \leftarrow$ **true**

Figure 10-10: Virtual Node Emulator 6 — calculate-status function

```
490 function calculate-status(temp-rnd, temp-prev, temp-ballots, temp-last-reset)
491 Signature:
492     temp-rnd, temp-last-reset ∈ ℕ0
493     temp-ballot[ ], an array indexed by ℕ of records with the following fields:
494         prev-rnd ∈ ℕ0, initially 0
495         clientM ⊆ msgs, initially ∅
496         clientCD ⊆ {±, null}, initially ∅
497         vnM ⊆ msgs, initially ∅
498         vnCD ⊆ {±, null}, initially ∅
499 Local State:
500     temp-status[ ], array indexed by ℕ of ⟨green, yellow, orange, red, ⊥⟩, initially [⊥, ⊥, . . . , ⊥]
501     r, p ∈ ℕ0
502 Code:
503     r = temp-rnd +1
504     p ← temp-prev
505     while r > temp-last-reset +1 do
506         r ← r − 1
507         if (r = p) then
508             temp-status[r] ← green
509             p ← temp-ballots[p].prev-rnd
510         else
511             temp-status[r] ← red
```

Figure 10-11: Virtual Node Emulator 7 — calculate-state function

512 **function** calculate-state(*temp-rnd*, *temp-status*, *temp-ballot*, *temp-last-reset*, *temp-state*)$_v$
513 **Signature:**
514 *temp-rnd*, *temp-last-reset* $\in \mathbb{N}_0$
515 *temp-status*[ ], array indexec by $\mathbb{N}$ of $\langle$green, yellow, orange, red, $\perp\rangle$
516 *temp-ballot*[ ], an array indexed by $\mathbb{N}$ of records with the following fields:
517 *prev-rnd* $\in \mathbb{N}_0$, initially 0
518 *clientM* $\subseteq$ *msgs*, initially $\emptyset$
519 *clientCD* $\subseteq \{\pm, \text{null}\}$, initially $\emptyset$
520 *vnM* $\subseteq$ *msgs*, initially $\emptyset$
521 *vnCD* $\subseteq \{\pm, \text{null}\}$, initially $\emptyset$
522 *temp-state* $\in$ *states*
523 **Local State:**
524 $i \in \mathbb{N}_0$
525 *temp-msg* $\in msgs_V$
526 *inCD* $\in \{\pm, \text{null}\}$
527 *inCM* $\in$ {active, passive}
528 *exec*, execution, initially empty
529 *e-frag-bc*, execution, initially empty
530 *e-frag-rcv*, execution, initially empty
531 **Constants:**
532 δ_v, transition function for virtual node v
533 $loc(v') \in \mathbb{R} \times \mathbb{R}$, $\forall v' \in V$, location of virtual node v'
534 $R_v \in \mathbb{R}$, broadcast radius for virtual node v
535 *schedule*[1..SMAX], schedule array, where schedule[r] $\subseteq$ V and
536 (1) $\forall r \in$ [0..SMAX-1], $\forall v,w \in$ *schedule*[r], $|loc[v]$ -$loc[w]| \geq 2R'$
537 (2) $\forall v \in V$, $\exists\ r \in$ [0..SMAX-1] **such that** $v \in$ *schedule*[r]
538 **Code:**
539 *exec* $\leftarrow \tau_0$, a point trajectory
540 **for** $r =$ *temp-last-reset* $+1$ **to** *temp-rnd* **do**
541 **if** ($v \in$ *schedule*[$r+1$ mod SMAX]) **then**
542 *inCM* $\leftarrow$ active
543 **else**
544 *inCM* $\leftarrow$ passive
545 **if** (*temp-status*[r] = green) **then**
546 **if** (*temp-ballot*[r].*clientCD* $= \pm$) **or** (*temp-ballot*[r].*vnCD* $= \pm$) **then**
547 *inCD* $\leftarrow \pm$
548 **else**
549 *inCD* $\leftarrow$ null
550 *inM* $\leftarrow \{m : \langle \text{client}, m, \cdot\rangle \in$ *temp-ballot*[r].*clientM*$\} \cup \{m : \langle \text{vn}, v', m\rangle \in$ *temp-ballot*[r].*vnM*, $v' \neq v\}$
551 **if** ($r =$ *temp-last-reset*$+1$) **then**
552 *e-frag-bc* $\leftarrow \perp$
553 **else**
554 $\langle$*temp-state*, *temp-msg*, $\cdot$, *e-frag-bc*$\rangle \leftarrow$ do-bcast(temp-state)$_v$
555 *inM* $\leftarrow$ *inM* $\cup$ {*temp-msg*}
556 $\langle$*temp-state*, *e-frag-rcv*$\rangle \leftarrow$ do-recv(*temp-state*, *inM*, *inCD*, *inCM*)$_v$
557 *exec* $\leftarrow$ *exec* . *e-frag-bc* . *e-frag-rcv*
558 **else if** (*temp-status*[r] = red) **then**
559 *inM* $\leftarrow \emptyset$
560 *inCD* $\leftarrow \pm$
561 **if** ($r =$ *temp-last-reset*$+1$) **then**
562 *e-frag-bc* $\leftarrow \perp$
563 **else**
564 $\langle$*temp-state*, *temp-msg*, $\cdot$, *e-frag-bc*$\rangle \leftarrow$ do-bcast(temp-state)$_v$
565 *inM* $\leftarrow$ *inM* $\cup$ {*temp-msg*}
566 $\langle$*temp-state*, *e-frag-rcv*$\rangle \leftarrow$ do-recv(*temp-state*, {*temp-msg*}, $\pm$, *inCM*)$_v$
567 *exec* $\leftarrow$ *exec* . *e-frag-bc* . *e-frag-rcv*
568 **return** $\langle$*temp-state*, *exec*$\rangle$

Figure 10-12: Virtual Node Emulator 8 — do-bcast and do-recv functions

569 do-bcast($temp\text{-}state$)$_v$
570 **Signature**:
571 $temp\text{-}state \in states$
572 **Local State**:
573 $next\text{-}vn\text{-}msg \in msgs$
574 $out\text{-}req \in \{\text{active}, \text{passive}\}$
575 $a \in$ **internal**
576 $s' \in states$
577 $exec$, a timed execution, initially empty
578 **Constants:**
579 δ_v, transition function for virtual node v
580 $loc(v') \in \mathbb{R} \times \mathbb{R}$, $\forall v' \in V$, location of virtual node v'
581 **Code**:
582 **while** $\nexists\ s' \in states$, $nxt\text{-}msg \in msgs_\perp$, $req \in CM\text{-}names_\perp$ s.t. $\langle tmp\text{-}state, \text{bcast}(nxt\text{-}msg, req)_{v,*}, s' \rangle \in \delta_v$ **do**
583 **Choose** $a \in internal$ and $s' \in states$ (deterministically) such that: $\langle$tmp-state, a, s$'\rangle \in \delta_v$
584 $exec \leftarrow exec\ .\ \langle tmp\text{-}state, a, s' \rangle$
585 $tmp\text{-}state \leftarrow s'$
586 **Choose** $nxt\text{-}msg \in M_\perp, req \in CM\text{-}names_\perp, s' \in states$ such that:
587 $\langle tmp\text{-}state, \text{bcast}(nxt\text{-}msg, req)_{v,*}, s' \rangle \in \delta_v$
588 $exec \leftarrow exec\ .\ \langle tmp\text{-}state, \text{bcast}(nxt\text{-}msg, req)_{v,*}, s' \rangle$
589 $tmp\text{-}state \leftarrow s'$
590 **return** $\langle tmp\text{-}state, nxt\text{-}msg, req, exec \rangle$
591
592 do-recv($temp\text{-}state$, M, cd, cm)$_v$
593 **Signature**:
594 $temp\text{-}state \in states$
595 $M \subseteq msgs$
596 $cd \in \{\pm, \text{null}\}$
597 $cm \in \{\text{active}, \text{passive}\}$
598 **Local State**:
599 $a \in$ **internal**
600 $s' \in states$
601 $timepassage \in \mathbb{R}$
602 $t \in \mathbb{R}$
603 $exec$, a timed execution, an initially empty
604 **Constants:**
605 δ_v, transition function for virtual node v
606 $loc(v') \in \mathbb{R} \times \mathbb{R}$, $\forall v' \in V$, location of virtual node v'
607 **Code**:
608 $timepassage \leftarrow RndLength_V$
609 **Choose** $t \leq timepassage$ to be the maximum value such that in state $temp\text{-}state$ time passage t is enabled.
610 **while** $timepassage > 0$ **do**
611 **while** $(t = 0)$ **do**
612 **Choose** $a \in internal$ and $s' \in states$ (deterministically) such that: $\langle$temp-state, a, s$'\rangle \in \delta_v$
613 $exec \leftarrow exec\ .\ \langle temp\text{-}state, a, s' \rangle$
614 $temp\text{-}state \leftarrow s'$
615 **Choose** $t \leq timepassage$ to be the maximum value such that in state $temp\text{-}state$ time passage t is enabled.
616 $exec \leftarrow exec\ .\ \tau(t)$ where $\tau(t)$ is a trajectory of length t
617 $timepassage \leftarrow timepassage - t$
618 **Choose** $s' \in states$ such that:
619 $\langle temp\text{-}state, \text{recv}(M, cd, cm, loc[v])_{v,*}, s' \rangle \in \delta_v$
620 $exec \leftarrow exec\ .\ \langle temp\text{-}state, \text{recv}(M, cd, cm, loc[v])_{v,*}, s' \rangle$
621 $temp\text{-}state \leftarrow s'$
622 **return** $\langle temp\text{-}state, exec \rangle$

Figure 10-13: Multiplexer Automaton

State:
$phase \in \{\text{in}, \text{out}\}$, initially out
$rnd \in \mathbb{N}_0$, initially 0
$clientBcast \in \{\textbf{true}, \textbf{false}\}$, initially false
$inVNs \subseteq V$, initially $\emptyset$
$inM \subseteq msgs_B$, initially $\emptyset$
$inCD \in \{\pm, \text{null}\}$, initially null
$inCM \in \{\text{active}, \text{passive}\}$, initially $\perp$
$inLoc \in \mathbb{R} \times \mathbb{R}$, initially $\langle 0, 0\rangle$
$bgnRnd \in \mathbb{R} \times \mathbb{R}$, initially $\langle 0, 0\rangle$
$outM \subseteq msgs_V$, initially $\emptyset$
$outCM \in \{\text{global}, \perp\}$, initially $\perp$
$failed \in \{\textbf{true}, \textbf{false}\}$, initially false

Signature:
Input:
vn-client-output$(vnm, vnCD)_v$, $vnm \in \{\langle \text{vn}, m, \ell\rangle : m \in msgs_V, \ell \in \mathbb{R} \times \mathbb{R}\}_\perp$, $vnCD \in \{\pm, null\}$
bcast$(m, cm)_*$, $m \in msgs_V$, $a \in CM\text{-}names_V \cup \{\perp\}$
recv$(M, cd, cm, loc)_0$, $M \subseteq msgs_B$, $cd \in \{\pm, null\}$, $cm \in \{\text{active}, \text{passive}\}$, $loc \in \mathbb{R} \times \mathbb{R}$
fail()

Output:
bcast$(m, cm)_0$, $m \in msgs_B$, $cm \in CM\text{-}names_B \cup \{\perp\}$
recv$(inM, inCD, inCM, inLoc)_*$, $inM \subseteq msgs_V$, $inCD \in \{\pm, null\}$, $inCM \in \{\text{active}, \text{passive}\}$, $inLoc \in \mathbb{R} \times \mathbb{R}$

Transitions:

Input vn-client-output$(vnm, vnCD)_v$
Effect:
  if $(vnm \neq \perp)$ **and** $(vnm = \langle \text{vn}, \cdot, \cdot\rangle)$**then**
    $inM \leftarrow inM \cup \{vnm\}$
  if $(\pm = vnCD)$ **then**
    $inCD \leftarrow \pm$
  $inVNs \leftarrow inVNs \cup v$

Output recv$(M, inCD, inCM, inLoc)_*$
Local State:
  $M\text{-}c \subseteq msgs_V$, initially $\emptyset$
  $M\text{-}vn \subseteq msgs_V$, initially $\emptyset$
Precondition:
  $inVNs = I_V$
  $phase = \text{out}$
  $rnd \bmod \text{RndLength}_V = 0$
  $M\text{-}c = \{m : \ell \in \mathbb{R} \times \mathbb{R}, \langle \text{client}, m, \ell\rangle \in inM, m \neq \perp, |\ell - bgnRnd| \leq R_V\}$
  $M\text{-}vn = \{m : \ell \in \mathbb{R} \times \mathbb{R}, \langle \text{vn}, m, \ell\rangle \in inM, m \neq \perp, |\ell - bgnRnd| \leq R_V\}$
  if $(inCD = \text{null})$ **then**
    $M = M\text{-}c \cup M\text{-}vn$
  else
    $M = \emptyset$
  $failed = \textbf{false}$
Effect:
  $inVNs \leftarrow \emptyset$
  $inM \leftarrow \emptyset$
  $bgnRnd \leftarrow inLoc$
  $inLoc \leftarrow 0$

Input bcast$(m, cm)_*$
Effect:
  $outM \leftarrow m$
  $outCM \leftarrow cm$
  $clientBcast \leftarrow \textbf{true}$

Output bcast$(m, cm)_0$
Precondition:
  $phase = \text{out}$
  $rnd \neq 0$
  if $(rnd \bmod \text{RndLength}_V \neq 0)$
  then
    $m = \perp$
  else
    $clientBcast = \textbf{true}$
    if $(outM = \perp)$ **then**
      $m = \perp$
    else
      $m = \langle \text{client}, outM, bgnRnd\rangle$
  if $(outCM = \text{client})$ **then**
    $cm = \text{global}$
  else
    $cm = \perp$
  $failed = \textbf{false}$
Effect:
  $phase \leftarrow \text{in}$
  $clientBcast \leftarrow \textbf{false}$
  $outM \leftarrow \perp$

Input recv$(M, cd, cm, loc)_0$
Effect:
  if $(rnd \bmod \text{RndLength}_V = 0)$ **then**
    $inM \leftarrow inM \cup \{m : \langle \text{client}, m, \cdot\rangle \in M\}$
    $inCD \leftarrow cd$
    $inCM \leftarrow cm$
  $inLoc \leftarrow loc$
  $rnd \leftarrow rnd + 1$
  $phase \leftarrow \text{out}$
  $clientBcast \leftarrow \textbf{false}$

Input fail()$_i$
Effect:
  $failed \leftarrow \textbf{true}$

Trajectories:
stops when:
  $phase = \text{out}$ **and** $failed = \textbf{false}$

Delivering Messages to the Network/Client

Figure 10-4 describes two types of behavior: broadcasting messages to other nodes/processes, and delivering messages to the client via the multiplexer. It also includes the fail input transition.

The bcast$(m, cm)_v$ action (lines 158–171) transmits a previously prepared message. The first two preconditions ensure that the emulator, when composed with a client process, satisfies the dynamic restrictions of a process. Specifically, Restriction 5 of a process (Definition 8.1.2) requires that a process perform no external actions after a fail input event. The flag *failed*—see line 160—ensures that the (external) broadcast event does not occur in this case. Restriction 4 of a process requires that it perform a broadcast immediately upon receiving a message, and that broadcasts occur only in response to receive events. This is enforced by the *do-net-bcast* flag, which is set to true only when a message is received (e.g., line 222, Figure 10-5) and reset to false when a broadcast occurs. The next precondition checks the *do-client-recv* flag; when this flag is set to true, a *vn-client-recv* event can deliver messages to the client.

The remaining preconditions determine which message to transmit, and whether to contend in the following (basic) round. If some message has been prepared, i.e., *outgoing-msg* $\neq \perp$, then that message is broadcast. Otherwise, no message is broadcast, as indicated by the *outgoing-msg* $= \perp$ symbol. Finally, the replica contends for contention manager v if it has successfully completed the join protocol, i.e., *joined* = true.

The second part of Figure 10-4 is the vn-client-output$(vnm\text{-}out, vnCD\text{-}out)_v$ action (lines 173–209), which delivers messages to the client (via the multiplexer, to be discussed later). The vn-client-output event occurs at the end of each virtual round, triggering the client to begin the next virtual round.

As usual, the *failed* flag in the precondition ensures that such events occur only when the process has not failed. The flag *do-client-recv* plays a similar role as the *do-net-bcast* flag: it ensures that the client receives its messages at exactly the correct time at the end of the virtual round. This flag is set by the last recv in the virtual round (line 489, Figure 10-5), and no further time can pass until the *vn-client-output* event occurs (see Figure 10-2, Trajectories). This flag ensures that the vn-client-output event precedes the bcast event at the end of the (virtual) round.

The next set of preconditions (lines 177–207) determines the data to send to the client. This decision depends on a series of questions: (1) Is round *rnd* designated as green, $\perp$, or otherwise? (2) Is v scheduled for virtual round *rnd*? (3) Does the round r ballot indicate that v broadcast a message, as determined by the $ballot[rnd].vnM$ state variable? (4) Does *vnphaseBcast* indicate that some node sent a message on behalf of virtual node v in the vn phase? (5) Is round $r - 1$ designated as green? (6) Does the round r ballot indicate a collision among virtual nodes, as determined by the $ballot[rnd].vnCD$ state variable? These conditions result in a series of cases:

- Round status is green, v is scheduled, and there exists some message in the ballot for the current round, i.e., $ballot[rnd].vnM$: In this case, the message is delivered, and no collision is detected. This is the only case in which a message is delivered to the client, via the multiplexer.

- Round status is green, v is scheduled, there is no message specified by the ballot, and the *vnphaseBcast* flag is set: In this case, no message ($\perp$) is delivered; however, a collision is detected, as the *vnphaseBcast* flag is used to indicate that some message may have been sent, but not received.

- Round status is green, v is scheduled, there is no message specified by the ballot, and the *vnphaseBcast* flag is not set: In this case, no message ($\perp$) is delivered, and no collision is detected.

- Round status is green and v is not scheduled: There are four sub-cases. First, if vnphaseBcast = true, then no message is delivered, and a collision is detected. (This is the second of four cases in which a collision is detected.) Second, if the preceding round $rnd - 1$ is designated as either green or $\perp$, then no message is delivered and no collision is detected. Third, if either the ballot or the *vnCD* variable indicates a collision in the current virtual round, then no message is delivered and a collision is detected. (This is the third of four cases in which a collision is detected.) Finally, in all other cases, no message is delivered, and no collision detected.

- Round status is $\perp$: In this case, no message is delivered, and no collision is detected. This case indicates that the current node is not involved with the round emulation.

- Round status is neither green nor $\perp$: In this case, no message is delivered, and a collision is detected, as the algorithm can not determine whether a message was sent or not. (This is the last of four cases in which a collision is detected.)

The effect of the vn-client-output transition is to reset the *do-client-recv* flag to false, indicating that the event has completed.

The last transition defined in Figure 10-4 is the fail action which sets the flag *failed* to true.

Processing Messages

Figures 10-5–10-9 contain the main portion of the protocol, the recv transition definitions, which processes messages received from the broadcast service. The recv transition is broken up into cases, one for each phase of the protocol. That is, the choice of which recv transition to invoke depends on the phase of the protocol, as determined by the local state of the emulator. For example, when phase = vn, the recv transition on lines 242–276 (Figure 10-5) is invoked.

There are four main portions to the protocol: the broadcast portion—phases client and vn (see Figure 10-5), the scheduled agreement portion—phases scheduled-ballot, scheduled-veto-1, scheduled-veto-2 (see Figures 10-6 and 10-7), the unscheduled agreement portion—phases unscheduled-ballot, unscheduled-veto-1, unscheduled-veto-2 (see Figure 10-8), and the join portion—phases join, join-ack, join-veto (see Figure 10-9).

The purpose of the broadcast portion of the protocol is to allow the clients an opportunity to broadcast messages, and to allow the replicas an opportunity to broadcast messages on behalf of the virtual node. The remainder of the two agreement protocols is, essentially, agreeing on which of these messages to receive. Notice that the clients and replicas broadcast their messages in separate rounds, ensuring non-interference.

The scheduled agreement portion of the protocol is designed to allow the replicas for "active" virtual nodes to run the agreement protocol, that is, it is designated for replicas for virtual nodes advised to be active in that virtual round. Since no two nearby virtual nodes are scheduled in the same virtual round, we can conclude that there is no interference caused by neighboring virtual nodes during this portion of the protocol. That is, the replicas for each scheduled virtual node can run the scheduled agreement portion of the protocol free from interference from other replicas for other virtual nodes.

The unscheduled agreement portion of the protocol is designed to allow the replicas for "passive" virtual nodes to run the agreement protocol, that is, it is designated for replicas for virtual nodes advised to be passive during that virtual round. In order to prevent virtual nodes from interfering with each other during the unscheduled agreement portion, we use the *schedule* to determine which virtual nodes participate in which (basic) rounds. Thus, the unscheduled agreement portion of the protocol requires SMAX rounds so that replicas for each virtual node have a chance to broadcast a ballot with no interference.

Both the scheduled and unscheduled instantiations of the agreement protocol implement the rules in Figure 10-1. They use the veto phases to ensure that no two replicas differ in color by more than one shade.

The three join phases allow a replica to join in the emulation. The replica sends a join request and receives in response a copy of the replicated state. When there are no other replicas that have already joined, the virtual node can be reset; the last join phase ensures that the virtual node is reset only when it has previously failed.

We now proceed to describe each recv transition definition—and each of the phases—in more detail. Each recv transition begins with some preliminary steps that are common to all of them: first, setting the flag *do-net-bcast* to true, ensuring that a response is broadcast immediately; second, checking if the replica is still close enough to participate in the emulation; third, initializing *outgoing-msg* to $\perp$; fourth, updating the phase flag to the next phase, thus ensuring that in the next (basic) round, the correct recv transition is invoked. (The only exception to this fourth step is in the unscheduled-ballot phase, where the phase is updated only after SMAX basic rounds.) Each recv transition then continues to implement the specific requirements of the phase. The last set of steps in a phase are to prepare the message to be sent in the following phase.

Broadcast Phases: (All the line numbers here refer to Figure 10-5.)

- client phase: (lines 215–240) The first phase of the protocol is the client phase. In the client phase, the clients broadcast their messages. They do

this directly (via the multiplexer) and hence the virtual node emulator does not broadcast anything in this round. The replica does, however, perform some bookkeeping to prepare for the rest of the virtual round emulation. We begin describing the behavior on line 227 after the shared initial four steps described above:

- First, the emulator records the messages received from nearby clients in *clientM*, and any collision detection information in *clientCD*. This information is used throughout the virtual round; a subset of these messages is received, conceptually, by the virtual node. (Messages from clients that are far away can be safely discarded.) The emulator also saves the contention management information in *roundCM*. This contention information is used through the remainder of the current virtual round.
- Next, the emulator increments the virtual round number *rnd* (line 230).
- Next, the replica examines the schedule to determine whether its virtual node is scheduled in this round; it stores the result of this calculation in the *scheduled* flag. This flag determines whether the replica participates in the schedule or unscheduled instance of the agreement protocol.
- Next, the replica determines whether the virtual node being emulated should broadcast a message in the current virtual round. Using the emulator variables *prev-rnd*, *ballot*, *last-reset*, and *last-good-state*, the replica calculates the current state for the virtual node. Recall that the replica actually maintains a set of possible executions; the functions calculate-status and calculate-state determine one preferred execution. (These functions will be discussed in further detail in Section 10.2.4). The replica then calls the do-bcast function, which calculates whether, in the current state, the virtual node should broadcast a message.
- If there is some message to broadcast (*next-vn-msg* $\neq \perp$), and if the virtual node was not just reset (*rnd* $\neq$ *last-reset*$+1$) , and if the replica has completed the join protocol (*joined* = true), then the replica may choose to broadcast the message in the vn phase which follows (see line 238). If the replica is emulating a scheduled virtual node, then it broadcasts the message only if the regional contention manager is advising the replica itself to be active. Otherwise, if the replica is emulating an unscheduled virtual node, then it broadcasts the message regardless of the advice of the contention manager. (See line 239.) This may well cause a collision, as the contention manager is being ignored. The message broadcast in this latter case will not be received, regardless: recall that in the vn-client-output transition, a message is delivered from a virtual node v only if v is scheduled for the round. This is sufficient to guarantee the ECF[GA] property of the virtual broadcast service.

- vn phase: (lines 242–276) In this phase, the emulator chooses which mes-

sages sent by virtual nodes are to be received by the virtual node v in the virtual round being emulated. Replicas broadcast messages on behalf of their virtual nodes in the vn phase. If exactly one neighboring virtual node is scheduled and broadcasts a message (via exactly one of its replicas), then this message is set to be received by virtual node v; otherwise, virtual node v is set to detect a collision in the virtual round being emulated. The recv transition for the vn phase begins with the same four initial steps. it then proceeds as follows:

- First, the emulator records the incoming messages sent by virtual nodes that are near to virtual node v (*nearby-msgs*).
- Next, *vnM* and *vnCD* are initialized to $\emptyset$ and null, respectively. Also, *vnphaseBcast* is initialized to false.
- Next (line 261), the emulator checks whether the replica is close enough to v to proceed. If j is within distance $R_V + R_B/4$ of $loc(v)$, then the emulator proceeds to calculate the sets *vnM* and*vnCD* and the flag *vnphaseBcast*.
- The emulator then attempts to determine the appropriate value for the vnphaseBcast flag. If there is some message from v in *nearby-msgs*, then the flag is set. Alternatively, if $cd = \pm$, i.e., if the emulator detects a collision, then the flag is set. From the completeness and accuracy properties of the collision detector we can determine that if a message is sent in the vn phase by a replica within distance R_B, this flag is set; if, after "stabilization" this flag is set, then some replica within distance R'_B sent a message.
- The emulator then checks a series of conditions which determine whether any of the messages from the nearby virtual nodes can be received, and whether a collision should be detected. If more than one message is received from a nearby node, or if a collision is detected, or if a message is received from some unscheduled virtual node ($\neq v$), then a collision is reported ($vnCD \leftarrow \pm$), and no messages are received, i.e. *vnM* remains $\emptyset$. Otherwise, *vnM* is set to *nearby-msgs*, that is, virtual node v should receive all the messages sent by nearby nodes. Notice that this ideal situation occurs only when exactly one message is received from a nearby virtual node, that virtual node is scheduled, and no collision is detected in the vn phase.
- Finally, the emulator assembles the ballot b, and prepares the message *outgoing-msg* for the following phase. A ballot is sent only if virtual node v is scheduled, only if the emulator has completed the join protocol, and only if the contention manager has advised the emulator to be active. In this case, *outgoing-msg* is set to $\langle \mathsf{vn}, v, b \rangle$, where ballot b consists of the following components:
  - *prev-rnd*, the most recent yellow or green round,
  - *clientM*, the set of messages broadcast by clients in the current virtual round,

* *clientCD*, the collision detection information for the client phase,
* *vnM*, the set of messages broadcast by replicas on behalf of virtual nodes in the current virtual round,
* *vnCD*, the collision detection information for the virtual node phase,

The result of the ballot will (eventually) be a decision as to which messages the virtual node should recv in that virtual round.

Scheduled Agreement Phases: (All the line numbers here refer to Figures 10-6 and 10-7.)

- scheduled-ballot phase: (lines 277–321) In this phase, the replica records the current ballot, and begins the process of choosing a color for the current round. The recv transition begins with the usual four steps, and then proceeds as follows:
  - First (lines 292–293), we consider certain sets of ballots derived from *allM*, the set of messages received in the current basic round. The set *my-ballots* contains all the ballots broadcast on behalf of virtual node v. The set *all-ballots* contains all the ballots broadcast on behalf of any virtual node. (It is clear that *my-ballots* $\subseteq$ *all-ballots*.)
  - Next (lines 294–298), we derive from the set of ballots, the sets of messages that virtual node v may choose to receive in this virtual round. The set *ballot-msgs* is the entire set of messages received in any ballot from any virtual node. The set *nearby-vn-msgs* is the restricted set of messages sent by a virtual node v' that is near to v. The set *nearby-msgs* is the further restricted set of messages sent by a virtual node v' that is near to the current location of the replica. As a result, every message in *nearby-msgs* is (1) included in a ballot in *allM*, (2) broadcast by a virtual node v' near to v, and (3) broadcast by a virtual node v' near to the location of the replica.
  - Recall that previously, in the vn phase, we established a set of messages *vnM* that the virtual node can receive in the current virtual node. At this point, we further restrict this set of messages, taking the intersection of this set with *nearby-msgs*. Thus, for a virtual node to receive a message in a virtual round from another virtual node, the emulator must both receive that message in the vn phase, and also find the message in a ballot received during the scheduled-ballot phase. This ensures that the message is part of the proposal received by all the replicas.
  - Next (lines 300–302), we calculate the set of virtual nodes that are near to v and also scheduled. This is stored in the variable *near-scheduledV*.
  - The scheduled-status is initialized to $\perp$. As the round proceeds, the color will be further determined; it remains $\perp$ at the end of the virtual

round only if the emulator is too far away from virtual node v to determine a color for the round.

- The first case to consider is where the node running the emulator is too far away from v. In this case, vnM is set to $\emptyset$: no messages are to be received. Notice that *scheduled-status* remains $\perp$.
- The second case to consider is when $nearby\text{-}scheduled V = \emptyset$, that is, there is no virtual node v' such that v' is scheduled and v' is close to the node executing the emulator. Notice that if v is scheduled, then this case is never reached; this covers the case where v is unscheduled, the node executing the emulator is close enough to v, but there is no scheduled virtual node nearby. In this case too, no message is received, i.e., $vnM \leftarrow \emptyset$. When the vnM variable is next examined in the unscheduled-ballot phase, the emulator will conclude that there are no messages to be received from a nearby virtual node.
- In the third case, the emulator is near enough to a scheduled virtual node to potentially receive messages. There is at most one element in the set $nearby\text{-}scheduled V$, since the *schedule* is non-conflicting. We choose v' as the unique nearby scheduled virtual node. (Notice that v' may equal v.) If $cd = \pm$, that is, if the emulator detects a collision in the scheduled-ballot phase, then the *scheduled-status* is set to red, the set of messages to receive vnM is emptied, and a collision is set to be detected by the virtual node (as per $vnCD$). Similarly, if more than one ballot is received from v', the status is set to red, no messages are received by the virtual node, and a collision is detected. Notice that if v is unscheduled, this information will next be used in the unscheduled-ballot phase.

  At this point, we proceed to fix the ballot and, possibly, prepare a veto message if the following three conditions hold: $v = v'$, v is scheduled, and the emulator has completed the join protocol. If there exists a ballot b in *my-ballots*, then that ballot is recorded in the ballot data structure. If there is no ballot, however, then we designate the round as red. Next, the replica performs a veto if anything has gone wrong. That is, if the *scheduled-status* is not $\perp$, then it broadcasts a veto message in the following phase. Notice that only replicas that have completed the join protocol participate in broadcasting veto message.

- scheduled-veto-1 phase: (lines 322–342) In the first veto phase, if the replica receives a veto message or detects a collision, it downgrades the color of the round from green or yellow to orange. (If it is already designated as orange or red, the color remains unchanged, of course.) It then prepares to broadcast a veto, if necessary.

  First, (after the usual initial four steps), the replica removes any messages from vnM that are sent by virtual nodes that are (now) too far away. If at any point during the virtual round, the replica exceeds a certain distance from virtual node v', then it should not propose receiving any messages

from v'.

Next, the replica checks whether it is near enough to v to continue to participate. If not, the set of message vnM is set to $\emptyset$, and the ballot is cleared.

Next (line 334) the emulator checks whether it has received a veto message or detected a collision. If so, then the *scheduled-status* is downgraded to orange (if it is not already red).

Finally, the emulator prepares to veto in the second veto phase if the round is currently designated as red or orange. This veto ensures that if the replica designates the round as orange, then all other replicas designate the round as at best yellow. The replica vetos only if it has completed the join protocol and is representing a virtual node that is scheduled.

- scheduled-veto-2 phase: (lines 344–369) In the second veto phase, if the emulator receives a veto phase or detects a collision, it downgrades the color of the round from green to yellow. If it is already yellow or orange or red, the color is of course left unchanged.

  As in the scheduled-veto-1 phase, the recv transition begins (after the usual initial four steps), by removing any messages from vnM that are sent by virtual nodes that are (now) too far away, and then by checking whether it is near enough to v to continue to participate.

  Next (line 355) the emulator checks whether it received a veto message or detects a collision. If so, the *scheduled-status* is downgraded to yellow, if it is not already orange or red.

  If the replica represents a virtual node that is scheduled, then the color of the round is now determined, and the replica updates the *round-status* array to reflect this new determination. Specifically, if the *scheduled-status* remains unmodified at $\perp$, and if the replica is sufficiently close to v, then the *round-status* is set to green. Otherwise, the *round-status* is updated to equal the calculated *scheduled-status*. The emulator then updates the *prev-rnd* pointer, if the round is green or yellow, and if the replica has completed the join protocol. This maintains the invariant that *prev-rnd* refers to the most recent round locally designated as green or yellow.

  If, on the other hand, the replica represents a virtual node that is unscheduled, then the emulator prepares for the unscheduled agreement portion of the protocol. Notice that the replica has observed the scheduled agreement phases, even though it has not participated. Recall that it has updated *scheduled-status*, even though it has not performed vetos. If it believes the status is green, then is prepares to receive whatever messages were broadcast during the scheduled portion of the protocol. Otherwise, it throws away any messages in vnM, and decides to detect a collision. In more detail, if the *scheduled-status* has been downgraded, i.e., is no longer $\perp$ (line 366), then the emulator determines that it should not receive any messages from virtual nodes (i.e., $vnM \leftarrow \emptyset$), and that it should detect a collision (i.e., $vnCD \leftarrow \pm$).

Finally, it sets *unscheduled-ballot-rnd* to zero, preparing for the unscheduled agreement phases.

Unscheduled Agreement Phases: (All the line numbers here refer to Figure 10-8.)

- unscheduled-ballot phase: (lines 370–397) The unscheduled ballot phase lasts for SMAX basic rounds; it is the only phase that takes more than a single basic round. The code for the unscheduled-ballot phase is divided into three parts: first, the node receives and processes messages; then it prepares to broadcast a message in the next (basic) round; finally, it checks if the phase is complete.

  As there are SMAX rounds in the unscheduled-ballot phase, the variable *unscheduled-ballot-rnd* keeps a count of basic rounds that have elapsed during the phase. The *schedule* determines which virtual nodes use which basic rounds. If $v \in schedule[unscheduled\text{-}ballot\text{-}rnd]$, then a replica for v may choose to broadcast a ballot in the next basic round (if it chooses to broadcast a ballot). If virtual node $v \in schedule[unscheduled\text{-}ballot\text{-}rnd-1]$, then a replica for v processes a ballot in the current basic round. (Notice, then, that all messages associated with virtual node v are broadcast and received in the basic round where $v \in schedule[unscheduled\text{-}ballot\text{-}rnd - 1]$.)

  The recv transition begins (after three of the usual four initial steps), by checking whether virtual node v is scheduled, and whether the replica has completed the join protocol. As this is the *unscheduled* instance, the emulator proceeds only if virtual node v is not scheduled. We also proceed only if the replica has completed the join protocol. (Notice that this is unlike the scheduled agreement portion, in which some preparation must be made for the later unscheduled portion, regardless of whether the v is scheduled or unscheduled, and regardless of whether the replica has completed the join protocol.) The emulator then proceeds as follows:

  – In the first part, a replica processes ballots received in a basic round where virtual node $v \in schedule[unscheduled\text{-}ballot\text{-}rnd-1]$ (see lines 379–386. The emulator begins by checking whether $unscheduled\text{-}ballot\text{-}rnd > 0$, and then proceeds to determine whether virtual node v is selected by the schedule, i.e., if

  $$v \in schedule[unscheduled\text{-}ballot\text{-}rnd - 1] \ ;$$

  this determines whether virtual node v is using this basic round as its ballot round. If so, *my-ballots* is assigned the set of ballots received for virtual node v in the current round. If $my\text{-}ballots = \emptyset$, then there is no ballot, and the round is designated as red. Similarly, if there is a collision (i.e., $cd = \pm$), then the round is designated as red. Finally, if the round is not designated as red, then the ballot is recorded.

– In the second part, a replica prepares to broadcast a ballot in the following basic round, under certain circumstances (lines 387–394). The emulator first checks that this is the appropriate basic round, i.e., *unscheduled-ballot-rnd* $<$ SMAX and virtual node v is selected by the scheduled:

$$v \in \mathit{schedule}[\mathit{unscheduled\text{-}ballot\text{-}rnd}] \ .$$

Next, ballot b is formed from the *prev-rnd*, *clientM*, *vnM*, and *vnCD* variables. (Notice that the *vnM* and *vnCD* variables were prepared in the vn and scheduled-ballot phases above; the *clientM* and *clientCD* variables were prepared in the client phase.) The ballot is broadcast, then, only if the replica is designated by the contention manager to be active. (Recall we have already assumed that the replica has completed the join protocol.) In the case where *unscheduled-ballot-rnd* $=$ SMAX, the replica prepares a message for the following phase, the unscheduled-veto-1 phase. In this case, if the round status has been downgraded to red, then *outgoing-msg* is set to $\langle \mathsf{vn}, v, \mathsf{veto} \rangle$, i.e., a veto message is prepared.

– In the third part, the phase is updated (lines 395–397). If the phase is not yet complete, i.e., if *unscheduled-ballot-rnd* $<$ SMAX, then the basic round count is incremented. Otherwise, if *unscheduled-ballot-rnd* $=$ SMAX, then the unscheduled-ballot phase is complete, and the phase is advanced to the unscheduled-veto-1 phase.

- unscheduled-veto-1 phase: (lines 399–411 In the first veto phase, if the replica receives a veto message or detects a collision, it downgrades the color of the round to orange. (If it is already designated as red, it remains red, of course.)

  First (after the usual four steps), the emulator checks that virtual node v is unscheduled, and that the replica has completed the join protocol. Otherwise, the replica does not participate in this veto phase.

  Next, the emulator determines whether a veto message has been received, or a collision detected. If so, the round status is downgraded to orange, if it is not already red.

  Finally, if the round is currently designated as red or orange, the replica prepares to veto in the second veto phase, thus ensuring that all other replicas designate the round as, at best, yellow.

- unscheduled-veto-2 phase: (lines 413–433) In the second veto phase, if the emulator receives a veto message or detects a collision, it downgrades the color of the round to yellow. If it is already orange or red, the color is of course left unchanged.

  First (after the usual four steps), the emulator checks that virtual node v is unscheduled, and that the replica has completed the join protocol. In this case, if the replica receives a veto message, or detects a collision, then the round status is downgraded to yellow, if it is not already red or orange.

Next (and even if the replica has not completed the join protocol), the round status is finally determined: if the *round-status*[*rnd*] remains $\perp$, meaning that it has not been downgraded, and if the replica is close enough to v, then the round status is designated as green.

In all cases, the color of the round is now determined. Therefore, the replica updates the *prev-rnd* pointer: if the round is green or yellow, the replica stores the current round in *prev-rnd*. This maintains the invariant that *prev-rnd* designates the most recent round locally designated to be green or yellow.

Next, if the round status is red, the ballot is emptied. Since the round is red, we cannot rely on any information received in the ballot.

The final part of the unscheduled-veto-2 phase prepares the message for the join phase. Specifically, the mobile node broadcasts a join request under the following conditions: (1) the mobile node hosting the replica process is within distance $R_B/4$ of the virtual node, (2) the replica has not completed the join protocol, i.e., *joined* = false, and (3) virtual node v is scheduled for the current virtual round, i.e., $v \in$ *schedule*[*rnd* mod SMAX].

Join Phases: (All the following line numbers here refer to Figure 10-9.)

- join phase: (lines 434–446) In the join phase, a node that has not yet completed the joined protocol may broadcast a join request. This join request is received and processed by nodes that have already joined, and these nodes then send a join response.

  First (after the usual four steps), *join-req* is initialized to false. Under the following circumstances, it is set to true: (1) Virtual node v is scheduled for the current virtual round; this ensures that any join request comes from a replica for virtual node v. (2) Either $allM \neq \emptyset$, indicating some replica sent a join request, or $cd = \pm$, indicating that either some replica sent a join request or the collision detector returned a false positive. Thus, if some replica for virtual node v broadcast a join request, *join-req* is set to true.

  Under these circumstances, since there is a join request, the replica sends a join response in the next phase if it has, itself, completed the join protocol, and if cm = true, indicating that the current replica has been designated active for this basic round.

  The resulting join response includes all the state associated with the replica. Specifically, notice that all the input parameters to calculate-status and calculate-state are included in the join response, with the exception of *rnd* (which each replica can calculate on its own).

- join-ack phase: (lines 448–471) In this phase, nodes sent responses to the join requests. These responses are received by the unjoined nodes, which process these responses and join the emulation.

The set *join-msgs* represents the set of join responses for virtual node v. After the usual initial four steps, this set is derived from *allM*, the set of messages received in the join-ack phase. If the replica is within distance $R_B/4$ of virtual node v, and has not yet completed the join protocol, and if it has received a join response, then it adopts the replica state from the join response. If the set of *join-msgs* is empty, but $cd = \pm$, then some replica attempted (unsuccessfully) to send a join response; the replica broadcasts a veto in the following join-veto phase, preventing the virtual node from being accidently reset.

Finally, the replica prepares for the join-veto phase. It begins by initialize *reset* to false. If v is scheduled for the current virtual round and if the replica has completed the join protocol, then it broadcasts a veto in the following join-veto phase, thus ensuring that the virtual node is not accidently reset. Alternatively, if the replica has not successfully completed the join protocol, and yet is still close to virtual node v, then it indicates that a reset is desirable, setting *reset* to true.

- join-veto phase: (lines 473–489) If there is no veto in the join-veto phase, then a joining replica knows that no other replica has completed the join protocol, and thus the virtual node must be reset. After the usual initial four steps, the emulator determines whether a reset is appropriate, examining the following conditions: (1) the replica is within distance $R_B/4$ of the virtual node v, (2) virtual node v is scheduled for the current virtual round, (3) the *reset* flag is set to true, indicating that a reset is desirable, (4) the replica has not yet itself completed the join protocol, (5) the replica does not receive a veto message, and (6) the replica does not detect a collision. The last two conditions ensure that no other replica broadcast a veto message.

  Notice that a reset occurs only if v is scheduled; this is because it is only in virtual rounds where v is scheduled that the join protocol for v is invoked; hence we can determine that the join protocol has failed only in such rounds. Also, notice that the *reset* flag is used to determine whether a reset is desirable. The reset flag is set in the join-ack phase when a join is unsuccessful.

  When a reset has occurred, the replica sets its *joined* flag to true, the *join-req* flag to false, and resets its replica state as per lines 482–488.

  Finally, the replica ensures that messages are delivered to the client at the end of the virtual round by setting the *do-client-recv* flag, which forces a vn-client-output to occur immediately afterwards (as per the Trajectories in Figure 10-2).

10.2.4 Virtual Node Emulator Helper Functions

In this section we discuss the helper functions used by the virtual node emulator to construct an execution of the virtual node.

Calculating the Round Status

The calculate-status helper function in Figure 10-10 calculates the status of each round. It takes as input the following parameters:

- *temp-rnd*, the current virtual round number,
- *temp-prev*, a copy of *prev-rnd*, the most recent round locally designated as yellow or green,
- *temp-ballot*, a copy of the *ballot* array,
- *temp-last-reset*, a copy of *last-reset*, the most recent green round.

The goal of this helper function is to decide which virtual rounds to accept and which to reject. When a virtual round is accepted, the virtual node receives the messages designated by the ballot in that round. When a virtual round is rejected, the virtual node detects a collision, regardless of the information in the ballot. Thus, the output of the calculate-status function is the *temp-status* array in which each round has been designated as either green or red. This status array is then used by the calculate-state function to determine the current state of the virtual node.

The status is calculated backward, first starting with the current virtual round, *temp-rnd*, and counting backwards to *temp-last-reset*, the most recent time the virtual node was reset to its initial state. The variable r acts as the loop counter.

In the beginning, the pointer $p = $ *temp-prev*, the most recent "good" round $\leq r$. Thus, all rounds larger than p should be designated as red: as we count backwards in the array, until we reach round p, each round is designated as red. When we reach round p, we designate that round as green, and update p with the *prev-rnd* designated in the ballot from round r. We will show that when this happens, the ballot from round r always contains a valid entry.

We will show in Corollary 11.6.7 that if two replicas share the same *prev-rnd*, then the calculate-status helper function will calculate the same status for each round. Thus in any green round, every replica will designate the round as green or yellow, resulting in all the replicas agreeing on the *prev-rnd*, and as a result agreeing on the status of all prior rounds.

Calculating the Replicated State

The calculate-state helper function in Figure 10-11 calculates the current state of the virtual node. It reconstructs one plausible execution of the virtual node, based on the following parameters:

- *temp-rnd*, a copy of *rnd*, the current round of the virtual node,
- *temp-status*, an array indicating which rounds have been accepted and which rejected; this array is the result of the calculate-status helper function,
- *temp-ballot*, a copy of the *ballot* array,

- *temp-last-reset*, a copy of *last-reset*, the most recent round in which v was reset,
- *temp-state*, a copy of *last-good-state*, the state of the replica at the end of *temp-last-reset*.

The calculate-state function starts at the round following *last-reset*, and incrementally calculates the state of the virtual node. The variable r represents the virtual round being considered, and hence is iterated from *temp-last-reset* + 1 to *temp-rnd*. The function proceeds as follows:

- First, it calculates the contention management advice to provide to the virtual node: if $v \in schedule[r \mod \mathsf{SMAX}]$, then v is advised to be active; otherwise, v is advised to be passive. This information is stored in *inCM*.
- Next, it breaks into two cases, depending on whether the virtual round is accepted or rejected, that is, whether the calculate-status function designated this round as green or red.

  Green/Accept:

  First, it determines whether the virtual node should detect a collision based on the information in the ballot. This information is stored in *inCD*. Next, it calculates the set of messages the virtual node should receive, based on the information in the ballot. Specifically, it includes every message from the *clientM* field of the ballot, and every message from the *vnM* field of the ballot except those initiated by virtual node v. Any message from v itself will be calculated directly, rather than derived from the ballot: the next step proceeds to calculate the message broadcast by v using the do-bcast function. Notice that if the virtual node was just reset, i.e., if r = *temp-last-reset* + 1, then virtual node v does not send any messages in round r. The message calculated (if any) is added to *inM*, and the associated execution fragment is stored in *e-frag-bc*. Finally, it calls the do-recv function to simulate the virtual node receiving the specified messages *inM*, detecting a collision as specified by *inCD*, receiving contention management advice as per *inCM*, and receiving the the (constant) location. The fragment of the execution in which the virtual node receives the message is stored in *e-frag-rcv*, and hence the entire round execution fragment consisting of *e-frag-bc*.*e-frag-rcv* is appended to *exec*.

  Red/Reject:

  In this case, the replica simulates the virtual node receiving only its own message (if it broadcasts one), detecting a collision, and receiving the calculated contention management information and the (constant) location. Thus, the set of message, *inM* is initially set to be $\emptyset$, and *inCD* is set to $\pm$. As in the previous case, it calculates the message that virtual node v sends in this case, and adds the message to *inM*, storing the execution fragment in *e-frag-bc*. The receive portion of the execution fragment is calculated using the do-recv function, and the execution fragments concatenated to *exec*. Notice that in this case, no information from the ballot is used.

At the end of each iteration, i.e., after completing the calculation for round r, the execution *exec* consists of a (timed) execution of the virtual node through the end of round r. In the end, it returns the final state of the virtual node, and the execution of the virtual node that has been generated. This execution is used primarily in the proof, and is not needed for the algorithm.

Additional Helper Functions:

The remaining helper function in Figure 10-12 are relatively straightforward.

Broadcasting messages. The function do-bcast calculates which message the virtual node v sends when it is in the state *temp-state*. It also calculates the resulting new state, after the message is broadcast, the resulting contention manager request, and the execution fragment produced by the broadcast event.

The first step in this calculation is to perform any necessary internal actions such that a broadcast is enabled. The do-bcast function is called only when the last transition of the virtual node was a recv, and hence the properties of a process ensure that after some set of internal actions a broadcast will be enabled. (See Lemma 10.3.1 for a slightly more detailed rendition of this argument.) Next, the function calculates the state after the broadcast, and returns the requisite information.

Receiving messages. The function do-recv calculates the state after receiving a given set of information: a set of messages M, collision detection feedback cd, and contention manager advice cm. First, it advances time for one virtual $RndLength_V$ worth of time. The do-recv function is called immediately after a do-bcast, and hence time must be advanced to the end of the round, before the recv transition should occur. Since all the variables are constant with respect to time passage, it is sufficient to perform whatever internal actions are necessary to allow time to pass. Since the process is an internally-progressive TIOA, some set of actions are sufficient to accomplish this; since the process satisfies the immediate response property, none of the actions will be external output actions. (See Lemma 10.3.2 for a slightly more detailed rendition of this argument.) Finally, it calculates the resulting state, and returns it. It also returns the execution fragment which includes the time-passage events, the necessary internal events, and the recv event.

10.2.5 Virtual Node Multiplexer

The virtual node multiplexer, presented in Figure 10-13, is relatively straightforward. Its main purpose is to aggregate messages from the virtual node emulators, as well as the broadcast service, and deliver them to the client. It also acts as an intermediary between the client and the broadcast service, relaying client messages to the broadcast service.

Multiplexer State

The multiplexer contains the following state components:

- *phase*: specifies whether the multiplexer is relaying information inwards, from the broadcast service to the clients, or outwards, from the clients to the broadcast service. Whenever $phase = \mathsf{out}$, no time is allowed to pass, according to the restriction on trajectories; this indicates that the multiplexer needs to broadcast a message immediately in order to satisfy the requirement that a process immediately broadcast a message whenever one is received (Definition 8.1.2).
- *rnd*: stores the current basic round number, which is incremented every time a message is received from the broadcast service.
- *clientBcast*: this flag indicates whether the client has broadcast its message yet for the current virtual round. The multiplexer cannot re-broadcast the client message until the client has transmitted the message to the multiplexer.
- *inVNs*: stores the set of virtual nodes that have, so far, delivered information to the client.
- *inM*: stores the set of messages to deliver to the client.
- *inCD*: stores the collision detection information to deliver to the client.
- *inCM*: stores the contention management information to deliver to the client.
- *inLoc*: stores the location information to deliver to the client.
- *beginRound*: stores the location at the beginning of the virtual round.
- *outM*: stores the messages broadcast by the client.
- *outCM*: stores the contention management request from the client.
- *failed*: flag indicating whether the process has failed.

Multiplexer Signature

The multiplexer has the following input and output actions:

- Input $\mathsf{vn\text{-}client\text{-}output}(vnm, vnCD)_v$: receives a message and collision detection feedback from the emulator on behalf of virtual node v.
- Input $\mathsf{bcast}(m, cm)_*$: receives the broadcast from the client.
- Input $\mathsf{recv}(M, cd, cm, loc)_0$: receives messages from the broadcast service on port 0.
- Input fail: indicates that the process has failed.
- Output $\mathsf{bcast}(m, cm)_0$: broadcasts a message using the broadcast service port 0.
- Output $\mathsf{recv}(m, cd, cm, loc)_*$: delivers messages to the client.

Multiplexer Transitions

The main functionality of the multiplexer is as follows. Each virtual round consists of a series of basic rounds. A virtual round begins with a client broadcasting a message during the client phase (when $rnd \mod RndLength_V = 0$), in response to the receive event from the previous round. The multiplexer receives this bcast_* event, and relays this message to the broadcast service via a bcast_0 event. Throughout the rest of the phase, the multiplexer simply receives the messages from the broadcast service, and ignores them, broadcasting empty messages in response. Finally, when the last receive event in the virtual round occurs, the multiplexer collects messages from the virtual node emulators, via vn-client-output events, and delivers them to the client, along with any other messages received during the client phase, via a recv_* event. In more detail:

- Input $\mathsf{vn\text{-}client\text{-}output}(vnm, vnCD)_v$: These input events occur during the last phase of a virtual round. That is, they occur at exactly the end of a virtual round. The multiplexer collects the messages from the various virtual nodes in inM and the collision detection information in $inCD$. Notice that if any virtual node notifies the client of a collision, then $inCD \leftarrow \pm$. The multiplexer then adds virtual node v to the set of virtual nodes $inVNs$ that have delivered information to the client in this virtual round.

- Output $\mathsf{recv}(M, inCD, inCM, inLoc)_*$: These output events deliver information to the client. They occur at the end of a virtual round. The preconditions ensure that: (1) every virtual node has delivered information for the current round ($inVNs = I_V$), (2) the multiplexer is ready to broadcast a message ($phase = \mathsf{out}$), (3) the current basic round is the last phase in the virtual round ($rnd \mod RndLength_V = 0$; recall that the first basic round is numbered 1, and hence the last basic round in a virtual round is divisible by $RndLength_V$), (4) *M-client* and *M-vn* calculate the appropriate messages from the clients and virtual nodes; only messages sent by nodes that were nearby at the beginning of the virtual round are received; messages are delivered only if there is no collision detected, and (5) the process has not failed. On delivering the appropriate messages, collision detection information, contention management information, and location information to the client, each of these variables is reset to the $\emptyset$.

- Input $\mathsf{bcast}(m, cm)_*$: These input events are the broadcast events for the clients for a given virtual round. They happen immediately after the recv_* events, and initiate the next virtual round. The message being broadcast is stored in $outM$, and the contention management request is stored in $outCM$. The flag *clientBcast* is set to true to indicate that the multiplexer is now ready to broadcast the client's message.

- Output $\mathsf{bcast}(m, cm)_0$: These output events relay the client's messages to the broadcast service, during the client phase, and broadcast empty messages in all other phases. The preconditions ensure that the multiplexer is ready to broadcast the client's message, if it is the client phase, or an empty message

otherwise: First, the multiplexer is ready to broadcast a message (phase = out and $rnd \neq 0$); recall that a process does not broadcast a message in the first round. Next, if the current round is a client phase (as determined by examining the *rnd* variable), then the client has broadcast its message (*clientBcast* = true), and the message is of the form $\langle \mathsf{client}, m, \ell \rangle$, where m is the client's message, and ℓ is the location of the client when the message was sent at the beginning of the round. (This additional information helps the emulator to process the message.) Next, The process has not failed (*failed* = false). After a broadcast event, the phase is set to in, indicating the multiplexer is now waiting to receive a message. The flag *clientBcast* is reset, and the message *outM* is reset to $\perp$.

- Input $\mathsf{recv}(M, cd, cm, loc)_0$: These input events deliver messages from the broadcast service to the multiplexer. In the client phase, these messages originate at other clients, and hence are saved, to be delivered to the client at the end of the virtual round. Similarly, the collision detection and contention management information is saved in the client phase. (As the round counter has not yet been incremented, $rnd \bmod RndLength_V = 0$ in the client phase recv event.) The location *inLoc* is updated in each round, even after the client phase so that at the end of the virtual round the client receives the most up-to-date location estimate. The round counter is then incremented, and the phase is set to out indicating that the multiplexer needs to broadcast a message immediately. Lastly, the *clientBcast* flag is reset to false.

- Input fail: This input action indicates that the process has failed; therefore the *failed* flag is set to true.

10.2.6 Putting the Pieces Together

In this section, we assemble the components previously described, and provide the full specification for the basic system, that is, fixing the parameters

$$P_B, \mathit{process\text{-}ports}_B, \mathit{msgs}_B, A_B, \mathit{CM\text{-}names}_B, CM_B \ .$$

First, we define the emulator automaton E to be the composition of the following automata:

$$\{E(v) : v \in I_V\} \cup \{\mathit{Multiplexer}\} \ .$$

We continue to use the notation $E(v)$, for $v \in I_V$, to refer to the emulator automaton for v in the composition; similarly $E(\mathit{multiplexer})$ refers to the multiplexer portion of the composition.

Next, we define the set of processes, P_B. For each client process in the virtual infrastructure system, we instantiate a new process consisting of the client composed with the emulator. That is:

$$P_B = \{A_V(i) \times E : i \in I_B\} \ .$$

Each process $A_V(i)$ is a client process in the virtual infrastructure system, and thus is

composed with the emulator automaton E to form a new process in the basic system. We show in Section 10.3 that $A_V(i) \times E$ is, in fact, a process as per Definition 8.1.2.

Next, we define an algorithm A_B which assigns a process to each mobile node. That is, for each $i \in I_B$:

$$A_B(i) = A_V(i) \times E \ .$$

Notice that A_B is anonymous if A_V is anonymous.

We provide each process with one port per virtual node, and one additional port. That is:

$$\mathit{process\text{-}ports}_B = I_V \cup \{0\} \ .$$

The set of broadcast ports $\mathit{bcast\text{-}ports}_B = I_B \times \mathit{process\text{-}ports}_B$, as defined in Chapter 8.

We define one contention manager for each virtual node location, along with one additional "global" contention manager. The contention managers have names $\mathit{CM\text{-}names}_B = I_V \cup \{\mathsf{global}\}$, and thus

$$CM_B = \{CM_v : v \in I_V\} \cup \{CM_{\mathit{global}}\}$$

We assume that CM_v, for $v \in I_V$, is an ℓ-regionally fair contention manager for the port set $\mathit{idbcast} - \mathit{ports}_B$, where $\ell = \mathit{loc}(v)_V$. Additionally, we assume that every contention manager is conservative. We assume that the contention manager CM_{global} is conservative, but make no further assumptions as to its operation. (In Section 11.10 we discuss further the relationship of CM_{global} in the basic system to CM_{client} in the virtual infrastructure system.)

Finally, we define the set of messages msgs_B to be all the possible messages that the emulator automaton might send. That is:

- $\langle \mathsf{client}, m, \ell \rangle$, $\forall m \in \mathit{msgs}_V$, $\forall \ell \in \mathbb{R} \times \mathbb{R}$,
- $\langle \mathsf{vn}, v, m \rangle$, $\forall m \in \mathit{msgs}_V$, $\forall v \in I_V$,
- $\langle \mathsf{vn}, v, b \rangle$, for b a ballot, $\forall v \in I_V$,
- $\langle \mathsf{vn}, v, \mathsf{veto} \rangle$, $\forall v \in I_V$,
- $\langle \mathsf{vn}, v, \mathsf{join} \rangle$, $\forall v \in I_V$,
- $\langle \mathsf{join}, v, \langle b, r, p, f, l, s \rangle \rangle$, where b is a ballot, r is a round status array, p is a round number, f is a boolean flag, l is a round number, and s is a state of a virtual node v.

We refer to the virtual infrastructure system defined by this particular set of parameters:

$$VS(P_B, \mathit{process\text{-}ports}_B, \mathit{msgs}_B, A_B, \mathit{CM\text{-}names}_B, CM_B)$$

as the **emulator system**.

10.3 Well-Formedness of the Emulator

Before proceeding to discuss the correctness of the emulator system (in Chapter 11), we must first establish that the automata presented in Section 10.2 in fact describes a process. Specifically, we show in Theorem 10.3.7 that for all $i \in I_B$, if A is a process, then $A \times E$ is a process, according to Definition 8.1.2. As a result, the set of processes P_B described in Section 10.2.6 is, in fact, a set of processes, and the emulator system described is reasonable.

In order to verify that automaton $A \times E$ is a process, for all processes A, we need to show that E satisfies a set of basic properties. We begin by showing that all the transitions in E are finite (Lemma 10.3.4). We then show that E is a TIOA, and finally that $A \times E$ is internally progressive. We are then able to show in Theorem 10.3.7 that $A \times E$ is a process.

We begin by showing that the "functions" described in Figures 10-10–10-12 in fact terminate. First, consider the do-bcast function (Figure 10-12). The function do-bcast takes one parameter, s, the state of virtual node v. The do-bcast function terminates when the state represented by s is the last state in an execution fragment that concludes with a recv event (with no bcast event that follows it). The argument depends on the requirement that a bcast event occur immediately in response to a recv event; hence if the do-bcast function is used to extend an execution of v, it is applicable only when there is an immediately preceding recv event.

Lemma 10.3.1. *Let $v \in I_V$ be a virtual node and γ an execution of v. Let $s = \ell state(\gamma)$ and $t = \ell time(\gamma)$. If a $\mathsf{recv}_{v,*}$ event occurs at time t in γ and no $\mathsf{bcast}_{v,*}$ event occurs at time t in γ after the* recv *event, then $\mathsf{do\text{-}bcast}(s)_v$ terminates.*

Proof. The key requirement is to show that the initial **while** loop terminates. At the beginning of the loop the virtual node v is in state s. The loop terminates as soon as there exists some $\textit{next-vn-msg} \in \textit{msgs}_\perp$ and $\textit{out-req} \in \textit{CM-names}_\perp$ such that virtual node v is enabled in state s to perform a $\mathsf{bcast}(\textit{next-vn-msg}, \textit{out-req})_{v,*}$ transition.

Assume for the sake of contradiction that do-bcast does not terminate. We claim, then, that the infinite **while** loop continues to add locally-controlled events to execution fragment *exec*: Since the automaton at v is a process, and since γ contains a recv event at time t with no following bcast event, there must be a bcast event prior to any time-passage event, by Definition 8.1.2. Since the automaton at v is time-passage enabled, and yet time-passage is not enabled since there has been no bcast event, there must be some enabled locally-controlled event. Thus the **while** loop continues to add an infinite number of locally-controlled events to the execution $\gamma.\textit{exec}$.

Since the process at v is internally-progressive, we know that only a finite number of locally-controlled events can occur at time t in any execution $\gamma.\textit{exec}$, assuming only a finite number of input events occur at time t. Notice, however, that the construction never adds an input event to $\gamma.\textit{exec}$. This contradicts our previous statement that $\gamma.\textit{exec}$ contains an infinite number of locally-controlled events at time t, implying that the do-bcast function terminates. □

Next we show that the do-recv function (Figure 10-12) terminates. Specifically, if s represents the last state in an execution γ where every recv event has a matching

bcast event, then do-recv$(s, \ldots)$ terminates. (In fact, we are only interested in bcast and recv events that occur at the last time in γ.) The argument depends primarily on the fact that a TIOA is time-passage enabled and internally progressive.

Lemma 10.3.2. *Let $v \in I_V$ be a virtual node and γ an execution of v. Let $s = \ell state(\gamma)$ and $t' = \ell time(\gamma)$. Assume that every $\mathsf{recv}_{v,*}$ event at time t' in γ is followed by a $\mathsf{bcast}_{v,*}$ event at time t' in γ. Then do-recv$(s, \ldots)_v$ terminates.*

Proof. We argue that the **while** loop which adds time-passage events terminates. First, notice that it is always possible on line 612 (Figure 10-12) to find an appropriate internal action a: by the fact that $t = 0$, we know that time-passage is not enabled; however, the automaton at v is time-passage enabled, implying that some locally-controlled action is enabled; this action cannot be the output action bcast, as a bcast can happen only in response to a recv event, and we have assumed that every recv event already has a matching bcast event.

Next, notice that each iteration of the **while** loop adds a locally-controlled event to an interval of time no larger than $RndLength_V$. Since there are only a finite number of input events during this interval, we know that there are only a finite number of internal events, since the automaton at v is internally progressive. Therefore we conclude that the **while** loop terminates. □

We are now able to show that the two calculation functions, calculate-status and calculate-state, terminate. The argument is a straightforward examination of the loop constructs used, along with an application of Lemmas 10.3.1 and 10.3.2.

Lemma 10.3.3. *The functions* calculate-status$(\ldots)$ *and* calculate-state$(\ldots, \textit{temp-state})_v$ *(in Figures 10-10 and 10-11) terminate under the condition that temp-state is the initial state of virtual node v.*

Proof. First, it is relatively straightforward to see that the calculate-status terminates: the round counter r is decremented during each iteration of the loop, and hence eventually $r \leq \textit{temp-last-reset} + 1$, at which point the loop terminates.

Second, we examine the calculate-state function. The main loop iterates the round counter r from *temp-last-reset*$+1$ up to *temp-rnd*. It remains only to show that its use of do-bcast and do-recv themselves terminate. These two functions are always called in pairs, thus ensuring that the requisite conditions of Lemmas 10.3.1 and 10.3.2 hold.

In the first iteration of the main **while** loop, $r = \textit{temp-last-reset} + 1$, and hence no broadcast is included in either branch of the **if** clause. (See lines 551 and 561.) Thus in the first iteration, the functions calls do-recv, where the state *temp-state* is the initial state of virtual node v, i.e., the last state in an empty execution *exec*. We conclude, then, that the call to do-recv terminates, by Lemma 10.3.2, as there are no recv events in the empty execution *exec*. After the call to do-recv, and after *e-frag-rcv* is concatenated to *exec*, notice that the last event in *exec* is a recv event; this property is maintained throughout the rest of the function execution.

In each later iteration of the main **while** loop, the calculate-state function invokes both do-bcast and do-recv. Since *temp-state* is the last state in *exec*, and the last event in *exec* is a $\mathsf{recv}_{v,*}$ event, we can conclude that the do-bcast event terminates

by Lemma 10.3.1. At this point, *temp-state* is the last state of *exec*.*e-frag-bc*, and the last event in *e-frag-bc* is a $\mathsf{bcast}_{v,*}$ event. Thus, by Lemma 10.3.2, the do-recv event terminates. Again, after the call to do-recv and after the execution fragments are concatenated to *exec*, the last event in *exec* is a recv event, maintaining the requisite property.

Thus in each iteration, all the calls to do-bcast and do-recv terminate, and thus the calculate-state function terminates. □

We are now able to show that each transition of the emulator algorithm terminates, that is, that none of the transitions is infinite. The only place where problems may arise is during the recv transition during the client phase, when the algorithm invokes the calculate-status, calculate-state, and do-bcast functions. In each case, we show that the previous lemmas indicate that the function terminates, leading to the desired conclusion.

Lemma 10.3.4. *Each transition in emulator E is finite.*

Proof. Since no loop constructs appear in the emulator algorithm, the only potential problems arise in the use of the various sub-functions. The calculate-status function is invoked on line 235, Figure 10-5; by Lemma 10.3.3 we know that it terminates, implying that it is a finite calculation.

Next, consider the calculate-state function that is invoked on line 236, Figure 10-5 (in order to calculate the last state of the proposed execution of v). Notice that the variable *last-good-state* is initially the initial state of virtual node v, and is never changed during an execution of the algorithm. Thus, we can conclude by Lemma 10.3.3 that calculate-state terminates, implying that it is a finite calculation.

Finally, notice that the do-bcast function is used in the emulation algorithm on line 237, Figure 10-5 (in order to calculate the next message that virtual node v should send). In this case, *temp-state* is the state calculated by calculate-state on the previous line (line 236, Figure 10-5). Recall that calculate-state also returns an execution which has *temp-state* as its last state, and which has $\mathsf{recv}_{v,*}$ as its last action. Thus by Lemma 10.3.1 we conclude that the do-bcast function call terminates. □

We can now show that emulator E is, in fact, a TIOA. In particular, this requires showing that E is time-passage enabled.

Lemma 10.3.5. *Emulator E is a TIOA.*

Proof. We have already shown in Lemma 10.3.4 that each transition in E is finite. We need to show that E is input enabled and time-passage enabled. The only syntactic restriction in inputs is the **where** clauses that restrict the recv_v transitions, depending on the phase. However, for each phase, there is exactly one recv transition enabled.

We now argue that emulator E is time-passage enabled. The trajectories disable time passage in two circumstances:

- $E(v).$*do-client-recv* $=$ true and $E(v).$*failed* $=$ false: In this case, under appropriate settings of *vnm-out* and *vnCD-out*, the $\mathsf{vn\text{-}client\text{-}output}(\mathit{vnm\text{-}out}, \mathit{vnCD\text{-}out})_v$ event is enabled. This transition resets $E(v).$*do-client-recv* to false.

- $E(v).$*do-net-bcast* = true and $E(v).$*failed* = false: In this case, we can assume that *do-client-recv* = false; otherwise, we fall into the previous case. In this case, under appropriate settings of m and cm, the $\mathsf{bcast}(m, cm)_v$ transition is enabled, which resets *do-net-bcast* to false.

As a result, emulator E is time-passage enabled, and hence E is a TIOA. □

The last fact we need to show about the emulator is that, when composed with another process, i.e., a client, the resulting composed automaton is a process. We show this fact in two steps: first we show that the composed automaton is internally progressive; then we show that the automaton meets the other requirements of a process.

Lemma 10.3.6. *If A is a process, then $A \times E$ is internally progressive.*

Proof. Consider an execution in which each finite interval of time has a finite number of input actions. We begin by examining the locally-controlled actions of E, and argue that each finite interval has a finite number locally-controlled actions by E. Assume for the sake of contradiction that this is not the case. We consider each of the possible locally-controlled actions of E in turn:

- bcast_v: Assume there are an infinite number of bcast_v events in some finite interval of time, for some $v \in I_V$. The precondition for the transition is that *do-net-bcast* = true (line 161, Figure 10-4), and in the effect, *do-net-bcast* is reset to false. Moreover, the only transition which sets *do-net-bcast* to true is the recv_v input event. Thus, we conclude that for each bcast_v event, there is an intervening recv_v event that precedes it. This implies that there are also an infinite number of recv_v input events in the same finite interval of time, contradicting our assumption.

- bcast_0: Assume there are an infinite number of bcast_0 events in some finite interval of time. The argument in terms of bcast_0 is almost equivalent, except with respect to the *phase* flag (rather than the *do-net-bcast* flag). Specifically, a bcast_0 occurs only when *phase* = out, and its effects reset *phase* to in. The only event that sets *phase* to out is a recv_0 input event. Thus we conclude that for each bcast_0 event, there is a preceding recv_0 event, implying a contradiction.

- $\mathsf{vn\text{-}client\text{-}output}_v$: Assume there are an infinite number of $\mathsf{vn\text{-}client\text{-}output}_v$ events in some finite interval of time, for some $v \in I_V$. Again, the argument is almost equivalent, except with respect to the do-client-recv flag. The precondition of each vn-client-output transition is that *do-client-recv* = true, and this flag is reset by the transition's effects. Moreover, *do-client-recv* is set to true only by the recv_v input event. Thus we conclude that for each $\mathsf{vn\text{-}client\text{-}output}_v$ event, there is a preceding recv_v event, implying a contradiction.

- recv_*: Assume there are an infinite number of recv_* events in some finite interval of time. The precondition of the recv_* event requires that $inVNs = V$, and the set $inVNs$ is reset to $\emptyset$ by the transition's effects. The only transition which

adds $v \in I_V$ to *inVNs* is the $\mathsf{vn\text{-}client\text{-}output}_v$ event. Thus, if there are an infinite number of recv_* events in a finite interval of time, there must also be an infinite number of $\mathsf{vn\text{-}client\text{-}output}_v$ events for some $v \in I_V$, contradicting the previous claim.

Thus we conclude that if there are a finite number of input events in some finite interval, then there are also a finite number of events controlled by automaton E. Finally, we examine automaton A. Since automaton A is a process, we know that it is internally progressive. Since for every finite interval there are a finite number of events controlled by E, we can conclude that for A, there are a finite number of input events. Thus, we conclude that there are also a finite number of events controlled by A. Overall, then, we conclude that automaton $A \times E$ is internally progressive. □

Finally, we present the main claim of the section: when emulator E is composed with a process, the resulting composed automaton is also a process.

Theorem 10.3.7. *If A is a process, then $A \times E$ is a process.*

Proof. First, since A is a TIOA and E is a TIOA (Lemma 10.3.5), we conclude that $A \times E$ is a TIOA. Next, we conclude by Lemma 10.3.6 that $A \times E$ is internally progressive. It is immediately clear that $A \times E$ satisfies the static restrictions of a process: Its input actions are recv_p for port $p \in \textit{process-ports}_B$, and fail; its output actions are bcast_p for port $p \in \textit{process-ports}_B$. (Recall that $\textit{process-ports}_B = I_V \cup \{0\}$.) Additionally, all of the variables of E are discrete; since A is a process, all of the variables of A are discrete. Thus all the variables of $A \times E$ are discrete. It thus remains only to show that $A \times E$ satisfies the dynamic restrictions of a process: immediate response and failure.

We first notice that each bcast_v event follows a recv_v event, with no intervening bcast_v event: The precondition for the bcast_v event requires that $E(v).\textit{do-net-bcast} = \mathsf{true}$, and the flag is reset during the transition's effects. The only event which sets the $E(v).\textit{do-net-bcast}$ flag true is recv_v, implying that each bcast_v event follows a recv_v event.

By essentially the same argument, a bcast_0 event follows a recv_0 event, with no intervening bcast_0 event: the precondition of the bcast_0 event requires that $\textit{phase} = \mathsf{out}$; this flag is set only by a recv_0 event.

We next show that each recv_v event is followed immediately by a bcast_v event or a fail event. The recv_v event sets the $E(v).\textit{do-net-bcast}$ flag to true. According to the restrictions on the trajectories, time cannot continue until either the flag is reset or $E(v).\textit{failed} = \mathsf{true}$. Thus, if time passes further, either a bcast_v event or a fail event immediately follows the recv_v event.

By essentially the same argument, each recv_0 event is followed immediately by a bcast_0 event. The restriction on trajectories ensures that time cannot continue until either $\textit{phase} = \mathsf{in}$ or $\textit{failed} = \mathsf{true}$. Thus, if time passes further, either a bcast_0 event or a fail event immediately follows the recv_0 event.

We next consider the failure property: no locally-controlled event occurs after a fail event. Notice that in each automaton described, the fail event sets a flag *failed*

to true; each locally-controlled action is only enabled when *failed* = false. Since automaton A is also a process, the same failure property holds immediately, and no locally-controlled events in A occurs after a fail event.

Thus we conclude that $A \times E$ satisfies all the requirements for a process. □

Chapter 11

Proof of Correctness

In this chapter we show that the protocol presented in Chapter 10 implements a virtual infrastructure system. Fix execution α to be an infinite length execution of the emulator system (which is equivalent to an infinite time execution, see Lemma 8.1.31) in which the basic broadcast service satisfies eventual collision freedom. (We are interested only in executions that satisfy eventual collision freedom.) Our main goal in this chapter is to construct an execution γ of the virtual infrastructure system $VS(I_V, msgs_V, P_V, A_V, loc_V)$. We will then show that the execution γ satisfies the appropriate liveness properties with respect to eventual collision freedom with good advice, collision detector accuracy, and contention management. The following is an outline of this chapter:

- Section 11.1 contains an overview of the proof.
- Section 11.2 describes the basic round structure.
- Section 11.3 presents lemmas describing which nodes are participating in the emulation and which are not. In addition, we define an "updown" sequence which describes when a virtual node is up and down.
- Section 11.4 analyzes the effects of a virtual node being reset.
- Section 11.5 defines good and bad rounds, as well as green and red rounds. It then proves some basic properties about non-red rounds, particularly related to the agreement among ballots.
- Section 11.6 proves a series of lemmas related to the calculate-status helper function.
- Section 11.7 proves a series of lemmas related to the calculate-state helper function. A major conclusion of this section is that in a good round, the emulators of a virtual node agree on an execution.
- Section 11.8 shows that eventually, after the basic system stabilizes, every round is a green round. The claims in this section depend on the fact that the underlying execution guaranteeing eventual collision freedom.

- Section 11.9 constructs an execution γ_v for every $v \in I_V$.
- Section 11.10 constructs an execution γ_i for every $i \in I_B$, as well as executions for the virtual contention managers.
- Section 11.11 shows that the traces defined in Sections 11.9 and 11.10 are consistent with a virtual broadcast service that guarantees integrity.
- Section 11.12 defines the virtual collision detector rule. Section 11.12.2 shows that the virtual collision detector rule is complete. Section 11.12.3 shows that the virtual collision detector rule is eventually accurate.
- Section 11.13 constructs an execution γ_{bc} of the virtual broadcast service.
- Section 11.14 pastes together all the sub-executions into a single execution γ of the virtual infrastructure system, thus concluding the proof.
- Section 11.15 shows that execution γ of the virtual system satisfies the eventual collision freedom with good advice (ECF[GA]) property. This section depends on the underlying execution guaranteeing eventual collision freedom.
- Section 11.16 discusses under what conditions a virtual node is failed, and under what conditions a virtual node is not failed. This section depends on the underlying execution guaranteeing eventual collision freedom.

11.1 Overview of the Proof

Recall that an emulator system contains the following automata:

- remapped processes $P(A_B, I_B)$, where each process is an automaton $A_V(i) \times E$, for some $i \in I_B$,
- broadcast service $\mathcal{B}_B$, and
- contention managers $\{CM_v : v \in I_V\} \cup \{CM_{\mathsf{client}}\}$.

Throughout this section, as a slight abuse of terminology, we refer to the process $\mathsf{Remap}(A(i)_B, i)$ associated with name $i \in I_B$ as **node** i. By a similar abuse of terminology, when talking about the virtual infrastructure system, we refer to the process $\mathsf{Remap}(A(i)_V, i)$ associated with name $i \in I_B$ as **client** i, and the process $\mathsf{Remap}(A(v)_V, v)$ associated with name $v \in I_V$ as **virtual node** v.

Recall that the automaton $A_B(i)$ itself consists of two composed automata: the client automaton, $A_V(i)$, and the emulator E. Further, recall that the emulator is the composition of individual emulators for each virtual node $v \in I_V$, which we refer to as $E(v)$, and a multiplexer, which we refer to as $E(\mathit{multiplexer})$.

The main part of the proof involves constructing an execution γ_v of each virtual node $v \in I_V$. We extract from execution α the individual executions of each of the clients; we then construct an execution for each of the virtual contention managers.

Finally, we use the traces of these constructed executions to produce a trace of the broadcast service, and from there construct an execution of the broadcast service. The key lemma in this part of the proof, Lemma 11.13.1, shows that the resulting construction is in fact an execution of the broadcast service. Once we have constructed individual executions for each of the components in the virtual infrastructure system, we conclude by pasting the various executions together to form γ, which we then show has the desired properties.

The main technical challenge, then, is correctly constructing the individual executions γ_v so that the resulting execution-pasting works. Consider some particular virtual node v. We begin by identifying in which rounds virtual node v is "up," i.e., not failed, and which rounds virtual node v is "down," i.e., failed. Next, we show how to extract from the state of the replicas a set of executions for virtual node v that are consistent with execution α: We define $exec(r_1, r_2, i)_v$ as the preferred execution fragment for rounds $[r_1, r_2]$ according to replica i for virtual node v from round r_1 to round r_2. This execution is constructed in a two-step process: first, the calculate-status function is used to determine from the $ballot_{i,v}$ data structure which virtual rounds are "good" and which virtual rounds are "bad." Then, the calculate-status function is used to produce the actual execution fragment. We define $execs(r_1, r_2)_v$ to be the set of all possible executions for all "participating" replicas i.

We need to show that each execution in $execs(r_1, r_2)_v$ has certain properties. First and foremost, whenever there is some externally-visible behavior, the executions should agree. Thus, a key lemma (Corollary 11.7.12), shows that if any node receives a message from virtual node v in some round $r \in [r_1, r_2]$, then $|execs(r_1, r_2)_v| = 1$, i.e., there is only one possible execution among all the replicas. This ensures that when we later construct the execution of the virtual broadcast service, the "integrity" property will be maintained. Other externally visible behavior relates to the detection of collisions: Lemma 11.12.10 shows that if any execution in $execs(r_1, r_2)_v$ of virtual node v includes the broadcast by v of a message in some round $r \in [r_1, r_2]$, and if client i does not receive that message in γ_i in round r, then client i receives a collision in γ_i in round r, where γ_i is the execution constructed for client i in the virtual infrastructure system. This leads to the completeness property.

Once we have constructed the various execution fragments, we can concatenate them together—with appropriate time passage events to simulate failed rounds—to arrive at an infinite execution γ_v. Finally, we show that the construction of γ_v results in certain liveness properties. The key lemma, in this case, is Lemma 11.8.12, which shows that every virtual round $\geq r_{gst}$, where r_{gst} is some round in the virtual infrastructure system, is a "green" round, meaning that the replicas successfully agree on the next portion of the virtual node execution. As a result, we can conclude that the virtual infrastructure system guarantees eventual collision freedom with good advice (ECF[GA]), eventual accuracy, and appropriate contention management.

11.2 Rounds, Phases, and Other Preliminary Definitions

We begin with some preliminary lemmas and definitions that elucidate virtual rounds, phases, and other terminology that will be used throughout the proof. We focus in particular on how rounds are designated and how rounds are counted. Throughout this proof, we occasionally refer to a sequence of rounds: the set of rounds $[r_1, r_2]$, where $r_2 \geq r_1$, is defined as $\{r \in \mathbb{N}_0 \mid r_1 \leq r \leq r_2\}$.

11.2.1 Round Length

Recall (from Chapter 9) that the $RndLength_V$, as specified in the virtual infrastructure system specification, is in fact equal to $10 + \mathsf{SMAX}$, which is the number of phases in the emulator protocol. Since a virtual round, by assumption, is the time interval $[(r-1) \cdot RndLength_B, r \cdot RndLength_V]$, and a basic round is, by assumption, the time interval $[r-1, r]$, we can conclude:

Lemma 11.2.1. ***Virtual round*** *r is equivalent to basic rounds $[(r-1)(RndLength_V)+1, (r)(RndLength_V)]$.*

There is, therefore, a straightforward translation between virtual rounds and basic rounds. As the emulator protocol operates in "phases," and since in most cases (other than the unscheduled-ballot phase) each phase corresponds to exactly one basic round, for the sake of clarity we will occasionally to use the term "phase" to refer to a basic round, thus distinguishing it from a virtual round. We will use the full terms "basic round" and "virtual round" when it is otherwise unclear.

We often refer to some particular phase of the emulator protocol associated with virtual round r. For example, when we refer to the vn phase associated with virtual round r, we are referring to the basic round $(r-1)(RndLength_V) + 2$, since the vn phase is the second phase in the virtual round.

Definition 11.2.2. *The following is the translation between phases of virtual round r and basic rounds:*

client *phase of round r:*	$(r-1)(RndLength_V) + 1$
vn *phase of round r:*	$(r-1)(RndLength_V) + 2$
scheduled-ballot *phase of round r:*	$(r-1)(RndLength_V) + 3$
scheduled-veto-1 *phase of round r:*	$(r-1)(RndLength_V) + 4$
scheduled-veto-2 *phase of round r:*	$(r-1)(RndLength_V) + 5$
unscheduled-ballot *phases of round r:*	$[(r-1)(RndLength_V) + 6,$ $(r-1)(RndLength_V) + \mathsf{SMAX} + 5]$
unscheduled-veto-1 *phase of round r:*	$(r-1)(RndLength_V) + \mathsf{SMAX} + 6$
unscheduled-veto-2 *phase of round r:*	$(r-1)(RndLength_V) + \mathsf{SMAX} + 7$
join *phase of round r:*	$(r-1)(RndLength_V) + \mathsf{SMAX} + 8$
join-ack *phase of round r:*	$(r-1)(RndLength_V) + \mathsf{SMAX} + 9$
join-veto *phase of round r:*	$(r-1)(RndLength_V) + \mathsf{SMAX} + 10$

Notice that each phase of the emulator protocol corresponds to exactly one basic round in the virtual round, with the exception of the unscheduled-ballot-phase, which lasts for SMAX basic rounds.

11.2.2 Emulator State at the Beginning and End of a Round

We next need some terminology to refer to the state of the emulator process at various points in the execution. In particular, we will be interested in the state of the emulator at the end of each basic round, and at the end of each virtual round. (Notice that it makes sense to talk about the end of a virtual round in α, since virtual rounds are simply defined by intervals of time, not events; hence the virtual round ends when the last basic round associated with the virtual round ends.) Since rounds are simply defined in terms of time, however, the state of a process at the "end of a round" is so far ill-defined, as multiple events may modify its state at that exact instant in time. Throughout this proof, then, when we refer to the state at the end of a round, we mean the state of a node immediately after the receive event associated with that round:

Definition 11.2.3.

- *The state of node $i \in I_B$ at the **end of basic round** r is the state of i immediately after the round r receive event.*
- *The state of node $i \in I_B$ at the **end of virtual round** r is the state of i at the end of the last basic round in virtual round r.*
- *The state of node $i \in I_B$ at the **beginning of basic round** r is the state of i at the end of basic round $r-1$, or the initial state if $r = 1$.*
- *The state of node $i \in I_B$ at the **beginning of virtual round** r is the state of i at the end of virtual round $r-1$, or the initial state if $r = 1$.*

11.2.3 The Current Round

The emulator process for virtual node v keeps track of the current virtual round in the state variable $E(v).rnd_i$. (Recall that the notation $E(v).rnd_i$ refers to the *rnd* state component of the individual virtual node emulator $E(v)$ associated with node i.) By simple counting, it is immediately clearly that the emulator correctly counts virtual rounds:

Lemma 11.2.4. *For all $i \in I_B$, if i has not failed by the end of basic round r_a, where r_a is part of virtual round r, then $E(v).rnd_i = r$ at the end of basic round r_a.*

Proof. This lemma follows immediately from the fact that $RndLength_V = 10 + \mathsf{SMAX}$, and that there are exactly $10 + \mathsf{SMAX}$ basic rounds in the protocol: every client phase increments $E(v).rnd_i$ by 1. □

Notice that this lemma says nothing about the actual behavior of the virtual node; the beginning and end of a virtual round are defined simply by the times at which the round starts and finishes, which is itself derived from $RndLength_V$.

Lemma 11.2.5. *At the beginning of every basic round $r_a \in \mathbb{N}$, for all $i \in I_B$, for virtual node $v \in I_V$, $E(v).rnd_i = \lceil r - 1/RndLength_V \rceil$.*

Proof. There are $RndLength_V = 10 + \mathsf{SMAX}$ basic rounds in the phase progression: one basic round for each of:

1. client,
2. vn,
3. scheduled-ballot,
4. scheduled-veto-1,
5. scheduled-veto-1,
6. unscheduled-veto-1,
7. unscheduled-veto-2,
8. join,
9. join-ack, and
10. join-veto;

Additionally, there are SMAX basic rounds for the unscheduled-ballot phase. The variable $E(v).rnd_i$ is incremented in the client phase, and initially $E(v).rnd_i = 0$ at the beginning of the basic round 1. The expression $\lceil r-1/RndLength_V \rceil$ is incremented when $r = 1, RndLength_V+1, 2(RndLength_V)+1, 3(RndLength_V)+1, \ldots$; each of these basic rounds is a client phase. □

Corollary 11.2.6. *At the beginning of every basic round r, for all $i, j \in I_B$, $E(v).rnd_i = E(v).rnd_j$.*

Proof. Immediate as a result of Lemma 11.2.5. □

We also prove the same claim for the multiplexer:

Lemma 11.2.7. *For every $i \in I_B$, if i has not failed by the end of basic round r_a, then $E(multiplexer).rnd_i = r_a$ at the end of basic round r_a.*

Proof. This lemma follows immediately from the fact that $E(multiplexer).rnd_i$ begins at zero, and is incremented with each $\mathsf{recv}_{i,0}$ event, i.e., with every recv event by the broadcast service. Since i does not fail prior to the end of r_a, there is a $\mathsf{recv}_{i,0}$ event in every basic round $\leq r_a$. □

11.2.4 Scheduled and Unscheduled Rounds

Recall that we say virtual node is scheduled in virtual round r if $v \in schedule[r \bmod \mathsf{SMAX}]$; otherwise, we say that v is unscheduled. For node $i \in I_B$ and virtual node $v \in I_V$, the variable $E(v).scheduled_i$ records whether virtual node v is scheduled in round r.

Lemma 11.2.8. *Assume node $i \in I_B$ has not failed by the end of basic round r_a, and that round r_a is part of virtual round r.*

Node $i \in I_B$ has $E(v).scheduled_i =$ true at the end of basic round r_a in virtual round r if and only if virtual node v is scheduled for virtual round r.

Proof. Since i has not failed by the end of r_a, we know that a recv event for the round r client phase occurs, as the client phase is the first phase in r. At the beginning of the client phase, that is, the end of round $r-1$, $E(v).rnd_i = r-1$, as per Lemma 11.2.4. During the client phase, first $E(v).rnd_i$ is incremented to r, and then the flag $E(v).scheduled_i$ is set if and only if $v \in schedule(E(v).rnd_i \mod \mathsf{SMAX})$, that is, if and only if virtual node v is scheduled for virtual round r. The $E(v).scheduled_i$ flag is not modified during any other phase of round r prior to the end of round r_a, since node i does not fail in a basic round $\leq r_a$. □

We can thus conclude that any two nodes share the same value for the *scheduled* flag for the same v at the end of each basic round in a virtual round:

Corollary 11.2.9. *Let r_a and r_b be basic rounds that are part of the same virtual round. Let $i, j \in I_B$ be two nodes that have not failed by the end of r_a and r_b, respectively. Then at the end of round r_a, $E(v).scheduled_i = E(v).scheduled_j$, at the end of round r_b.*

Proof. Follows immediately from Lemma 11.2.8: if r_a and r_b are part of virtual round r, then: if v is scheduled, then both equal true; if v is not scheduled, then both equal false. □

11.3 Participating in a Virtual Node Emulation

Our goal in this section is to determine when a node i "participates" in emulating a virtual node, and when it does not. We will then show how to construct an "updown" sequence that indicates in which rounds a virtual node v is up, and in which rounds virtual node v is down. For every consecutive sequence of rounds in which virtual node v is up, we will show, in the following sections, how to construct an appropriate execution segment.

In part, this section implies that the "join protocol" works as expected, in that it discusses how nodes join the emulation and what can be guaranteed of nodes participating in the emulation.

11.3.1 Preliminary Definitions

Before discussing node participation, we first need some terminology relating to the schedule. Recall that in each virtual round, the emulator examines the *schedule* to determine whether to simulate a broadcast by the virtual node. We therefore define the notion of "scheduled" as follows:

Definition 11.3.1. *We say that virtual node v is **scheduled** in virtual round r if $v \in schedule[r \mod \mathsf{SMAX}]$. Otherwise, we say that virtual node v is **unscheduled** in virtual round r.*

When virtual node v is scheduled for some virtual round r, the emulator for v behaves differently than when v is unscheduled. Specifically, the emulator for virtual

node v uses the "scheduled agreement instance," rather than the "unscheduled agreement instance." Thus, we define the "last agreement phase" as follows, depending on whether virtual node v is scheduled:

Definition 11.3.2. *For virtual node $v \in I_V$ and virtual round $r > 0$, we define the **last agreement phase** of v in virtual round r as follows:*

- *If v is scheduled in round r, then the last agreement phase is the* scheduled-veto-2 *phase.*
- *If v is unscheduled in round r, then the last agreement phase is the* unscheduled-veto-2 *phase.*

We can now determine whether a replica is participating in the emulation of virtual node v for some given virtual round r. Specifically, a replica is participating in the emulation if it has completed the "join protocol" prior to virtual round r and does not fail until the end of the appropriate agreement instance:

Definition 11.3.3. *We say that $i \in I_B$ **participates** in virtual node v in virtual round r if $E(v).joined_i =$ true and $E(v).failed_i =$ false at the beginning of virtual round r, and through the end of the last agreement phase of virtual round r.*

A related concept is when a given replica "begins" a virtual round; a replica may begin a virtual round, but fail prior to reaching the end of the agreement instance. Similarly, we may be interested in the case where a replica "completes" a virtual round r, which we define to be the same as the case when a virtual round "begins" the following round $r+1$; in this case, by the end of virtual round r, i.e., the beginning of virtual round $r + 1$, i has completed the join protocol and not failed.

Definition 11.3.4. *We say that $i \in I_B$ **begins** virtual round r for virtual node v if $E(v).joined_i =$ true and $E(v).failed_i =$ false at the beginning of virtual round r. We say that $i \in I_B$ **completes** virtual round r for virtual node v if it begins round $r + 1$.*

Notice that it is immediate from Definition 11.3.3 that if i participates in v in virtual round r, then i also begins r for v. On the other hand, a node i may complete a virtual round r for v, and yet not participate in v in round r; for example, a node may not begin round r for v, but may join v in round r and complete round r for v.

Next, we define what it means for a virtual node to be reset in a given virtual round. Specifically, we say that a virtual node is reset when the replicas simulate a reset event in the virtual system:

Definition 11.3.5. *We say that node $i \in I_B$ **executes a reset** of virtual node v in virtual round r if automaton $E(v)_i$ executes lines 482–488 (Figure 10-9) in virtual round r.*

Next, we define what it means for a replica to join the emulation. Specifically, we say that a replica joins the emulation when it executes the appropriate lines of code:

Definition 11.3.6. *We say that node $i \in I_B$ **joins** virtual node v in virtual round r if automaton $E(v)_i$ executes lines 462–463 (Figure 10-9) in virtual round r.*

11.3.2 Defining Up and Down

At this point, we can define what it means for a virtual node to be up or down in a given virtual round. Specifically, we say that a virtual node v is down in two different cases: first, v is down if there are no replicas that remain involved in the emulation of v, i.e., there are no replicas that have joined, not failed, and not left the region at the end of the round; in this case, the virtual node is down; second, v is down if some replica executes the reset protocol.

Definition 11.3.7. *We say that virtual node $v \in I_V$ is **down** in virtual round $r > 0$ of execution α in either of the following two cases:*

1. *There exists some $i \in I_B$ such that i executes a reset of v during virtual round r.*
2. *For all $i \in I_B$, i does not complete virtual round r for v.*

*We say that virtual node $v \in I_V$ is **up** in virtual round r of execution α if it is not down.*

11.3.3 Properties of Participating Nodes

We now prove some basic properties about nodes participating in a given round for a given virtual node. We begin with a basic lemma on when a node can join the emulation; the main point of this lemma is to show that if a node i has joined and not failed at any point in a virtual round prior to the join phase, then i must have begun the virtual round, since a node cannot join the emulation prior to the join phase.

Lemma 11.3.8. *Let $r > 0$ be a virtual round, and $v \in I_V$ a virtual node. If node $i \in I_B$ has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the end of any basic round prior to the* join *phase of round r, then i begins round r for v.*

Proof. First, since the basic system processes are non-recoverable, we know that $E(v).failed_i = \mathsf{false}$ at the beginning of virtual round r, since $E(v).failed_i = \mathsf{false}$ after the beginning of virtual round r.

Assume, then, for the sake of contradiction that $E(v).joined_i = \mathsf{false}$ at the beginning of virtual round r. The first opportunity for node i to set $E(v).joined_i = \mathsf{true}$ is *line* 463, which occurs in the join-ack phase. Thus, $E(v).joined_i = \mathsf{false}$ through the beginning of the join phase, contradicting our assumption. □

Next, we can conclude from Lemma 11.3.8 that any non-failed node i that has $E(v).joined_i = \mathsf{true}$ through the end of the last agreement phase must also participate in the virtual round, by Definition 11.3.3. This gives us a characterization of participating nodes for a given virtual node and virtual round: the set of nodes that have joined and not failed at the end of the last agreement phase. In fact, we show a slightly stronger result: if a node has joined and not failed at the end of either veto-2 phase, then it has joined and not failed from the beginning of the round up until that point:

Lemma 11.3.9. *Let $r > 0$ be a virtual round, and $v \in I_V$ a virtual node. If node $i \in I_B$ has $E(v).joined_i =$ true and $E(v).failed_i =$ false at the end of either (1) scheduled-veto-2 phase or (2) unscheduled-veto-2 phase of round r, then, in either case, i has $E(v).joined_i =$ true and $E(v).failed_i =$ false from the beginning of virtual round r through the end of the specified* veto-2 *phase.*

Proof. We can conclude from Lemma 11.3.8 that node i begins round r. Moreover, if $E(v).joined_i =$ false in any phase prior to—or including—the specified veto-2 phase, then $E(v).joined_i =$ false at the end of the veto-2 phase, contradicting our assumption, as there is no opportunity prior to the join-ack phase for i to reset $E(v).joined_i$ to true. Also, since processes in the basic system are non-recoverable, we know that $E(v).failed_i =$ false in all rounds prior to the last agreement phase. Thus, node i has $E(v).joined_i =$ true and $E(v).failed_i =$ false from the beginning of round r through the end of both the scheduled-veto-2 and the unscheduled-veto-2 phase. □

Finally, we prove that the definition of a virtual node being "up" captures something about the real status of the virtual node. Specifically, we show that if the virtual node is up, then there is at least some $i \in I_B$ that is participating in the round for v. In fact, we show a slightly stronger result, specifically, that some replica has joined and not failed at least through the end of the unscheduled-veto-2 phase. (A participating node is only guaranteed to be joined and not failed through the end of the last agreement phase, which for a virtual node scheduled in round r may be the earlier scheduled-veto-2 phase.)

Lemma 11.3.10. *Let $r > 0$ be a virtual round, and $v \in I_V$ be a virtual node that is up in virtual round r. Then there exists some node $i \in I_B$ such that (1) i begins round r for virtual node v, and (2) $E(v).joined_j =$ true and $E(v).failed_j =$ false through the end of the* join *phase of round r.*

Proof. First, if there exists any node $i \in I_B$ such that $E(v).joined_i =$ true and $E(v).joined_i =$ false at the end of the unscheduled-veto-2 round of round r, then i must begin round r, by Lemma 11.3.9, satisfying our claim.

Assume, then, that at the beginning of the join-ack phase, there is no node $i \in I_B$ that has $E(v).joined_i =$ true and $E(v).failed_i =$ false. In this case, the condition on line 445 (Figure 10-9) is not satisfied by any node during the join phase, and thus no join-ack messages are broadcast. Thus, no node $i \in I_B$ joins v in round r. As a result, virtual node v must be down in round r: either some node $j \in I_B$ executes the reset protocol, setting $E(v).joined_j =$ true, or at the end of the virtual round there are no nodes $j \in I_B$ such that $E(v).joined_j =$ true. Either way, v is down in round r, contradicting the assumption that v is up in round r. □

11.3.4 Delineating an Execution

We now define what it means for virtual rounds r_1 and r_2 to "delineate" an execution:

Definition 11.3.11. *Let $r_1 \in \mathbb{N}$, $r_2 \in \mathbb{N} \cup \{\infty\}$, $r_1 \leq r_2$. We say that $\langle r_1, r_2 \rangle$ is **an epoch**, delineated by $[r_1, r_2]$, of virtual node $v \in I_V$ in α if the following properties*

hold: (1) if $r_1 > 1$, then virtual node v is down in virtual round $r_1 - 1$; (2) virtual node v is up in virtual rounds $[r_1, r_2]$. We say that an epoch $\langle r_1, r_2 \rangle$ is finite if $r_2 \neq \infty$.

Notice that we do not assume that v is down in round $r_2 + 1$; that is, these are not necessarily maximal length sequences of rounds in which v is up.

If $\langle r_1, r_2 \rangle$ delineates an epoch, then the endpoints of any prefix of the interval $[r_1, r_2]$ also delineate an epoch:

Lemma 11.3.12. *Assume $\langle r_1, r_2 \rangle$ delineates an epoch of virtual node $v \in I_V$, and $r \in [r_1, r_2]$. Then $\langle r_1, r \rangle$ delineates an epoch of virtual node v.*

Proof. Immediate, by Definition 11.3.11. □

11.3.5 UpDown Sequences

We now define an "updown" sequence, which is a sequence of pairs $\langle u_i, d_i \rangle$. Each pair is a maximum-length sequence of rounds in which virtual node v is up. That is, if $\langle u_i, d_i \rangle$ is in the updown sequence, then we can conclude that v is down in round $u_i - 1$; if $d_i \neq \infty$, then we can also conclude that v is down in round $d_i + 1$.

Definition 11.3.13. *Let s be a (possibly infinite) sequence $\langle u_0, d_0 \rangle, \langle u_1, d_1 \rangle, \langle u_2, d_2 \rangle, \ldots$. We say that s is an **updown** sequence if it satisfies the following properties:*

- *(*The list of pairs is sorted.*) For all $i > 0$, if $\langle u_{i+1}, d_{i+1} \rangle$ is in the sequence, then $d_i < u_{i+1}$.*
- *(*Each pair delineates an epoch of v.*) For all $\langle u_i, d_i \rangle$ in the sequence, $\langle u_i, d_i \rangle$ delineates an epoch of v in α.*
- *(*The list of pairs is complete.*) For ever round $r > 0$, if virtual node v is up in round r, then there exists some $\langle u_i, d_i \rangle$ in the sequence such that $r \in [u_i, d_i]$.*

These properties imply that the updown sequence contains maximum-length sequences of rounds in which the virtual node is up. To see this fact, assume instead that there were some $\langle u_i, d_i \rangle$ in the updown sequence for v where v is up in round $d_i + 1$: The third property of an updown sequence guarantees that for some $\langle u', d' \rangle$ in the updown sequence, $d_i + 1 \in [u', d']$. The second property of an updown sequence implies that v is down in round $r' - 1$. The first property of an updown sequence implies that the pairs are ordered and do not overlap. From this we can conclude that $j' = d_i + 1$, and thus that v is down in round d_i. But the second property of an updown sequence implies that v is up in round d_i, resulting in a contradiction. From this we can conclude that there is a unique updown sequence for v. In more detail, we show that such a sequence always exists, and is unique. (Recall that we have fixed an execution α.)

Lemma 11.3.14. *There is exactly one sequence $s = \langle u_0, d_0 \rangle, \langle u_1, d_1 \rangle, \langle u_2, d_2 \rangle, \ldots$ such that s is an updown sequence.*

Proof. Claim 1: First, we argue that there exists some $u_i = r$ if and only if v is down in round $r-1$ and up in round r. If $\langle r, d_i \rangle \in s$, it is immediately clear that v is down in round $r-1$ and up in round r. Assume, then, that v is down in $r-1$ and up in r: then there exists some $\langle u, d \rangle \in s$ such that $r \in [u, d]$; yet $r-1 \notin [u, d]$ since v is up in every round in $[u, d]$; thus we conclude that $u = r$.

Claim 2: Next, we argue that there exists some $d_i = r$ if and only if v is up in round r and down in round $r+1$. Assume v is up in round r and down in round $r+1$: then there exists some $\langle u, d \rangle \in s$ such that $r \in [u, d]$; yet $r+1 \notin [u, d]$, since v is up in every round in $[u, d]$; thus we conclude that $d = r$. Alternatively, assume that $\langle u_i, r \rangle \in s$. It follows immediately that v is up in round r. Assume for the sake of contradiction that v is also up in round $r+1$. Then there is some $\langle u', d' \rangle$ such that $r+1 \in [u', d']$, and where v is down in round $u'-1$. From this we conclude that $r+1 \leq d'$, and hence $r < d'$. There are thus two possibilities for u': either $u' \leq u$, which contradicts our assumption that the pairs in s are sorted and strictly increasing, or $u < u'$, which contradicts the assumption that v is down in orund $u'-1$.

Claim 3: Finally, we argue that there exists some $d_i = \infty$ if and only if there exists some round r such that v is up in every round $\geq \infty$. Assume there exists some $d_i = \infty$. Then by assumption, v is up in every round $\geq u_i$. Assume instead that there exists some round r such that v is up in every round $\geq r$. Assume for the sake of contradiction that every $d_i \neq \infty$. In this case, there must exist some $\langle u_i, d_i \rangle$ were $d_i > r$. Moreover, there must exists some $\langle u_{i+1}, d_{i+1} \rangle$ where $d_i < u_{i+1}$. But this contradicts our assumption that v is down in round $u_{i+1} - 1$.

Conclusion: It follows immediately from these three claims that there is exactly one updown sequence s: Given execution α, we can determine the exact set of u_i and d_i in the updown sequence; we can construct the updown sequence by sorting the u_i and d_i and matching them. (No other arrangement of the u_i and d_i can yield an updown sequence due to the ordering property.) □

We thus define *updown*(v) to be the (unique) updown sequence for v:

Definition 11.3.15. *For all $v \in I_V$, define* ***updown*(v)** *to be the unique updown sequence for v.*

11.4 Resetting a Virtual Node

In this section, we present a series of lemmas and invariants that describe the state of the virtual node after it is reset. Most of the lemmas proved in this section are related to the state variables *last-reset* and *last-good-state*. Recall that *last-reset* stores the virtual round in which, conceptually, the virtual node was reset; *last-good-state* stores the initial state of the virtual node. One of the main conclusions of this section (see Corollary 11.4.5) is that this intuition is correct: if some $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v, and hence v is down in round $r_1 - 1$, then each node that begins virtual round $r \in [r_1, r_2]$ has *last-reset* $= r_1 - 1$ at the beginning of round r. (In fact, we extend this claim slightly to include the case where $r = r_2 + 1$; that is, the claim holds at the end of virtual round r_2 as well.)

We begin with a basic lemma relating *last-reset* and *prev-rnd*: for each virtual node emulator, for each node in the system, these values are non-decreasing, and *last-reset* $\leq$ *prev-rnd* $\leq$ *rnd*.

Lemma 11.4.1. *For all $i \in I_B$, for all $v \in I_V$:*

1. $E(v).\textit{last-reset}_i$ *and* $E(v).\textit{prev-rnd}_i$ *are non-decreasing.*
2. $E(v).\textit{last-reset}_i \leq E(v).\textit{prev-rnd}_i \leq E(v).\textit{rnd}_i$.

Proof. We show this by induction on events in α. Initially, both conclusions are clearly true. Let e be the first event that causes one of the conclusions to be violated for any $j \in I_B$. Then e must be a recv event, as no other action modifies these state variables, and e must occur during one of the following phases, which are the only ones in which these state variables are modified:

- The *client phase* increments $E(v).\textit{rnd}_j$ by one. Clearly, the invariant is maintained.
- The *scheduled-veto-2 phase* sets $E(v).\textit{prev-rnd}_j$ to $E(v).\textit{rnd}_j$. Prior to event e, we know that $E(v).\textit{prev-rnd}_j \leq E(v).\textit{rnd}_j$; thus, conclusion (1) is maintained. Prior to e, $E(v).\textit{last-reset}_j \leq E(v).\textit{rnd}_j$; thus, after e we can conclude that $E(v).\textit{last-reset}_j \leq E(v).\textit{prev-rnd}_j$, and hence conclusion (2) is maintained.
- The *unscheduled-veto-2 phase* sets $E(v).\textit{prev-rnd}_j$ to $E(v).\textit{rnd}_j$, as in the previously discussed scheduled-veto-2 phase, hence the invariant is maintained.
- The *join-ack* updates $E(v).\textit{prev-rnd}_j$ and $E(v).\textit{last-reset}_j$ with data from a message that was broadcast by some other node $k \in I_B$. Since e is the first event to violate the invariant for any $j \in I_B$, we know that the state of k satisfied the invariant when the message was sent, and hence the values contained in the message also satisfy the invariant.
- The *join-veto* sets both $E(v).\textit{last-reset}_j$ and $E(v).\textit{prev-rnd}_j$ to $E(v).\textit{rnd}_j$, and thus the invariant is maintained.

□

We next consider what happens in a virtual round when a virtual node v is reset. Specifically, we assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of some virtual node v. This implies that v is down in round $r_1 - 1$. Our goal (informally) is to show that during each of the virtual rounds in the range $[r_1, r_2]$, for each emulator of virtual node v, the *last-reset* variable is set to round $r_1 - 1$. That is, the *last-reset* variable reflects the most recent round in which the virtual node is reset. This requires a sequence of lemmas. First, we show in Lemma 11.4.2 that this fact is true at the beginning of round r_1. Second, we extend Lemma 11.4.2 to show that *last-reset* $\geq r_1 - 1$ not only at the beginning of round r_1, but also in every round $\geq r_1$. Third, we show an invariant that is, loosely, the converse of the preceding lemmas: if *last-reset* is set to some value r, then the virtual node is down in virtual round r. Finally, we conclude

with a corollary proving the main claim that *last-reset* always reflects the most recent round in which virtual node v was reset.

Notice that this sequence of lemmas plays a key role in showing that the "reset protocol" works correctly. For example, from this argument, we can conclude that all the nodes emulating v at the beginning of virtual round r_1 are in agreement that virtual node v was reset in virtual round r_1-1. This set of lemmas therefore precludes the possibility that some node was participating in the emulation, and yet was not aware that the "reset" protocol was being invoked.

We begin by arguing that at the beginning of virtual round r_1, if v is down in round $r_1 - 1$ and the reset protocol has worked correctly, then the *last-reset* variable of each node that begins round r_1 is set to $r_1 - 1$. We also show that *prev-rnd* is set to $r_1 - 1$ at the beginning of round $r_1 - 1$. The *prev-rnd* pointer, intuitively, keeps track of the last "locally good" round, and is discussed further in Section 11.5. It is also set to $r_1 - 1$ during the reset protocol.

Lemma 11.4.2. *For virtual round $r_1 > 1$, assume virtual node $v \in I_V$ is down in round $r_1 - 1$ in execution α. Assume $i \in I_B$ begins round r_1 for v. Then at the beginning of round r_1:*

1. $E(v).\textit{last-reset}_i = r_1 - 1$;
2. $E(v).\textit{prev-rnd}_i = r_1 - 1$.

Proof. Since virtual node v is down in α in round r_1-1 one of two situations occurred in round $r_1 - 1$, by the definition of "down":

1. For some $k \in I_B$, k resets v in round r_1-1. Let J be the set of nodes that reset v in round r_1-1. (Recall that we say that a node resets a virtual node when it executes specific lines of pseudocode, described in Definition 11.3.5.) Then for every $j \in J$, the conclusions of the lemma are immediately satisfied, as the reset pseudocode occurs during the last phase of round r_1-1, the join-veto phase, and sets the state variables as described. (By Lemma 11.2.4, $E(v).rnd_j = r_1 - 1$ at the beginning and end of the join-veto phase, and hence throughout the join-veto phase.)

   Notice that since at least one node resets v, we know that $v \in \textit{schedule}[r_1 - 1 \bmod \mathsf{SMAX}]$; otherwise, the reset pseudocode cannot be executed in virtual round $r_1 - 1$.

   Assume for the sake of contradiction that there exists some $j \in I_B$, $j \notin J$, where j begins round r_1. Thus, at the end of round $r_1 - 1$, $E(v).\textit{joined}_j = \mathsf{true}$. Therefore node j has $E(v).\textit{joined}_j = \mathsf{true}$ prior to the recv event for the join-ack phase for round $r_1 - 1$. (There is no other opportunity for j to set *joined* to true, as by assumption j does not execute the reset code.) Since v is scheduled for round $r_1 - 1$ and j has *joined* set to true, node j sets $E(v).\textit{outgoing-msg}_j$ to veto, broadcasting a veto message in the join-veto phase.

   Since j has $E(v).\textit{joined}_j = \mathsf{true}$ at the end of the join-ack phase, node j is within distance $R_B/4$ of $loc(v)$ at the end of the join-ack phase, i.e., the beginning of the

join-veto phase. Similarly, i has $E(v).reset_i = \mathsf{true}$ at the end of the join-veto phase; otherwise, it would not execute the reset pseudocode. Thus, node i was within distance $R_B/4$ of $loc(v)$ at the end of the join-ack phase, i.e., the beginning of the join-veto phase.

Hence, nodes j and k are within distance $R_B/2$ of each other, and the broadcast service guarantees (by Lemma 8.1.40) that k receives either a collision or a veto message in the join-veto phase. If node k receives either a collision or veto message in the join-veto phase, however, then automaton $E(v)_k$ would not have executed lines 482–488, as assumed, leading to a contradiction.

2. For all $j \in I_B$, either $E(v).failed_j = \mathsf{true}$ or $E(v).joined_j = \mathsf{false}$ at the end of virtual round $r_1 - 1$, that is, the beginning of round r_1. This contradicts our assumption that node i begins round r_1.

□

The next lemma, Lemma 11.4.3, extends Lemma 11.4.2 in the following manner: Lemma 11.4.2 shows that immediately after a virtual node is reset, the variable *last-reset* is set appropriately for each emulator; Lemma 11.4.3 shows that *last-reset* is always at least as large as the most recent round in which the virtual node was reset. The proof follows from the fact that *last-reset* is assigned a new value during the reset pseudocode: during the join pseudocode, its value is copied from another emulator. This lemma depends primarily on Lemma 11.4.2, along with the claim that the join protocol works correctly:

Lemma 11.4.3. *Let $\langle r_1, r_2 \rangle$ delineates a finite epoch of v. If i has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the beginning of some basic round r_a and r_a is part of virtual round $r \geq r_1$, then $E(v).last\text{-}reset_i \geq r_1 - 1$ at the beginning of basic round r_a.*

Proof. First, consider the case where r_a is the first basic round in virtual round r_1: Lemma 11.4.2 states that if i begins virtual round r_1, then $E(v).last\text{-}reset_j = r_1 - 1$ at the beginning of virtual round r_1, and hence at the beginning of the first basic round in virtual round r_1.

Consider, for the sake of contradiction, the first basic round r_b associated with virtual round $r' \geq r_1$ in which some node k has joined and not failed at the beginning of basic round r_b and also has $E(v).last\text{-}reset_k < r_1 - 1$ at the beginning of basic round r_b. There are two phases in which $E(v).last\text{-}reset_k$ is updated during a basic round: join-veto and join-ack. There are also the only phases in which $E(v).joined_k$ is set to true; there are no phases in which $E(v).failed_k$ is set to false. Thus it is sufficient to examine these two phases.

First, $E(v).last\text{-}reset_k$ and $E(v).joined_k$ are modified during the reset pseudocode in lines 482–488 (Figure 10-9). In this case, $E(v).last\text{-}reset_k$ is set to $rnd = r' \geq r_1$. (As usual, the relationship between rnd and r' follows from Lemma 11.2.4.)

Second, $E(v).last\text{-}reset_k$ and $E(v).joined_k$ are modified during the join-ack phase in lines 460–463 (Figure 10-9). In this case, $E(v).last\text{-}reset_k$ is set to a value that was broadcast by some other replica k' during the join-ack phase. But we know that at the

beginning of the join-ack phase, node k' has $E(v).joined_{k'} = \mathsf{true}$ and $E(v).failed_{k'} = \mathsf{false}$, since otherwise k' would not send a join acknowledgment (lines 445–446, Figure 10-9). Thus, by the inductive hypothesis, $E(v).last\text{-}reset_{k'} \geq r_1 - 1$ when k' broadcasts the join acknowledgment, contradicting our assumption that k sets $E(v).last\text{-}reset_k < r_1 - 1$. □

The following lemma shows, loosely, the converse of Lemma 11.4.2. Lemma 11.4.2 argues that if v is down in some virtual round $r-1$ and up in some virtual round r, then the *last-reset* state variable has value $r-1$ (immediately afterward). The following invariant argues that if *last-reset* state variable has value $r-1$, then the virtual node was down in virtual round $r-1$. It then follows, as shown in Corollary 11.4.5, that the virtual node is up in the following virtual round r.

Lemma 11.4.4. *If $E(v).last\text{-}reset_i = r$, for some $i \in I_B$, $r \in \mathbb{N}$, then v is down in virtual round r.*

Proof. The proof proceeds by induction on events in α. Assume for the sake of contradiction that event e is the first event that causes the invariant to be violated for some $j \in I_B$. Then e must be a recv event that updates $E(v).last\text{-}reset_i$ for some $i \in I_B$. There are two phases in which $E(v).last\text{-}reset_i$ is modified. First, in the join-veto phase, $E(v).last\text{-}reset_j$ is set to $E(v).rnd_j$, which immediately implies by definition that v is down in round $E(v).rnd_j$. Second, in the join-ack phase, $E(v).last\text{-}reset_j$ is set to a value broadcast by another $k \in I_B$, specifically the value of $r' = E(v).last\text{-}reset_k$ at the beginning of the join-ack phase. Since prior to event e the invariant is satisfied, we know that virtual node v is down in virtual round r', and thus after event e in $E(v).last\text{-}reset_j$. In both cases, the invariant is maintained after event e, contradicting our assumption. □

We can then show the following corollary. This result is perhaps the most frequently used claim from this section.

Corollary 11.4.5. *If $\langle r_1, r_2 \rangle$ delineates a finite epoch of v and i begins virtual round $r \in [r_1, r_2 + 1]$, then $E(v).last\text{-}reset_i = r_1 - 1$ at the beginning of virtual round r.*

Proof. By Lemma 11.4.3, we know that $E(v).last\text{-}reset_i \geq r_1$ at the beginning of virtual round r. Assume, then, for the sake of contradiction that $E(v).last\text{-}reset_i = r'$ at the beginning of virtual round r, and virtual round $r' \geq r_1$. By Lemma 11.4.4, virtual node v is down in round r'. We also know that $r' \leq r - 1$ by Lemma 11.4.1, as $E(v).last\text{-}reset_i \leq E(v).rnd_i$ at the end of virtual round $r-1$. Thus, $r_1 \leq r' \leq r_2$ and v is down in round r'. This contradicts the assumption that $\langle r_1, r_2 \rangle$ delineates an epoch in which v is up in each round. □

We conclude with one final corollary on the relationship between *last-reset* at different emulators for virtual node v:

Corollary 11.4.6. *Assume virtual node v is up in virtual round r. For some $i, j \in I_B$:*

1. *If i and j begin virtual round r for v, then $E(v).last\text{-}reset_i = E(v).last\text{-}reset_j$ at the beginning of virtual round r.*

2. *If i and j begin virtual round r for v, then $E(v).\textit{last-reset}_i = E(v).\textit{last-reset}_j$ throughout virtual round r.*

Proof. Choose r' to be the largest round $< r$ such that virtual node v is down in r'. Then $\langle r'+1, r\rangle$ delineates a finite epoch of v. Conclusion (1) holds by Corollary 11.4.5: $E(v).\textit{last-reset}_i = r'$ and $E(v).\textit{last-reset}_j = r'$. Conclusion (2) follows since $E(v).\textit{last-reset}$ is non-decreasing and $\leq r$ through the end of virtual round r, by Lemma 11.4.1; thus $r' \leq E(v).\textit{last-reset} \leq r$: if $E(v).\textit{last-reset} > r'$ at any point during virtual round r, then we can conclude by Lemma 11.4.4 that v is down in round $E(v).\textit{last-reset}$, contradicting our prior assumption that v is up for every round in range $[r'+1, r]$. Thus, $E(v).\textit{last-reset}$ is equal to r' throughout virtual round r for both i and j. □

11.5 Round Colors

This section contains some of the key definitions and lemmas used in the proof. We identify "good rounds" and "bad rounds," as well as "green rounds" and "red rounds." Having assigned colors to rounds, we can then argue about which combinations of colors can co-exist in a virtual round; much of this proof has a flavor similar to the analysis of the three-phase-commit protocols. (See [95, 96] and the discussion in Section 10.1.6 on the similarities.)

We begin in Section 11.5.1 by defining green rounds and red rounds, and also good rounds and bad rounds. The key definitions here are Definition 11.5.1 and Definition 11.5.2.

In Section 11.5.2, we take a detour through the analysis of the join protocol to show that when a node joins the emulation, it correctly receives the ballot and color information. The key lemma in this argument that is used in later sections of this proof is Lemma 11.5.3. As a result, we can conclude that the color of a round (i.e., whether it is green or red) and the status of a round (i.e., whether it is good or bad) persist, eventually as nodes join and leave (see Lemmas 11.5.4 and 11.5.5).

In Section 11.5.3, we discuss the relationship between good rounds and green rounds, showing that all green rounds are good (Corollary 11.5.7), and the relationship between bad rounds and red rounds, showing that all red rounds are bad (Lemma 11.5.9). In the process of proving these lemmas, we show that if two replicas assign a color to a virtual round, then that color differs by at most one shade.

In Section 11.5.4, we relate the value of the *prev-rnd* pointer (in the emulator state) to the status of a round. Intuitively, the *prev-rnd* pointer remembers the most recent round that a node, locally, believes to be good; globally, across all the emulators, we can conclude from the results in Section 11.5.3 that the round in *prev-rnd* is not red, as is stated in Lemma 11.5.12.

Finally, in Section 11.5.5, we show (in Corollary 11.5.16) that if a round is not red for some virtual node v, then it has a unique "proposer" and a unique "ballot" (see Definition 11.5.17). Notice that when combined with the results from Section 11.5.4, we can conclude that any round stored in the *prev-rnd* pointer has a unique proposer and a unique ballot. This fact will be key in reconstructing executions of the virtual

node from the ballot history. Moreover, since there is a unique ballot for the virtual round, we can conclude that all the replicas with a round r ballot agree on that ballot. We show in Corollary 11.5.20 that in fact all the nodes that complete virtual round r—or any larger round prior to v being down—agree on the ballot for round r.

11.5.1 Definitions

In this section, we define what it means for a round to be a "good round" a "bad round," a "green round," or a "red round." We begin with the definition of a "green round." A green round is, essentially, an ideal round in that at least one of the virtual node emulators detects no problems and hence designates it as green. A "red round," by contrast, is a round in which at least one of the virtual node emulators detects significant problems, and hence designates it as red.

Definition 11.5.1. *Let $v \in I_V$ be a virtual node, and $r > 0$ be a virtual round.*

- *We say that round r is a **green round** for v if there exists some node $i \in I_B$ that participates in round r and has $E(v).\textit{round-status}[r]_i = \mathsf{green}$ at the end of the last agreement phase of virtual round r.*
- *We say that round $r > 0$ is a **red round** for v if there exists some node $i \in I_B$ that participates in virtual round r and $E(v).\textit{round-status}[r]_i = \mathsf{red}$ at the end of the last agreement phase of virtual round r.*

Notice that a round is designated as green or red based on only a single node locally assigning a virtual round to be red or green. That is, a round is green or red based on a local observation. By contrast, we designate a round as good or bad based on the state of all participating nodes. We say that a round is a "good round" if *every* participating emulator considers the round to be green or yellow. Similarly, we designate a virtual round as bad if every participating emulator consider the round to be red or orange.

Definition 11.5.2. *Let $v \in I_V$ be a virtual node, and $r > 0$ be a virtual round.*

- *We say that round r is a **good round** for $v \in I_V$ if for every $i \in I_B$ that participates in round r, either $E(v).\textit{round-status}[r]_i = \mathsf{green}$ or $E(v).\textit{round-status}[r]_i = \mathsf{yellow}$ at the end of the last agreement phase of virtual round r.*
- *We say that round r is a **bad round** for $v \in I_V$ if for every $i \in I_B$ that participates in round r, either $E(v).\textit{round-status}[r]_i = \mathsf{red}$ or $E(v).\textit{round-status}[r]_i = \mathsf{orange}$ at the end of the last agreement phase of virtual round r.*

11.5.2 Extending the Color Definitions: the Join Protocol, Revisited

Notice that each of the definitions in Section 11.5.1 specifies the color of some virtual round r based only on the state of the nodes at the end of the last agreement phase

of round r. In fact, the color designated at the end of the last agreement phase permanently fixes the color designation of a virtual round, until the virtual node is reset. (After a virtual node is reset, its prior history is no longer relevant.)

This fact follows from the fact that after the last agreement phase of virtual round r, the color designation of a round is modified only during the join and reset phases; the join phases simply transmit the already-determined color information to new nodes; the reset phase only modifies the color information when the virtual node is, in fact, reset. Thus, this section begins with a technical lemma which shows that the join protocol acts as expected, and concludes with the claim that the color designation is unchanged after the last agreement phase.

In more detail, the first lemma here argues, essentially, that if some node i has some particular setting for the state variable *round-status*$[r]$, then either node i itself was participating in virtual round r, or node i received a copy of *round-status*$[r]$ (either directly or indirectly) from some node j that itself was participating in virtual round r. The same property also holds for the state stored in *ballot*$[r]$.

In the following lemma, the virtual round r represents the virtual round of interest, in terms of *round-status*$[r]$ and *ballot*$[r]$ The round $r' \geq r$ represents the "current" virtual round, i.e., the virtual round during which the state of node i is being observed. Basic round r_a is a basic round associated with virtual round r, and basic round r_b is a basic round associated with virtual round r'. More specifically, basic round r_a is the last agreement phase of virtual round r, i.e., the basic round in which the *round-status*$[r]$ and *ballot*$[r]$ data structures are updated. Basic round r_b is the "current" phase, i.e., the the phase after which the state of node i is being observed. While seemingly a small building block for the main lemmas in this section, Lemma 11.5.3 is actually used several times in later sections of the proof to show that the join protocol acts as expected.

Lemma 11.5.3. *Assume that $\langle r_1, r_2 \rangle$ delineates an epoch of virtual node $v \in I_V$ in α, and let $r, r' \in [r_1, r_2]$ be virtual rounds, $r' \geq r$. Let r_a be the basic round associated with the last agreement phase of virtual round r, and let r_b be a basic round in virtual round r', $r_b \geq r_a$.*

If node $i \in I_B$ has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the end of basic round r_b, then there exists some $j \in I_B$ where j particpates in virtual round r, and:

- *at the end of basic round r_a, $E(v).ballot[r]_i = E(v).ballot[r]_j$ at the end of basic round r_b;*
- *at the end of basic round r_a, $E(v).round\text{-}status[r]_i = E(v).round\text{-}status[r]_j$ at the end of basic round r_b.*

Proof. We proceed by induction on basic rounds. For the base case, consider round $r_b = r_a$: Since node i itself has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_j = \mathsf{false}$ at the end of round r_a, then we choose $j = i$ and the criteria are met since node i participates in round r by Lemma 11.3.9.

For the sake of contradiction, let $r_b > r_a$ be the first basic round in which the lemma fails to be true. Thus, in round r_b, node i must have modified either $E(v).joined_i$, $E(v).ballot[r]_i$ or $E(v).round\text{-}status[r]_i$. After the last agreement

phase of round r, i.e., after round r_a, there are only two phases in which i modifies $E(v).\mathit{round\text{-}status}[r]_i$ or $E(v).\mathit{round\text{-}status}[r]_i$: when i joins v and when i resets v. Similarly, these are the only two phases in which i sets $E(v).\mathit{joined}_i$ to true. (Since node i is unrecoverable, it cannot have set $E(v).\mathit{failed}_i$ to false.)

First, we can conclude that node i does not reset v in round r', since $r' \in [r_1, r_2]$, and hence by assumption v is up in round r.

Thus, we can conclude that i must join v in basic round r_b. In that case, however, node j received a round r_b message from some other node k that had $E(v).\mathit{joined}_k = \mathsf{true}$ and $E(v).\mathit{failed}_k = \mathsf{false}$ at the beginning of basic round r_b; moreover, since i copied the ballot and round status values from the message, we can conclude that node k also had ballot or round status values causing the lemma to be violated, contradicting our assumption that r_b is the first such basic round. □

We can now conclude that if a round is designated as good or bad at the end of the last agreement phase, then every node continues to designate the round accordingly, until the virtual node is reset:

Lemma 11.5.4. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$ in α. Let $r \in [r_1, r_2]$ be a virtual round. If $i \in I_B$ completes virtual round r_2 for v, then we conclude the following:*

1. *If round r is good, then $E(v).\mathit{round\text{-}status}[r]_i \in \{\mathsf{green}, \mathsf{yellow}\}$ at the end of round r_2.*
2. *If round r is bad, then $E(v).\mathit{round\text{-}status}[r]_i \in \{\mathsf{red}, \mathsf{orange}\}$ at the end of round r_2.*
3. *If round r is not red, then $E(v).\mathit{round\text{-}status}[r]_i \neq \mathsf{red}$ at the end of round r_2.*

Proof. Let basic round r_a be the last agreement phase of virtual round r, and let r_b be the last basic round in virtual round r_2; thus $r_b > r_a$. Since $i \in I_B$ completes round r_2, we know that $E(v).\mathit{joined}_i = \mathsf{true}$ and $E(v).\mathit{failed}_i = \mathsf{false}$ at the end of basic round r_b. Therefore by Lemma 11.5.3, we conclude that there exists some $j \in I_B$ where j participates in virtual round r and at the end of basic round r_a, $E(v).\mathit{round\text{-}status}[r]_j = E(v).\mathit{round\text{-}status}[r]_i$ at the end of round r_2.

If round r is good, since j participates in r, we know that $E(v).\mathit{round\text{-}status}[r]_j = \mathsf{green}$ or $E(v).\mathit{round\text{-}status}[r]_j = \mathsf{yellow}$ at the end of round r_a by Definition 11.5.2, which leads to the desired conclusion.

Similarly, if round r is bad, since j participates in round r, we know that the status $E(v).\mathit{round\text{-}status}[r]_j = \mathsf{red}$ or $E(v).\mathit{round\text{-}status}[r]_j = \mathsf{orange}$ at the end of round r_a by Definition 11.5.2, which leads to the desired conclusion.

Similarly, if round r is not red, then since j participates in round r, we know that $E(v).\mathit{round\text{-}status}[r]_j \neq \mathsf{red}$ at the end of round r_1 by Definition 11.5.1, which leads to the desired conclusion. □

We can also draw conclusions relating to green and red rounds: if a node designates a round as green or red at the end of a virtual round, then the virtual round is green or red:

Lemma 11.5.5. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$ in α. Let $r \in [r_1, r_2]$ be a virtual round. If $i \in I_B$ completes virtual round r_2, then we conclude the following:*

1. *If $E(v).\textit{round-status}[r]_i = \mathsf{green}$ at the end of round r_2, then round r is green.*
2. *If $E(v).\textit{round-status}[r]_i = \mathsf{red}$ at the end of round r_2, then round r is red.*

Proof. Let basic round r_a be the last agreement phase of virtual round r, and let r_b be the last basic round in virtual round r_2; thus $r_b > r_a$. Since $i \in I_B$ completes round r_2, we know that $E(v).\textit{joined}_i = \mathsf{true}$ and $E(v).\textit{failed}_i = \mathsf{false}$ at the end of basic round r_b. Therefore by Lemma 11.5.3, we conclude that there exists some $j \in I_B$ where j participates in virtual round r and at the end of basic round r_a, $E(v).\textit{round-status}[r]_j = E(v).\textit{round-status}[r]_i$ at the end of round r_2.

If $E(v).\textit{round-status}[r]_i = \mathsf{green}$ at the end of round r_2, then $E(v).\textit{round-status}[r]_j = \mathsf{green}$ at the end of the last agreement phase of r, and hence by definition round r is green.

Similarly, if $E(v).\textit{round-status}[r]_i = \mathsf{red}$ at the end of round r_2, then the status $E(v).\textit{round-status}[r]_j = \mathsf{red}$ at the end of the last agreement phase of r, and hence by definition round r is red. □

11.5.3 Relating Good, Bad, Green, and Red Rounds

In this section, we prove two key lemmas that relate green rounds with good rounds and red rounds with bad rounds. Specifically, we show the following two properties:

- (Corollary 11.5.7) If a virtual round r is green for virtual node v, then it is good for v. Recall that a virtual round is green if even one (participating) replica designates it as green; a virtual round is good only if *every* participating replica designates it as green or yellow. This lemma plays a key role in proving that the protocol has the desired agreement property: in any good round, we will show that the replicas agree on an execution of the virtual node; however, an individual emulator may not be able to detect locally whether a round is good or bad. If, however, an emulator determines (locally) that a round is green, then this property guarantees that the round is also good. Thus, in later sections of this proof (for example, Corollary 11.7.12), we use this property to show that under certain conditions (i.e., when the round is green), an emulator can have confidence that a round is good, which will imply that agreement has been reached.

- (Lemma 11.5.9) If a virtual round r is red for virtual node v, then it is bad for v. As in the first property, a virtual node is red if even one (participating) replica designates it as red; a virtual round is bad only if *every* participating replica designates it as red or orange. In later sections, the contrapositive of this lemma plays a key role in ensuring that all the replicas have sufficient data to reconstruct the virtual node execution that has been chosen by the agreement protocol: we will show (in Corollary 11.5.18) that if a round is not red for some

round r, then every emulator has the required ballot information about that round; if any emulator designates a round as green or yellow, then by definition it is not a bad round; hence by the contrapositive of Lemma 11.5.9, we can conclude that every emulator has enough data to reconstruct the virtual node execution.

Our first goal is to show that every green round is a good round. We instead prove a somewhat more general lemma: instead of assuming that the round is green (which requires a participating node to designated the round as green), we assume only that some non-failed node within distance $3R_B/4$ of $loc(v)_V$ designates the round as green. Note that this node may not be participating in emulating v. The desired result then follows as an immediate corollary, as in a green round there is a participating node that satisfies the required conditions.

Lemma 11.5.6. *Let $v \in I_V$ be a virtual node, $r > 0$ a virtual round, and $i \in I_B$ a node that is non-failed through the end of the last agreement phase. If the following two conditions hold:*

1. *$E(v).\textit{round-status}[r]_i \in \{\mathsf{green}, \perp\}$ at the end of the last agreement phase of virtual round r; and*

2. *node i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the last agreement phase of virtual round r;*

Then virtual round r is good for v.

Proof. We need to show for every $j \in I_B$, if j participates in virtual round r, then either $E(v).\textit{round-status}[r]_i = \mathsf{green}$ or $E(v).\textit{round-status}[r]_i = \mathsf{yellow}$ at the end of the last agreement phase of virtual round r.

Choose some $j \in I_B$ that participates in virtual round r. Assume, for the sake of contradiction, that $E(v).\textit{round-status}[r]_j \in \{\mathsf{red}, \mathsf{orange}\}$ at the end of the last agreement of virtual round r. There are two nearly identical cases depending on whether v is scheduled or unscheduled in virtual round r.

First, assume that v is scheduled in r. At a high level, the argument proceeds as follows: since j has designated round r as red or orange, we can show that j broadcasts a veto message in the scheduled-veto-2 phase; as a result, node i either receives the veto message or detects a collision in the scheduled-veto-2 phase; thus we conclude that if i believes the round to be green, then i downgrades round r to a yellow round, resulting in a contradiction. We now proceed in more detail.

The first part of the argument involves showing that j broadcasts a message in the scheduled-veto-2 phase. Notice that at the end of the scheduled-veto-2 phase, j sets $E(v).\textit{round-status}[r]_j$ equal to $E(v).\textit{scheduled-status}_j$, from which we conclude that $E(v).\textit{scheduled-status}_j \in \{\mathsf{red}, \mathsf{orange}\}$ at the end of the scheduled-veto-2 phase.

Notice, however, that during the scheduled-veto-2 phase recv event, *scheduled-status* can be updated to yellow, but *not* to red or orange. Thus, we can assume that $E(v).\textit{scheduled-status}_j \in \{\mathsf{red}, \mathsf{orange}\}$ prior to the scheduled-veto-2 phase of round r.

Specifically, the latest that $E(v).\mathit{scheduled\text{-}status}_j$ can be updated to red or orange is the recv event of the scheduled-veto-1 phase.

We now argue that this fact implies the following: at the end of the scheduled-veto-1 recv event, the message buffer $E(v).\mathit{outgoing\text{-}msg}_j = \langle \mathsf{vn}, v, \mathsf{veto} \rangle$. In particular there are three conditions which must be met at the end of the recv event for this to be true:

- $E(v).\mathit{scheduled\text{-}status}_j \in \{\mathsf{red}, \mathsf{orange}\}$: as already argued.
- $E(v).\mathit{joined}_j = \mathsf{true}$: since j participates in round r, we know that $E(v).\mathit{joined}_j = \mathsf{true}$ through the end of the scheduled-veto-2 phase.
- $E(v).\mathit{scheduled}_j = \mathsf{true}$: as already assumed.

We know that node j is within distance $R_B/4$ of $\mathit{loc}(v)$ at the beginning of the scheduled-veto-2 phase, since $E(v).\mathit{joined}_j = \mathsf{true}$.

We now argue that this contradicts the assumption that i has designated the round as green. By assumption, we also know that i is within distance $3R_B/4$ of $\mathit{loc}(v)$ at the beginning of the last agreement phase, i.e., the scheduled-veto-2 phase. Thus nodes i and j are within distance R_B of each other at the beginning of the scheduled-veto-2 phase, and hence by Lemma 8.1.40, node i either receives the veto message from j, or detects a collision. In either case, however, if $E(v).\mathit{scheduled\text{-}status}_i \in \{\perp, \mathsf{green}\}$, then $E(v).\mathit{scheduled\text{-}status}_i$ is set to yellow. Thus, we know that $E(v).\mathit{scheduled\text{-}status}_i \neq \mathsf{green}$, contradicting our assumption on i.

Next, we consider the case where v is not scheduled in round r. The argument is identical, except that the scheduled-veto-2 phase is replaced with the unscheduled-veto-2 phase, and the protocol updates $\mathit{round\text{-}status}[r]$ directly, rather than $\mathit{scheduled\text{-}status}$. In more detail, notice that if $E(v).\mathit{round\text{-}status}[r]_j$ is red or orange at the end of the unscheduled-veto-2 phase, it must be red or orange at the beginning of the unscheduled-veto-2 phase, as it is only updated to yellow in the scheduled-veto-2 recv event. However, if j has $\mathit{round\text{-}status}[r]$ either red or orange by the end of the unscheduled-veto-1 recv event, then node j broadcasts a veto message in the unscheduled-veto-2 phase, since, as before, we know that node j has *joined* set to true, since j participates in round r. As before, node j and node i are within distance R_B of each other, and hence node i either receives the veto message or detects a collision. Again, in either case, node i sets $\mathit{round\text{-}status}[r] \neq \mathsf{green}$ and $\neq \perp$, contradicting our assumption on i. □

It is then a straightforward corollary that if virtual round r is green, then it is also good:

Corollary 11.5.7. *If virtual round r is green for virtual node v, then virtual round r is good for virtual node v.*

Proof. Since virtual round r is green, there exists some $i \in I_B$ that participates in virtual round r and has $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$ at the end of the last agreement phase of virtual round r. Thus we can conclude that i is non-failed through the end

of the last agreement phase, and that node i is within distance $R_B/4$ of $loc(v)$ at the end of each phase prior to the last agreement phase. (Otherwise, *joined* is set to false during the recv event.) Thus, by Lemma 11.5.6, we conclude that round r is good for virtual node v. □

We also show a second corollary which describes the situation at the end of a virtual round:

Lemma 11.5.8. *Let $v \in I_V$ be a virtual node, $r > 0$ a virtual round, and $j \in I_B$ a node that does not fail prior to the end of virtual round r. Assume that $E(v).round\text{-}status[r]_j = \mathsf{green}$ at the end of virtual round r. Then round r is good for v.*

Proof. Since $E(v).round\text{-}status[r]_j \neq \perp$ at the end of round r, we can conclude that either node j updates $E(v).round\text{-}status[r]_j$ directly during the agreement phases, or node j receives $E(v).round\text{-}status[r]_j$ during the join protocol. We thus consider two subcases:

- Node j joins v in virtual round r: In this case, there exists some node k that participates in round r for v that broadcasts the join acknowledgment to j. Thus, we can conclude that at the end of the last agreement phase, $E(v).round\text{-}status[r]_k = \mathsf{green}$. This implies that round r is a green round, and hence a good round by Corollary 11.5.7.
- Node j does not join v in virtual round r: In this case, node j must have set $E(v).round\text{-}status[r]_j = \mathsf{green}$ in either line 360 or line 426 during the last agreement phase of round r. We can also conclude, then, that node j is within distance $R_V = R_B/2$ of virtual node v at the end of the last agreement phase. Since the velocity of a node is bounded by $R_B/4$, we can conclude that j is within distance $3R_B/4$ at the beginning of the last agreement phase. We thus apply Lemma 11.5.6 to conclude that round r is good.

□

Next, we show that every red round is a bad round:

Lemma 11.5.9. *If virtual round r is red for virtual node v, then virtual round r is bad for virtual node v.*

Proof. By the definition of a red round, there exists some $i \in I_B$ that participates in round r and has $E(v).round\text{-}status[r]_i = \mathsf{red}$ at the end of the last agreement phase of virtual round r.

We need to show that for every $j \in I_B$, if j participates in virtual round r, then either $E(v).round\text{-}status[r]_i = \mathsf{red}$ or $E(v).round\text{-}status[r]_i = \mathsf{orange}$ at the end of the last agreement phase of virtual round r. Choose some arbitrary $j \in I_B$ that participates in virtual round r.

There are two nearly identical cases, depending on whether v is scheduled in round r. First, assume that v is scheduled in round r. The high level argument proceeds as

follows: since i has designated the round as red, i broadcasts a veto message in the scheduled-veto-1 phase; we can then conclude that node j receives this veto message or detects a collision in the scheduled-veto-1 phase, and thus downgrades the round to orange or red, as required. We now proceed in more detail.

Since i has $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{red}$ at the end of the scheduled-veto-2 phase, we know that $\mathit{scheduled\text{-}status} = \mathsf{red}$ at the end of the scheduled-ballot phase: in no other phase is *scheduled-status* is set to red, and *round-status*$[r]$ is set to *scheduled-status* in the scheduled-veto-2 phase recv event.

Since $E(v).\mathit{scheduled}_i = \mathsf{true}$ and $E(v).\mathit{joined}_i = \mathsf{true}$ by the end of the scheduled-ballot phase, and since $\mathit{scheduled\text{-}status} = \mathsf{red}$, node i sets *outgoing-msg* equal to $\langle \mathsf{vn}, v, \mathsf{veto} \rangle$. Since both i and j are participating, we know that both have $\mathit{joined} = \mathsf{true}$ at the end of the *scheduled-ballot* phase, and hence both are within distance $R_B/4$ of $\mathit{loc}(v)$ at the beginning of the scheduled-veto-1 phase. Thus, by Lemma 8.1.40, node j either receives the veto message from node i, or detects a collision in the scheduled-veto-1 phase. In either case, if $E(v).\mathit{scheduled\text{-}status}_j$ is not already red, then node j sets *scheduled-status* to orange. No later update to *scheduled-status* results in a color better than orange, since during the scheduled-veto-2 phase, the status is only updated if it is in $\{\perp, \mathsf{green}\}$. Thus we can conclude that at the end of the scheduled-veto-2 phase, node j has $E(v).\mathit{round\text{-}status}[r]$ equal to either red or orange.

Next, we consider the case where v is not scheduled in round r. The argument is identical, except that the scheduled-veto-2 phase is replaced with the unscheduled-veto-2 phase, and the protocol updates *round-status*$[r]$ directly, rather than *scheduled-status*. In more detail, notice that $E(v).\mathit{round\text{-}status}[r]_i$ must be equal to red by the end of the last passive-ballot phase, as there is no later opportunity to set it to red. In the last passive-ballot phase, when $\mathit{unscheduled\text{-}ballot\text{-}rnd} = \mathsf{SMAX}$, node i sets *outgoing-msg* to broadcast a veto message, and hence broadcasts a veto message in the unscheduled-veto-1 phase. Since i and j both participate in round r, we can conclude that they are both within distance $R_B/4$ of $\mathit{loc}(v)$ at the beginning of the scheduled-veto-1 phase, and hence by Lemma 8.1.40, node j either receives the veto message from node i or detects a collision. In either case, if $E(v).\mathit{round\text{-}status}[r]_j$ is not red, then node j sets $E(v).\mathit{round\text{-}status}[r]_j$ to orange by the end of the unscheduled-veto-1 phase. Moreover, there is no later opportunity for node j to upgrade the color to either green or yellow, as *round-status* is only updated in the unscheduled-veto-2 phase if it is in $\{\perp, \mathsf{green}\}$. Thus, at the end of the unscheduled-veto-2 phase, node j has $E(v).\mathit{round\text{-}status}[r]_j$ set to either red or orange, as required. □

We conclude with a corollary that combines Corollary 11.5.7 and Lemma 11.5.9 with Lemma 11.5.4 to draw conclusions about the color designation after the last agreement phase:

Corollary 11.5.10. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of v in α, and that $r \in [r_1, r_2]$ is a virtual round. Assume that node $i \in I_B$ completes round r_2.*

- *If round r is a green round, then $E(v).\mathit{round\text{-}status}[r]_i \in \{\mathsf{green}, \mathsf{yellow}\}$ at the end of round r_2.*

- *If round r is a red round, then $E(v).\textit{round-status}[r]_i \in \{\textsf{red}, \textsf{orange}\}$ at the end of round r_2.*

Proof. If round r is green, then by Corollary 11.5.7 virtual round r is a good round for virtual node v. Next, we invoke Lemma 11.5.4, to conclude that $E(v).\textit{round-status}[r]_i \in \{\textsf{green}, \textsf{yellow}\}$ at the end of round r_2.

If round r is red, then by Lemma 11.5.9 virtual round r is a bad round for virtual node v. Next, we invoke Lemma 11.5.4, to conclude that $E(v).\textit{round-status}[r]_i \in \{\textsf{red}, \textsf{orange}\}$ at the end of round r_2. □

11.5.4 Tracking of the Previous Round Pointer

Recall from the discussion in Section 10.2.2 that the *prev-rnd* variable is supposed to store the most recent round that, locally, appeared to be good. As a result of Lemma 11.5.9, an emulator can conclude that a round is not bad if it achieves a status of green or yellow. Thus, whenever a round is either green or yellow, the *prev-rnd* variable is updated. The following set of lemmas capture this behavior. This first lemma argues that *prev-rnd* always reflects a non-red round. It follows from the fact that a node updates *prev-rnd* only when the current round is either green or yellow.

Lemma 11.5.11. *Assume that $r > 0$ is a virtual round, and that $v \in I_V$ is up in round r. If $E(v).\textit{prev-rnd}_i = r$ at the end of any basic round, then r is not a red round.*

Proof. Initially, for all $j \in I_B$, node j has $E(v).\textit{prev-rnd}_j$ set to $\perp$. We proceed by contradiction: let r_a be the smallest basic round in which any $j \in I_B$ has $E(v).\textit{prev-rnd}_j = r'$ at the end of basic round r_a, for some red virtual round r' in which v is up. We now proceed to examine the various ways in which $E(v).\textit{prev-rnd}_j$ is modified during the algorithm, and show that this is impossible. In particular, there are four different phases in which $E(v).\textit{prev-rnd}_j$ is updated in basic round r_a: the scheduled-veto-2 phase, the unscheduled-veto-2 phase, the join-ack phase, and the join-veto phase. In the first two of these cases, $E(v).\textit{prev-rnd}_j$ is updated only if the current round is not red; in the join case, j copies the *prev-rnd* pointer from another node; in the last case, the virtual node is reset. In more detail:

1. lines 363–365, r_a is a scheduled-veto-2 phase: In this case, $E(v).\textit{prev-rnd}_j$ is set to $E(v).\textit{rnd}_j = r'$, the current round. This occurs only if $E(v).\textit{round-status}[r']_j \in \{\textsf{green}, \textsf{yellow}\}$, $E(v).\textit{joined}_j = \textsf{true}$, and $E(v).\textit{scheduled}_j = \textsf{true}$ in round r_a. We need to show that this is impossible if round r' is a red round.

   First, we show that node j participates in round r'. Notice that $E(v).\textit{scheduled}_j = \textsf{true}$, indicating that v is scheduled in round r', and hence the scheduled-veto-2 is the last agreement phase of round r'. We can conclude that $E(v).\textit{failed}_j = \textsf{false}$ at the end of the last agreement phase, since the broadcast service only delivers receive events to non-failed nodes. Since we already know that $E(v).\textit{joined}_j = \textsf{true}$, we conclude that j participates in round r'.

Now we invoke Lemma 11.5.9. We have assumed, for the sake of contradiction, that round r' is a red round; we then conclude that round r' is also a bad round. From this, we conclude that $E(v).\textit{round-status}[r']_j = \textsf{red}$ or $E(v).\textit{round-status}[r']_j = \textsf{orange}$ at the end of the last agreement phase of virtual round r', which in this case is the scheduled-veto-2 phase. Thus we have a contradiction.

2. lines 427–428, r_a is an unscheduled-veto-2 phase: This case is identical, with the exception that v is unscheduled. In this case, $E(v).\textit{prev-rnd}_j$ is set to $E(v).\textit{rnd}_j = r'$, the current round. This occurs only if $E(v).\textit{round-status}[r']_j \in \{\textsf{green}, \textsf{yellow}\}$, $E(v).\textit{joined}_j = \textsf{true}$, and $E(v).\textit{scheduled}_j = \textsf{false}$. We need to show that this is impossible if round r' is a red round.

   As before, it is clear that j particpates in round r'. Thus, if round r' were red, as we have assumed for the sake of contradiction, then by Lemma 11.5.9, round r' would be bad, implying that $E(v).\textit{round-status}[r']_j = \textsf{red}$ or $E(v).\textit{round-status}[r']_j = \textsf{orange}$, which is a contradiction.

3. Node j joins v in basic round r_a: In this case, node j receives the $\textit{idprev} - \textit{rnd} = r'$ pointer as part of a join-ack message from some other node k. In this case, if round r' were red, then node k violated the lemma's claim at the beginning of round basic r_a, when the join-ack message was sent, contradiction our assumption that r_a was the earliest basic round after which the lemma was violated.

4. Node j resets v in basic round r_a: This implies immediately that virtual node v is down in round r', contradicting our assumption that v is up in round r'.

□

Our next lemma states that the *prev-rnd* field of the ballot is also a non-red round. This lemma follows immediately from the fact that a ballot is constructed by copying the local *prev-rnd* state variable.

Lemma 11.5.12. *Assume that $r, r' > 0$ are virtual rounds, and that virtual node $v \in I_V$ is up in round r. If $E(v).\textit{ballot}[r'].\textit{prev-rnd}_i = r$ at the end of any basic round, then r is not a red round.*

Proof. Initially, $E(v).\textit{ballot}[r].\textit{prev-rnd}_i = \perp$, for all r. We proceed by contradiction: let r_a be the first basic round for which there exists some node $j \in I_B$ such that $E(v).\textit{ballot}[r_1].\textit{prev-rnd}_j = r_2$, where v is up in round r_2 and r_2 is a red round. There are five cases to consider in which a ballot is modified:

- Round r_a is a vn phase: in this case, $E(v).\textit{ballot}[r_1].\textit{prev-rnd}_j$ is set equal to $E(v).\textit{prev-rnd}_j$, and Lemma 11.5.11 implies that round r_2 is either not a red round or v is not up in r_2.

- Round r_a is a scheduled-ballot phase: In this case, the ballot is received as a message from some other node k; hence $E(v).\textit{ballot}[r_1].\textit{prev-rnd}_k = r_2$ at the end of round $r_a - 1$, constradicting our assumptionat that r_a is the first round in which the claim is violated.

- Round r_a is a unscheduled-ballot phase: There are two ways that the $ballot[r]$ data structure may be updated, either by creating a new ballot (line 389) or by receiving a ballot (line 386).

  In the first case, as in the case of the vn phase, $E(v).ballot[r_1].prev\text{-}rnd_j$ is set equal to $E(v).prev\text{-}rnd_j$, and Lemma 11.5.11 implies that round r_2 is either not a red round or v is not up in r_2.

  In the second case, as in the scheduled-ballot phase, the ballot is received as a message from some other node k; hence $E(v).ballot[r_1].prev\text{-}rnd_k = r_2$ at the end of round $r_a - 1$, constradicting our assumptionat that r_a is the first round in which the claim is violated.

- Node j joins v in basic round r_a: In this case, node j receives the *ballot* data structure as part of a join-ack message from some other node k. In this case, if round r_2 were red, then node k violated the inductive hypothesis at the beginning of round basic r_a, when the join-ack message was sent, contradiction our assumption that r_a was the earliest basic round after which the inductive hypothesis was violated.

- Node j resets v in basic round r_a: In this case, node j sets $E(v).ballot[r].prev\text{-}rnd_j$ to $\perp$.

There are no other phases in which the *ballot* is updated. □

11.5.5 Unique Proposer / Unique Ballot

In this section, we argue about what happens in non-red rounds. The key lemma in this section is Lemma 11.5.14, which says that each non-red round has a unique "proposer" (i.e., a node that sends a ballot in a ballot phase) and a unique ballot (see Definition 11.5.17). From this, we prove a series of lemmas which show that in a non-red round r, participating emulators will share the same ballot for round r. This argument begins with Corollary 11.5.18, and culminates in Corollary 11.5.20.

We begin with Lemma 11.5.13, which says that if some virtual round r_2 is not red for some virtual node v, then there is a unique "proposer" and a unique ballot for that virtual node for round r_2. As in Lemma 11.5.6, we prove a slightly stronger lemma: instead of assuming that round r_2 is a red round, we assume only that there is some non-failed node k that is within distance $3R_B/4$ of $loc(v)_V$ and designates the round as red. We also assume that node k has a non-$\perp$ ballot for round r_2, as our goal is to find the unique ballot for round r_2.

Lemma 11.5.13. *Assume that $r_2 > 0$ is a virtual round, and $v \in I_V$ is a virtual node, and that there exists some node $k \in I_B$ that does not fail prior to the end of the last agreement phase, and for which the following three conditions hold:*

1. *$E(v).round\text{-}status[r_2]_k \neq$ red at the end of the last agreement phase of virtual round r_2;*

2. $E(v).ballot[r_2]_k \neq \perp$ *at the end of the last agreement phase of virtual round* r_2*; and*

3. *node* k *is within distance* $3R_B/4$ *of* $loc(v)_V$ *at the beginning of the virtual round, and at the beginning of each phase through the end of the last agreement phase of virtual round* r_2*.*

Then, there is exactly one $i \in I_B$ *where* i *begins virtual round* r_2 *such that:*

1. *Node* $i \in I_B$ *performs a* $\mathsf{bcast}(m, \cdot)_{i,v}$*,* $m \neq \perp$*, in either the* scheduled-ballot *or* unscheduled-ballot *phase of round* r_2*.*

2. *For every* $k' \in I_B$*, where* k' *is within distance* $3R_B/4$ *of* $loc(v)_V$ *at the beginning of the ballot phase for the appropriate agreement instance and does not fail prior to the end of the last agreement phase:* $E(v).ballot[r_2]_{k'} = E(v).ballot[r_2]_i$ *at the end of the last agreement phase of* r_2*.*

Proof. In the first part of the proof, we show that there is at most one $i \in I_B$ that broadcasts a message for v during a ballot phase of virtual round r_2. We argue that if two nodes, i, j both broadcast during the ballot phase, then by the completeness of the collision detector, either k will receive both i and j's message, or k will detect a collision. In either case, node k then designates the round as red, contradicting our assumption that k designates the round $\neq$ red. In the second part of the proof, we show that every $k \in I_B$ that is nearby and non-failed has the same ballot. This follows simply from the fact that only one ballot was broadcast in a ballot phase, and if any node did not receive the ballot, then it designates the round as red, which results in a contradiction.

Part 1: We first show that there is at most one $i \in I_B$ that broadcasts a message for v during a ballot phase of virtual round r_2. Assume for the sake of contradiction that $i, j \in I_B$ both broadcast messages $\neq \perp$ for virtual node v in either the scheduled-ballot or the unscheduled-ballot phase. We show that k has $E(v).round\text{-}status[r_2]_k =$ red at the end of the last agreement phase, contradicting the hypothesis. (Note that it is possible that $k = i$ or $k = j$.)

First, notice that if i broadcasts in the scheduled-ballot phase, j cannot broadcast in the unscheduled-ballot phase, and vice versa: by Corollary 11.2.9 we know that a long as i and j have not failed, at the end of each basic round in virtual round r, $E(v).scheduled_i = E(v).scheduled_j$; nodes i and j can only broadcast a ballot in the scheduled ballot phase when $E(v).scheduled =$ true.

Second, consider the case where both i and j broadcast during the same ballot phase. Both must have $E(v).joined =$ true, and thus remain within distance $R_B/4$ of $loc(v)_V$ at the beginning of the ballot phase. By assumption, node k has $E(v).failed_k =$ false during the ballot phase, and through the end of the agreement protocol. Also, node k is within distance $3R_B/4$ of $loc(v)_V$ throughout the scheduled ballot phase.

By Lemma 8.1.40, either k receives messages from both i and j, or k detects a collision. In either case, k sets $E(v).round\text{-}status[r_2]_k$ to red, either directly (in the unscheduled agreement case), or by setting $E(v).scheduled\text{-}status_k$ to red and

updating $E(v).\mathit{round\text{-}status}[r_2]_k$ to red at the end of the scheduled-veto-2 phase. This implies a contradiction, from which we conclude that at most one $i \in I_B$ broadcasts a message for v in a ballot phase.

Since $E(v).\mathit{ballot}[r_2]_k \neq \perp$, we can conclude that at least one $i \in I_B$ broadcast a ballot, as $E(v).\mathit{ballot}[r_2]_k$ is updated only in the scheduled-ballot and unscheduled-ballot phases. Since we have already shown that there cannot be more than one node that broadcasts in a ballot phase, we can thus conclude that there is a unique node $i \in I_B$ that broadcasts a message in a ballot phase of virtual round r_2.

Part 2: Fix some k' satisfying the assumptions required for the second conclusion. The second conclusion follows from noticing that k' receives the unique ballot message: as we have argued above, by Lemma 8.1.40, either k' receives the message from i or detects a collision; if k' detects a collision during a ballot phase, then k sets $E(v).\mathit{round\text{-}status}[r_2]_{k'}$ to red, contradicting our assumption. Thus k' receives the ballot from i, and copies it either in line 317 or line 386, depending on whether v is scheduled for virtual round r_2. □

In this next lemma, we consider the special case where a node k is participating in the virtual node:

Lemma 11.5.14. *Assume that $r_2 > 0$ is a virtual round, and $v \in I_V$ is a virtual node, and that there exists some node $k \in I_B$ that participates in virtual round r_2 with $E(v).\mathit{round\text{-}status}[r_2]_k \neq$ red at the end of the last agreement phase of r_2. Then there is exactly one $i \in I_B$ where i begins virtual round r_2 such that:*

1. *Node $i \in I_B$ performs a $\mathsf{bcast}(m, \cdot)_{i,v}$, $m \neq \perp$, in either the* scheduled-ballot *or* unscheduled-ballot *phase of round r_2.*

2. *For every node $k' \in I_B$ where k' is participating in v: $E(v).\mathit{ballot}[r_2]_{k'} = E(v).\mathit{ballot}[r_2]_i$ at the end of the last agreement phase of r_2.*

Proof. The proof follows from a straightforward application of Lemma 11.5.13. Notice that since k is participating in virtual node v in round r_2, we can conclude that k is within distance $R_B/4$ of $\mathit{loc}(v)_V$ at the beginning of r_2 and at the beginning of each phase through the last agreement phase. Similarly, we conclude that k does not fail until the end of the last agreement phase.

Next, we need to argue that $E(v).\mathit{ballot}[r_2]_k \neq \perp$ at the end of the last agreement phase. If no ballot is broadcast in the ballot phase (be it scheduled or unscheduled), then $E(v).\mathit{round\text{-}status}[r_2]_k$ also ends up red, either as a result of line 319 or line 382, depending on whether v is scheduled or unscheduled. Assume node i broadcasts a ballot in one of the two ballot phases; since node i broadcasts a ballot only when $E(v).\mathit{joined}_i =$ true and $E(v).\mathit{failed}_i =$ false, we can conclude that i is also within distance $R_B/4$ of $\mathit{loc}(v)_V$, and hence by the completeness of the basic broadcast service, either k receives the ballot or detects a collision. In the latter case, $E(v).\mathit{round\text{-}status}[r_2]_k$ is set to red; thus we can conclude that k receives the ballot. That is, $E(v).\mathit{ballot}[r_2]_k \neq \perp$ at the end of the last agreement phase of r_2. We then invoke Lemma 11.5.13 to conclude the proof. □

We next prove a corollary that is similar to Lemma 11.5.14, with the main difference that we consider nodes that complete virtual round r_2, instead of simply nodes that participate in round r_2. We assume that some node $k \in I_B$ completes virtual round r_2 and designates the round as non-red, and conclude both that there is a unique node that broadcasts a ballot, and also that all nodes that complete round r_2 have the same ballot. The proof relies on Lemma 11.5.3 to indicate that the join protocol works correctly if k does *not* participate.

Corollary 11.5.15. *Assume that $r_2 > 0$ is a virtual round, and $v \in I_V$ is a virtual node, and that there exists some node $k \in I_B$ that completes virtual round r_2 with $E(v).\textit{round-status}[r_2]_k \neq \mathsf{red}$ at the end of round r_2. Then there is exactly one $i \in I_B$ where i begins virtual round r_2 such that:*

1. *Node $i \in I_B$ performs a $\mathsf{bcast}(m, \cdot)_{i,v}$, $m \neq \perp$, in either the* scheduled-ballot *or* unscheduled-ballot *phase of round r_2.*
2. *For all $k' \in I_B$ where k' completes virtual round r_2: at the end of round r_2, $E(v).\textit{ballot}[r_2]_{k'} = E(v).\textit{ballot}[r_2]_i$ at the end of the last agreement phase.*

Proof. Let r_1 be a virtual round such that $\langle r_1, r_2 \rangle$ delineates a finite epoch of v. By Lemma 11.5.3, where $r = r' = r_2$, r_a is the last agreement phase of round r_2 and r_b is the last phase of round r_2, we can conclude that there exists some $j \in I_B$ where j participates in virtual round r and (1) at the end of the last agreement phase, $E(v).\textit{ballot}[r_2]_j = E(v).\textit{ballot}[r_2]_k$, at the end of round r_2, (2) at the end of the last agreement phase, $E(v).\textit{round-status}[r_2]_j = E(v).\textit{round-status}[r_2]_k$, at the end of round r_2.

Thus, since at the end of the last agreement phase, $E(v).\textit{round-status}[r_2]_k$ is not red, we conclude by Lemma 11.5.14 that the conclusions hold for j, and hence by transitivity for k. □

We next prove another corollary extending Lemma 11.5.14 to the case where a virtual round is not a red round. This follows almost immediately from the definition of a red round.

Corollary 11.5.16. *Assume that $r_2 > 0$ is a virtual round that is not a red round for virtual node $v \in I_V$, and that v is up in round r_2. Then there is exactly one $i \in I_B$ such that:*

1. *Node $i \in I_B$ performs a $\mathsf{bcast}(m, \cdot)_{i,v}$, $m \neq \perp$, in either the* scheduled-ballot *or* unscheduled-ballot *phase of round r_2.*
2. *For all j that participate in virtual round r_2, $E(v).\textit{ballot}[r_2]_j = E(v).\textit{ballot}[r_2]_i$ at the end of the last agreement phase of r_2.*
3. *For all j that complete virtual round r_2, at the end of r_2, $E(v).\textit{ballot}[r_2]_j = E(v).\textit{ballot}[r_2]_i$ at the end of the last agreement phase.*

Proof. We discuss each conclusion separately:

1. By Lemma 11.3.10, there exists some $k \in I_B$ that participates in virtual round r_2. Choose some such k. Since r_2 is not a red round, we can conclude that $E(v).\textit{round-status}[r_2]_k \neq \mathsf{red}$. The conclusion then follows from Lemma 11.5.14.

2. Since j participates in virtual round r_2, and round r_2 is not a red round, we can conclude that $E(v).\textit{round-status}[r_2]_j \neq \mathsf{red}$. The conclusion then follows from Lemma 11.5.14.

3. Since j completes virtual round r_2, we conclude by Lemma 11.5.4 that the status $E(v).\textit{round-status}[r_2]_j$ is not red at the end of round r_2. We thus conclude by Corollary 11.5.15 that the claim holds.

□

Since there is a unique "ballot" and a unique "proposer" for each non-red round, we can define the following terms:

Definition 11.5.17. *Assume that $r_2 > 0$ is a virtual round that is not a red round for virtual node $v \in I_V$, and that v is up in round r_2. We say that $m = \langle p, cM, cCD, vM, vCD \rangle$ is the **round** r_2 **ballot** if some $i \in I_B$ broadcasts $\langle \mathsf{vn}, v, m \rangle$ in a ballot phase of r_2. We say that this node i is the **round** r_2 **proposer**. (These are unique by Corollary 11.5.16.)*

Notice that if i is the proposer for some virtual round r, and virtual round r is not red, then $E(v).\textit{ballot}[r]_i$ is equal to the round r ballot at the end of the last agreement phase.

One immediate conclusion we can draw from this series of lemmas and corollaries is that at the end of some virtual round r_2, every node that completes the round has the same unique ballot:

Corollary 11.5.18. *Assume $r_2 > 0$ is a virtual round that is not a red round for virtual node $v \in I_V$. Assume that v is up in round r, and assume that $i, j \in I_B$ are a pair of nodes that complete round r_2. Then $E(v).\textit{ballot}[r_2]_i = E(v).\textit{ballot}[r_2]_j$ at the end of round r_2.*

Proof. The conclusion follows immediately from Corollary 11.5.16: let i' be the (unique) proposer for virtual round r_2; by Corollary 11.5.16, at the end of the last agreement phase of r_2, $E(v).\textit{ballot}[r_2]_{i'} = E(v).\textit{ballot}[r_2]_i$ at the end of round r_2. The same equality holds for node j, and hence the conclusion holds. □

Corollary 11.5.18 describes the situation at the end of some virtual round r_2. In fact, as long as the virtual node is not reset, any two emulators will continue to agree on the ballot for an earlier non-red round, even as new nodes join the emulation and old nodes leave. The main technical tool in this proof is Lemma 11.5.3, which shows that the join protocol works as expected. Combined with the fact that a ballot is otherwise not modified after a virtual round is complete, the conclusion follows from Corollary 11.5.18.

In more detail, we extend Corollary 11.5.18 to show that if v is up in some non-red virtual round r, then throughout the interval $[r, r_2]$ all the nodes agree on the ballot for round r at the end of each virtual round. We begin with a preliminary lemma that compares nodes that complete round r with nodes that complete round r_2, and then give the more general statement as a corollary.

Lemma 11.5.19. *Assume $\langle r_1, r_2 \rangle$ delineates an execution of virtual node $v \in I_V$, and that $r \in [r_1, r_2]$. Moreover, assume that $i, j \in I_B$ are a pair of nodes that complete rounds r and r_2, respectively. If round r, is not a red round, then $E(v).ballot[r]_i$, at the end of round r, $= E(v).ballot[r]_j$ at the end of round r_2.*

Proof. We proceed by backward induction from round r_2 back to round r, where at each stop we decrement r_i and show the following: there exists some $i' \in I_B$ that completes round r_i and at the end of round r_i, $E(v).ballot[r]_{i'} = E(v).ballot[r]_j$ at the end of round r_2. At the end, we invoke Corollary 11.5.18 to draw a conclusion for all i that complete round r.

The base case, where $r_i = r_2$, follows immediately where $i' = j$. For the inductive step, assume that the claim holds for virtual round $r_i + 1$ and that $r \leq r_i < r_2$. By inductive hypothesis there exists some $j' \in I_B$ that completes round $r_i + 1$ and at the end of round $r_i + 1$, $E(v).ballot[r]_{j'} = E(v).ballot[r]_j$ at the end of round r_2.

We now invoke Lemma 11.5.3, where $r = r' = r_i + 1$, r_a is the last agreement phase of $r_i + 1$, and r_b is the last phase of $r_i + 1$. In this case, node j' has $E(v).joined_j = \mathsf{true}$ and $E(v).failed_{j'} = \mathsf{false}$ at the end of round r_b (and $r_i + 1$), as required, since j' completes round $r_i + 1$. Thus we conclude that there exists some $i' \in I_B$ where i' participates in virtual round $r_i + 1$ and at the end of basic round r_a, $E(v).ballot[r]_{i'} = E(v).ballot[r]_{j'}$ at the end of basic round r_b. Since node i' participates in round $r_i + 1$, it also begins round $r_i + 1$, and hence completes round r_i. By transitivity, at the end of round r_i, $E(v).ballot[r]_{i'} = E(v).ballot[r]_j$ at the end of round r_2, proving our inductive claim.

To complete the proof, let $i' \in I_B$ be a node that completes round r and at the end of round r $E(v).ballot[r]_{i'} = E(v).ballot[r]_j$ at the end of round r_2; by induction we have just proved such an i' exists. Since both i and i' complete round r, we conclude by Corollary 11.5.18 that $E(v).ballot[r]_{i'} = E(v).ballot[r]_i$ at the end of virtual round r, concluding our proof. □

Corollary 11.5.20. *Assume $\langle r_1, r_2 \rangle$ delineates an execution of virtual node $v \in I_V$, and that $r' \in [r_1, r_2]$. Moreover, assume that $i, j \in I_B$ are a pair of nodes that complete rounds r' and r_2, respectively. If round $r \in [r_1, r']$ is not a red round, then $E(v).ballot[r]_i$, at the end of round r', $= E(v).ballot[r]_j$, at the end of round r_2.*

Proof. Since virtual node v is up in virtual round r, there exists some $i' \in I_B$ that completes round r. By Lemma 11.5.19, we can conclude that at the end of round r, $E(v).ballot[r]_{i'} = E(v).ballot[r_2]_j$ at the end of round r_2. Since $\langle r_1, r' \rangle$ also delineates a (finite) epoch, we can also conclude by Lemma 11.5.19 that at the end of round r, $E(v).ballot[r]_{i'} = E(v).ballot[r_2]_i$ at the end of round r'. The conclusion follows by transitivity. □

11.6 Calculating the Round Status and State

In this section, and the following Section 11.7, we define the possible executions for some virtual node v that can be extracted from α. Specifically, if $\langle r_1, r_2 \rangle$ delineates a finite epoch of v, our goal is to define $execs(r_1, r_2)_v$, a set of possible executions for the rounds $[r_1, r_2]$, starting in the initial state at the beginning of round r_1 when v is reset. In Section 11.9, we will construct a single infinite execution γ_v of virtual node v by choosing one execution in $execs(u_i, d_i)_v$ for every element $\langle u_i, d_i \rangle$ in the updown sequence, $updown(v)$, where d_i is finite, and concatenating these executions together, along with appropriate fail and recover events, as well as appropriate time passage. (When d_i is infinite, we make use of the constructed executions in a slightly different manner.)

We construct each of the executions in $execs(r_1, r_2)_v$ in two steps: first by calling calculate-status (Figure 10-10) to determine the status of each round, and second by calling calculate-state (Figure 10-11), which performs the actual construction, based on the status previously calculated for each round. (Notice that each process also performs this same two step process in the client phase when determining which message to broadcast in the vn phase.)

In this section we focus on that first step, calculating the status of each round. The main result of this section is Corollary 11.6.7, which shows that after a good virtual round, all the nodes that complete the round calculate the same status array. This corollary is used in Lemma 11.7.11 (in Section 11.7), along with some facts about the calculate-state function, to show that eventually there is only one possible execution in the set $execs(r_1, r_2)_v$, that is, that the replicas converge on a single execution of the virtual node.

A second important lemma from this section is Lemma 11.6.8, which states that if a round is red, then the calculate-status function determines that the round is red. Notice that whether a round is red depends on global information, while the calculate-status function is a local calculation.

This section breaks into three main subsections. First, we prove Lemma 11.6.1, a basic lemma about the calculate-status function which shows certain circumstances in which two nodes will calculate the same round status: i and j both complete a virtual round r_2, and both have the same previous round pointer: $E(v).prev\text{-}rnd_i = E(v).prev\text{-}rnd_j$.

The second part of this section, Section 11.6.2, is a sequence of lemmas attempting to determine when two nodes have the same $E(v).prev\text{-}rnd$ pointer. This sequence of lemmas culminates in Lemma 11.6.6, which shows that in a good round, all participating nodes have the same value of $E(v).prev\text{-}rnd$ at the end of the virtal round.

The third part, Section 11.6.3 contains the two main conclusions of this section: Corollary 11.6.7 and Lemma 11.6.8.

11.6.1 Analysis of calculate-status

In this section, we show that under certain circumstances, two nodes will calculate the same status using the calculate-status function. Specifically, we consider the case

where i and j are two nodes that complete some round r_2, and at the end of r_2, i and j have the same value for the $E(v).prev\text{-}rnd$ pointer. The main structure of the proof is an analysis of the calculate-status function (Figure 10-10), showing that at each iteration of the main loop in calculate-status, i and j perform the same calculations and produce the same results.

Lemma 11.6.1. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v, that $i, j \in I_B$ complete round r_2 for v, and that the following hold at the end of virtual round r_2:*

- $E(v).prev\text{-}rnd_i = E(v).prev\text{-}rnd_j$,
- $status_i = \text{calculate-status} \quad (\quad E(v).rnd_i \ , \quad E(v).prev\text{-}rnd_i \ , \quad E(v).ballot_i \ , \quad E(v).last\text{-}reset_i \ ) \ ,$
- $status_j = \text{calculate-status} \quad (\quad E(v).rnd_j \ , \quad E(v).prev\text{-}rnd_j \ , \quad E(v).ballot_j \ , \quad E(v).last\text{-}reset_j \ ) \ .$

Then for all $r \in [r_1, r_2]$, $status[r]_i = status[r]_j$.

Proof. Our goal is to show that the two computations of calculate-status produce equivalent status arrays for the range $[r_1, r_2]$. The calculate-status calculation consists primarily of a single loop that decrements r in each iteration and fixes $temp\text{-}status[r]$. We argue that at each iteration of the loop, i and j calculate the same status for $temp\text{-}status[r]$. We proceed line-by-line through the loop, showing that certain invariants are maintained. The main technical steps involve applying Corollary 11.5.18 to conclude that both calculations have the same ballot for non-red rounds, and applying Lemma 11.5.12 to conclude that certains rounds are not red.

In more detail, we argue that initially, and after each iteration of the loop, the following invariants hold:

1. $r_i = r_j$,
2. $p_i = p_j$,
3. $temp\text{-}last\text{-}reset_i = temp\text{-}last\text{-}reset_j$.
4. Either v is down in round p_i or p_i is not a red round; the same holds, naturally, for p_j.
5. For all $r \in [r_i, r_2]$, $temp\text{-}status[r]_i = temp\text{-}status[r]_j$.

The subscript i refers to the variables from the calculation of $status_i$, while the subscript j refers to the variables from the calculation of $status_j$.

Before proceeding, we note one initial fact. By Corollary 11.4.5, $E(v).\textit{last-reset}_i + 1 = r_1$. Thus, since r_i is decremented from $E(v).\textit{rnd}_i + 1 = r_2 + 1$ to $E(v).\textit{last-reset}_i + 1 = r_1$, we can conclude that throughout the calculation, $r_1 \leq r_i \leq r_2 + 1$. By the same argument, the same holds for r_j. When the loop terminates, $r_i = r_j = r_1$.

First, we argue that these invariants hold initially at the beginning of the while loop:

1. Notice that $E(v).\textit{rnd}_i = E(v).\textit{rnd}_j = r_2$ by Corollary 11.2.6, and thus initially $r_i = r_j = r_2 + 1$ after they are each assigned *temp-rnd* + 1.

2. By assumption, $E(v).\textit{prev-rnd}_i = E(v).\textit{prev-rnd}_j$, leading to the initial conclusion $p_i = p_j$.

3. By Corollary 11.4.6, $E(v).\textit{last-reset}_i = E(v).\textit{last-reset}_j$, leading to the initial conclusion $\textit{temp-last-reset}_i = \textit{temp-last-reset}_j$.

4. By Lemma 11.5.11, either v is down in $E(v).\textit{prev-rnd}_i$ or the round is not red, leading to the initial conclusion about p_i.

5. $r_i = r_2 + 1 > r_2$, hence for all $r \in [r_i, r_2]$, the claim about the status array is trivially true.

Next, we argue that these invariants are maintained by each iteration of the loop. The first step of the loop is to decrement r_i and r_j. Thus, after the first step (and hence at the end of the loop), $r_i = r_j$, maintaining Invariant 1. We can also now conclude that $r_1 \leq r_i \leq r_2$, and thus v is up in round r_i (and round r_j). As a result, we can conclude by the inductive hypothesis that round p_i (and hence p_j) is not a red round. By Corollary 11.5.18, we know that $\textit{temp-ballot}[p_i] = \textit{temp-ballot}[p_j]$, since *temp-ballot* is a copy of $E(v).\textit{ballot}$ at the end of virtual round r_2.

Next, after r_i and r_j are decremented, the calculation branches, based on whether $r_i = p_i$, and whether $r_j = p_j$. Since $r_i = r_j$ and $p_i = p_j$, both calculations proceed through the same branch of the if clause on line 507.

If they choose the first branch, both p_i and p_j are updated according to the $\textit{temp-ballot}[p_i].\textit{prev-rnd}$ data structure. According to Lemma 11.5.12, if round $r = E(v).\textit{ballot}[p_i].\textit{prev-rnd}_k$, then either round r is not a red round or v is down in round r; thus setting $p_i = r$ maintains Invariant 4 on p_i. Moreover, since prior to updating p_i and p_j, $\textit{temp-ballot}[p_i] = \textit{temp-ballot}[p_j]$, we can conclude that after the update $p_i = p_j$ at the end of the loop, maintaining Invariant 2.

The variable *temp-last-reset* is not modified in either case, and thus Invariant 3 is maintained.

Finally, $\textit{temp-status}_i$ and $\textit{temp-status}_j$ are updated depending on the branch chosen in the loop; as argued above, both calculations choose the same branch in the loop, and hence $\textit{temp-status}[r_i]_i = \textit{temp-status}[r_j]_j$. By induction, we already know that for $r \in [r_i + 1, r_2]$, $\textit{temp-status}[r]_i = \textit{temp-status}[r]_i$; thus, we have maintained Invariant 5. □

11.6.2 Previous Round Pointer Equality

We have concluded that when the *prev-rnd* pointers are equal, two nodes calculate the same status. We now need to show under what conditions nodes will choose the same value for *prev-rnd*. In particular, we are interested in the case where some round r is good, and we want to show that in this case, all nodes agree on the *prev-rnd* pointer.

Lemma 11.6.2 and Corollary 11.6.3 consider the set of nodes that participate in round r; Lemma 11.6.4 and Corollary 11.6.5 consider the set of nodes that complete round r. Thus the primary difference in these two sets of lemmas is that the second set of lemmas include nodes that join during the virtual round, and hence Lemma 11.6.4 includes an argument about the workings of the join protocol. Lemma 11.6.6 depends on these previous lemmas, and draws the desired conclusion that in a good round, all nodes that complete the round agree on the *prev-rnd*.

Our first goal in this progression is to show that in a good round r, every participating node sets *prev-rnd* to r. In fact, we prove a somewhat stronger claim: if some node i participates in round r, then i locally designates the round as green or yellow if and only if $E(v).\textit{prev-rnd}_i = r$. In this section, we use only one direction of this claim: if the status is green or yellow, then *prev-rnd* $= r$. The reverse direction (of Lemma 11.6.4, which follows from Lemma 11.6.2) is used later in Lemma 11.7.14. The proof follows from a straightforward examination of the phases in which *prev-rnd* is modified.

Lemma 11.6.2. *Assume virtual node $v \in I_V$ is up in virtual round $r > 0$, and that some node $i \in I_B$ participates in round r. Then, $E(v).\textit{round-status}[r]_i \in \{\text{green}, \text{yellow}\}$ at the end of the last agreement phase of r if and only if $E(v).\textit{prev-rnd}_i = r$ at the end of the last agreement phase of r.*

Proof. First, assume that the round status is either green or yellow. Since i participates in round r, we know that $E(v).\textit{joined}_i = \text{true}$ through the end of the last agreement phase. Consider the case where v is scheduled: during lines 363–365, $E(v).\textit{prev-rnd}_i$ is updated to $r = E(v).\textit{rnd}_i$ since the round status is green or yellow. Similarly, in the case where v is not scheduled: during lines 427–428, $E(v).\textit{prev-rnd}_i$ is updated to r since the round status is green or yellow.

If the round status is not green or yellow, then $E(v).\textit{prev-rnd}_i$ is not updated during virtual round r, prior to the last agreement phase. It remains to show that initially, at the beginning of the virtual round, $E(v).\textit{prev-rnd}_i \neq r$: by Lemma 11.4.1, we know that $E(v).\textit{prev-rnd}_i < r$ at the beginning of round r, and hence we can conclude that at the end of the last agreement phase, $E(v).\textit{prev-rnd}_i < r$. □

We get the following immediate corollary when round r is good:

Corollary 11.6.3. *Assume virtual node $v \in I_V$ is up in virtual round $r > 0$, and that round r is good for v. If node $i \in I_B$ participates in round r, then $E(v).\textit{prev-rnd}_i = r$ at the end of the last agreement phase of r.*

Proof. By definition $E(v).\textit{round-status}[r]_i \in \{\text{green}, \text{yellow}\}$ at the end of the last agreement phase of r, and hence the claim follows by Lemma 11.6.2. □

The previous lemma and corollary describe the behavior of nodes that participate in some virtual round r. We will be interested, however, not just in nodes that participate in a virtual round, but also those that complete the virtual round. Specifically, some nodes may join the emulation during a virtual round, and the same guarantees need hold at the end of the virtual round. Thus, we next show that at the end of every good round, every node that completes the round has set *prev-rnd* to the current round (Corollary 11.6.5). As before, we begin with a somewhat stronger lemma, Lemma 11.6.4. Notice that these two lemmas essentially extend Lemma 11.6.2 and Corollary 11.6.3 by including an argument that the join protocol works as expected.

Lemma 11.6.4. *Assume virtual node $v \in I_V$ is up in virtual round $r > 0$, and that some node $i \in I_B$ completes round r. Then $E(v).\textit{round-status}[r]_i \in \{\textsf{green}, \textsf{yellow}\}$ at the end of round r if and only if $E(v).\textit{prev-rnd}_i = r$ at the end of r.*

Proof. Since $E(v).\textit{joined}_i = \textsf{true}$ at the end of virtual round r, there are two possible behaviors of node i, both of which can occur: node i participates in v in round r, and/or i joins v in r. (By assumption, i does not reset v in r_2, as v is up in round r_2.)

Assume i joins v in virtual round r. Then in the join-ack phase, node i receives a message from some node k which includes a copy of $E(v).\textit{prev-rnd}_k$ and $E(v).\textit{round-status}[r]_k$ at the beginning of the join-ack phase. Since k has $E(v).\textit{joined}_k = \textsf{true}$ and $E(v).\textit{failed}_k = \textsf{false}$ at the beginning of the join-ack phase, we can conclude that k participates in round r, that is, k has joined and not failed from the beginning of virtual round r through the end of the last agreement phase. By Lemma 11.6.2, we can conclude that $E(v).\textit{round-status}[r]_k$ is green or yellow if and only if $E(v).\textit{prev-rnd}_k = r$. Since both of these values are copied by i, the claim holds for i at the end of round r.

Assume i does not join virtual node v in virtual round r. Then we can conclude that i participates in virtual round r. By Lemma 11.6.2, we can conclude directly that $E(v).\textit{round-status}[r]_i$ is green or yellow at the end of the last agreement phase of r if and only if $E(v).\textit{prev-rnd}_i = r$ at the end of the last agreement phase. The only way that *round-status*$[r]$ and *prev-rnd* change after the last agreement phase, prior to the end of the virtual round, is if i joins or resets v, which we have assumed does not occur in this case. Thus, we can conclude that the claim holds for i at the end of virtual round r. □

We get the following immediate corollary when round r is good:

Corollary 11.6.5. *If virtual node $v \in I_V$ is up in virtual round $r > 0$ and round r is a good round for v, then for every node $i \in I_B$ that completes round r, $E(v).\textit{prev-rnd}_i = r$ at the end of r.*

Proof. Choose r_1 such that $\langle r_1, r\rangle$ delineates a (finite) epoch of v. Since round r is good, we conclude by Lemma 11.5.4 that $E(v).\textit{round-status}[r]_i \in \{\textsf{green}, \textsf{yellow}\}$ at the end of virtual round r. We therefore conclude by Lemma 11.6.4 that $E(v).\textit{prev-rnd}_i = r$ at the end of virtual round r □

Finally, it follows immediately from Corollary 11.6.5 that any two nodes that complete a good round have the same value for *prev-rnd*:

Lemma 11.6.6. *If virtual round $r > 0$ is a good round, then for every pair of nodes $i, j \in I_B$ that complete round r for virtual node v, $E(v).prev\text{-}rnd_i = E(v).prev\text{-}rnd_j$ at the end of virtual round r.*

Proof. For node i, we can conclude by Corollary 11.6.5 that $E(v).prev\text{-}rnd_i = r$ at the end of r. Similarly for j, by Corollary 11.6.5, $E(v).prev\text{-}rnd_j = r$ at the end of r, leading to the desired conclusion. □

11.6.3 Two Conclusions: Calculating the Status

Combining the main lemmas of this section, we can conclude that at the end of any good virtual round, any two nodes that complete the virtual round calculate the same status:

Corollary 11.6.7. *Assume $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v. If round r_2 is a good round, then for every pair of nodes $i, j \in I_B$ that complete round r_2 for v, if:*

- $status_1 = \textsf{calculate-status}\ (\ E(v).rnd_i\ ,\ E(v).prev\text{-}rnd_i\ ,\ E(v).ballot_i\ ,\ E(v).round\text{-}status_i\ ,\ E(v).last\text{-}reset_i\ )$,
- $status_2 = \textsf{calculate-status}\ (\ E(v).rnd_j\ ,\ E(v).prev\text{-}rnd_j\ ,\ E(v).ballot_j\ ,\ E(v).round\text{-}status_j\ ,\ E(v).last\text{-}reset_j\ )$.

Then for every $r \in [r_1, r_2]$, $status[r]_1 = status[r]_2$.

Proof. Follows immediately from Lemmas 11.6.1 and 11.6.6. □

The second conclusion in this section is that the calculate-status function respects the fact that red rounds are, in fact, red. As mentioned earlier, it is important to notice that a round is red based on a global condition, while calculate-status performs a local calculation.

A related fact is that the calculate-status function does not take the *round-status* array as a parameter; it infers all the relevant color information directly from the ballot data structure instead. The *round-status* array is entirely local: each node calculates its round status independently of the other nodes. The ballot data structure, on the other hand, contains information that is shared: a proposer broadcasts a ballot that is then stored in each node's ballot data structure. Thus, the calculate-state function uses the data in the ballot data structure, rather than the local round status array, in order to approximate a global condition, i.e., whether the round is red or green.

Another relevant fact about this lemma is that it describes the result of a calculation about some round r' not only at the end of round r', but also at the end of some

later round r_2, assuming the virtual node is not reset between rounds r' and r_2. This fact, however, should be unsurprising, given the already-discussed persistence of the round color designation (e.g., Lemma 11.5.5).

Lemma 11.6.8. *Assume $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v, and i completes virtual round r_2. If $r' \in [r_1, r_2]$ is a red round for v, and at the end of round r_2:*

- $status_i = \mathsf{calculate\text{-}status}\ (\ E(v).rnd_i\ ,\ E(v).prev\text{-}rnd_i\ ,\ E(v).ballot_i\ ,\ E(v).last\text{-}reset_i\ )\ ,$

then $status[r'] = \mathsf{red}$.

Proof. The proof follows from the way in which calculate-status uses the *prev-rnd* pointers in the ballot. In each iteration of the loop, either the previous round pointer p is left unchanged, or it is updated as follows:

$$p \leftarrow ballot[p].prev\text{-}rnd\ .$$

Since $ballot[p].prev\text{-}rnd$ is always a non-red round (by Lemma 11.5.12), we can conclude that round p is always a non-red round. Moreover, in each iteration, $status[r]$ is set to green if $r = p$ and to red otherwise. Thus if $status[r]$ is green, then the round is non-red, proving the contrapositive of the desired lemma.

We now proceed in more detail. Assume for the sake of contradiction that $status[r'] = \mathsf{green}$. (Notice in calculate-status, the returned array contains only red and green colors.) Moreover, $status[r'] = \mathsf{green}$ only if $temp\text{-}status[r'] = \mathsf{green}$ at the end of the calculation; $temp\text{-}status[r']$ is set to green only if at some point during the calculation $p = r'$. We therefore show that the following invariant holds throughout the calculate-status calculation: the virtual round p is not red. This claim implies that at no point during the calculation does $p = r'$, which results in a contradiction, proving the lemma.

Initially, the invariant holds since p is set to $temp\text{-}prev$, which is equal to $E(v).prev\text{-}rnd_i$ at the end of round r_2; by Lemma 11.5.11, we can conclude that p is not a red round. Inductively, in each iteration of the loop, if $p = r$, then p is set equal to $temp\text{-}ballot[r].prev\text{-}rnd$. By Lemma 11.5.12, we know that round $E(v).ballot[r].prev\text{-}rnd_i$, at the end of round r_2, is not a red round, from which we conclude that the invariant holds, which concludes the proof. □

11.7 Virtual Node Executions

In this section, we begin the process of constructing executions of virtual nodes. As described at the beginning of Section 11.6, our goal is to extract from α the possible executions of each virtual node. In this section, for every $\langle r_1, r_2 \rangle$ that delineates a finite epoch for virtual node v, we define the set $execs(r_r, r_2)_v$ of possible executions

(Definition 11.7.7). Each node that completes a virtual round may have its own local view of the virtual node's execution; the set $execs(r_1, r_2)_v$ contains the local views of all nodes that complete virtual round r_2. When there is only one execution in $execs(r_1, r_2)_v$, i.e., when $|execs(r_1, r_2)_v| = 1$, then all the nodes have converged on a single execution for v.

We begin in Section 11.7.1 with some preliminaries. We define an $[r_1, r_2]$-execution fragment (Definition 11.7.1), and a $[r_1, r_2]$-execution (Definition 11.7.2). An "execution" is a well-defined TIOA concept; an $[r_1, r_2]$-execution is simply an execution that runs from round r_1 through the end of round r_2. We then examine how to construct such an execution. Specifically, the functions do-bcast and do-recv (see Figure 10-12) together construct a one round execution, which we prove in Lemma 11.7.3. Thus, when called repeatedly, they can construct a $[r_1, r_2]$-execution, as is implied by Lemma 11.7.5.

In Section 11.7.2, we define a function $exec_v : \mathbb{N} \times \mathbb{N} \times I_B \rightarrow$ executions of v. Specifically, if $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v, and $i \in I_B$, then $exec(r_1, r_2, i)_v$ returns an $[r_1, r_2]$ execution of virtual node v that is consistent with i's local view at the end of round r_2. We show in Lemma 11.7.10 that $exec(r_1, r_2, i)_v$ is in fact an $[r_1, r_2]$-execution. We also define (Definition 11.7.7) the set $execs(r_1, r_2)_v$, which is the set of all $exec(r_1, r_2, i)_v$, for all $i \in I_B$ that complete round r_2.

In the rest of Section 11.7, we present various properties of these constructed executions. In Section 11.7.3, we examine the case where a round is good. We show that after a good round for virtual node v, every node participating in the emulation has the same local view of virtual node v's execution (Lemma 11.7.11). That is, $exec(r_1, r_2, i)_v = exec(r_1, r_2, j)_v$, for all i and j that complete round r_2. This implies that $|execs(r_1, r_2)_v| = 1$ (Corollary 11.7.12).

In later sections of this chapter (e.g., Sections 11.11 and 11.12) we prove a variety of properties about these executions. For example, along with the client executions, they satisfy integrity, collision detector completeness, and collision detector eventual accuracy. In Section 11.7.4, we prove one simple property about the constructed executions: they each satisfy the self-delivery property. That is, if a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event occurs in $exec(r_1, r_2, i)_v$, then message m is received on port $\langle v, * \rangle$ in round r.

In Section 11.7.5, we prove a series of lemmas regarding these executions and their extensions. Our eventual goal is to construct a single execution of virtual node v via successive extension. Thus, in an ideal situation, $exec(r_1, r_2, i)_v$ would be a one-round extension of $exec(r_1, r_2 - 1, i)_v$, which itself would extend $exec(r_1, r_2 - 2, i)_v$, and so on. In this way, we could construct an entire execution by calling *exec* repeatedly to determine the next round. Unfortunately, the emulator cannot guarantee this desired behavior. For example, i may join the emulation in some round $r > r_1$, and thus $exec(r_1, r, i)_v$ cannot extend $exec(r_1, r-1, i)_v$. Alternatively, different nodes may disagree in their local view of virtual node v's execution; in order for the executions to converge to a single execution, it may be necessary for some emulators to adopt other emulators' executions.

In Section 11.7.5, then, we prove the following key claim: if $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v, and if $r_2 > r_1$, then for every $i \in I_B$, there exists some $j \in I_B$ that completes round $r_2 - 1$ such that $exec(r_1, r_2, i)_v$ extends $exec(r_1, r_2 - 1, j)_v$.

That is, after round r_2, node i's view of virtual node v's execution is a one round extension of some node j's view of virtual node v's execution after round $r_2 - 1$. We prove this claim in two steps, first considering good rounds (Corollary 11.7.15), and then considering bad rounds (Corollary 11.7.18), before rearching the desired conclusion in Lemma 11.7.19. This argument is applied inductively in Lemma 11.7.20 to generate multi-round execution extensions.

Finally, in Lemma 11.7.21, we return to considering the behavior of good rounds, combining Lemma 11.7.19 and Corollary 11.7.15 to show that if a round r is good and if node i is the round r proposer, then every node extends node i's view of virtual node v's execution up through round r. Thus, in a good round, the proposer fixes the virtual node's execution up through round r.

11.7.1 Preliminaries

We begin by defining an $[r_1, r_2]$-execution fragment. Recall from Appendix A that an execution fragment of a TIOA is a (finite or infinite) alternating sequence of trajectories and actions with certain properties: it begins with a trajectory τ_0, each trajectory τ_i is a trajectory of the TIOA, and the state at the beginning of each τ_i, $i > 0$, reflects the state at the end of τ_{i-1} and the intervening action. An execution is simply an execution fragment that begins with an initial state of the TIOA.

We define an $[r_1, r_2]$-execution fragment to be an execution fragment of virtual node v for rounds $[r_1, r_2]$. We place two restrictions on the execution fragment: (1) It must end with a recv event, since the last event in a virtual round is a recv; and (2) For every virtual round that is included in the execution fragment, there is exactly one recv event for v, since each virtual round in an execution includes one recv event. (Notice that we do not need to place any restrictions on bcast events since the definition of a process, Definition 8.1.2, ensures that each non-terminal recv event is follows by a bcast event.)

Definition 11.7.1. *Let $r_1, r_2 \in \mathbb{N}$, $r_2 \geq r_1$. We define an $[r_1, r_2]$-**execution fragment** γ of virtual node v to be an execution fragment of the automaton $\mathsf{Remap}(A_V(v), v)$ with starting time $(r_1 - 1) \cdot RndLength_V$ with the following properties:*

1. *The last event in γ is a round r_2 receive event for v.*
2. *For every $r \in [r_1, r_2]$, γ contains exactly one round r recv for virtual node v, and no other recv events.*

Notice that our definition of an $[r_1, r_2]$-execution fragment of v does not specify whether a round r_1 broadcast occurs. (For all rounds $r : r_1 < r < r_2$, a round r bcast event is required, since every execution of a process has a bcast immediately following every recv event.) We say that a $[r_1, r_2]$-execution fragment is **broadcast complete** if it includes a round r_1 broadcast, and **broadcast incomplete** otherwise. Every execution of a process begins with a broadcast incomplete round: a trace of a process always begins with a recv event (i.e., a dummy message), as it broadcasts a message only in response to receiving a message from the previous round. (See Section 8.1.5 for an explanation of how the broadcast service drives the round structure.)

Thus, an $[r_1, r_2]$-execution of v is an $[r_1, r_2]$-execution fragment that begins in the virtual node's initial state, $start_v$ and that has no round r_1 bcast event.

Definition 11.7.2. *Let $r_1, r_2 \in \mathbb{N}$, $r_2 \geq r_1$. An $[r_1, r_2]$-**execution** γ of virtual node v is a broadcast incomplete $[r_1, r_2]$-execution fragment of v in which $fstate(\gamma) = start_v$.*

Throughout the rest of this section, we will be constructing $[r_1, r_2]$-executions. The basic algorithm we use for constructing these executions is to repeatedly call the functions $\mathsf{do\text{-}bcast}_v$ and $\mathsf{do\text{-}recv}_v$ (see Figure 10-12). The function $\mathsf{do\text{-}bcast}_v$ inserts a $\mathsf{bcast}_{v,*}$ event, while the function $\mathsf{do\text{-}recv}_v$ inserts a single round of time passage, followed by a $\mathsf{recv}_{v,*}$ event. Together, these two functions construct a one round, broadcast-complete execution fragment for virtual node v:

Lemma 11.7.3. *If $s \in states_v$ is a state of virtual node $v \in I_V$ such that s is the last state of an execution, γ, which has a* recv *event as its last action, and if:*

- $\langle s', m, r, e_1 \rangle \leftarrow \mathsf{do\text{-}bcast}(s)$
- $\langle s'', e_2 \rangle \leftarrow \mathsf{do\text{-}recv}(s', M, cd, cm)$.

Then for every $r > 0$, $e_1.e_2$ is a broadcast-complete $[r, r]$-execution fragment of v and s'' is the last state of e_2, i.e., the state of the virtual node at the end of round r.

Proof. The lemma follows immediately by inspection. Notice that do-bcast simulates virtual node v broadcasting a message, after some number of internal actions. Similarly, do-recv simulates one round of time passage (including any internal actions), followed by v receiving a set of messages. Notice also that both functions terminate, producing a finite execution, as per Lemmas 10.3.1 and 10.3.2. Thus the criteria for a $[r, r]$-execution fragment are met. □

Along the same lines, the do-recv function constructs a one round broadcast-incomplete execution of a virtual node:

Lemma 11.7.4. *If $s \in states_v$ is a state of virtual node $v \in I_V$ such that s is the last state of an execution, γ, which does not end in a* recv *event, and if:*

- $\langle s', e \rangle \leftarrow \mathsf{do\text{-}recv}(s, \cdot, \cdot\cdot)$.

Then for every $r > 0$, e is a broadcast-incomplete $[r, r]$-execution fragment of v and s' is the last state of e, i.e., the state of the virtual node at the end of round r.

Proof. As in the Lemma 11.7.3, the lemma follows immediately by inspection. Notice that do-recv simulates one round of time passage (including any internal actions), followed by v receiving a set of messages. Thus the criteria for a broadcast-incomplete $[r, r]$-execution fragment are met. □

We can thus conclude from Lemma 11.7.4 that if $s = start_v$, the initial state of v, then do-recv constructs a one round execution of v.

Next, notice that by concatenating executions and execution fragments, we can construct longer executions:

Lemma 11.7.5. *If γ is an $[r_1, r_2]$-execution of virtual node v, and γ' is a broadcast-complete $[r_2+1, r_3]$-execution fragment of virtual node v, for some $r_2 \geq r_2 + 1$, and if the last state of γ is equal to the first state of γ', then $\gamma.\gamma'$ is an $[r_1, r_3]$-execution of virtual node v.*

Proof. Immediate by the definition of execution and execution fragment. □

11.7.2 Constructing an $[r_1, r_2]$ Execution

We now show how to construct an $[r_1, r_2]$-execution for virtual node v. Specifically, for every $i \in I_B$ that completes virtual round r_2, we construct node i's local view of virtual node v's execution. We later show, in Section 11.7.3, that after a green round, all the nodes have the same local view of virtual node v's execution.

We define the function $exec(r_1, r_2, i)_v$, which constructs an $[r_1, r_2]$-execution fragment for virtual node v, according to node i. The construction process takes two steps: first, it calls calculate-status (Figure 10-10) to determine the status of each round, and second, it calls calculate-state (Figure 10-11) to actually construct the execution. The calculate-state function calls do-bcast and do-recv repeatedly to construct the individual rounds.

Definition 11.7.6. *Assume $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$, and $i \in I_B$ is a node. Define $\boldsymbol{exec(r_1, r_2, i)_v}$ as follows, where each variable is referenced at the end of virtual round r_2:*

- *status* = calculate-status ($E(v).rnd_i$,
  $E(v).prev\text{-}rnd_i$,
  $E(v).ballot_i$,
  $E(v).last\text{-}reset_i$) ,
- $\langle$ *s,e* $\rangle$ = calculate-state ($E(v).rnd_i$,
  status,
  $E(v).ballot_i$,
  $E(v).last\text{-}good\text{-}state_i$,
  $E(v).last\text{-}reset_i$)$_v$.

*Then, $exec(r_1, r_2, i)_v = e$. We also define the **calculate-last state** of $exec(r_1, r_2, i)_v$ to be equal to s, the state returned by the* calculate-state *function.*

There are a few things to note about this definition. First, by Corollary 11.4.5, if i completes virtual round r_2, then $r_1 = last\text{-}reset + 1$. (Round r_1 does not play any other role in the calculation.)

Next, notice that in the client phase, when node i decides whether to broadcast a message in the following vn phase (lines 235–240), it calculates the last state of $exec(r_1, r_2, i)_v$ using the same two-step process described above, and then calls the do-bcast function to see if a broadcast is enabled in this state. It is for this reason that we describe $exec(r_1, r_2, i)_v$ as i's view of virtual node v's execution: i bases its decisions on this constructed execution.

We also define the set of possible executions:

Definition 11.7.7. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v in α. We define* $\boldsymbol{execs(r_1, r_2)_v}$ *to be the set of executions:*

$$\{exec(r_1, r_2, i)_v : i \in I_B, i \text{ completes round } r_2\} .$$

Notice the following straightforward observation:

Lemma 11.7.8. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v in α. If virtual node v is up in round r_2, then $execs(r_1, r_2)_v \neq \emptyset$.*

Proof. Since virtual node v is up in round r_2, there exists some node $i \in I_B$ that completes round r_2, and hence $exec(r_1, r_2, i) \in execs(r_1, r_2)_v$. □

We proceed to argue that $exec(r_1, r_2, i)_v$ is in fact an $[r_1, r_2]$-execution for v. The key point is that the calculate-state$_v$ function always returns an execution for virtual node v, and that this execution contains a recv event for each round. In fact, calculate-state$(r_2, \cdot, \cdot, \cdot, r_1 - 1)_v$ always returns an $[r_1, r_2]$-execution.

At the same time, we also show that the calculated-last state of $exec(r_1, r_2, i)_v$ is in fact the last state of $exec(r_1, r_2, i)_v$. That is, the function calculate-state returns a two-element pair: a state and an execution; in Lemma 11.7.10, we show that the state returned by the function calculate-state is in fact the last state of the execution returned by calculate-state.

We first need the following straightforward fact:

Invariant 11.7.9. *For all $i \in I_B$, for all $v \in I_V$, $E(v).last\text{-}good\text{-}state_i = start_v$.*

Proof. Immediate, by inspection, as $E(v).last\text{-}good\text{-}state_i$ is never modified throughout the execution. □

We now proceed to prove the claim that $exec(r_1, r_2, i)_v$ is an $[r_1, r_2]$-execution of v:

Lemma 11.7.10. *Assume $\langle r_1, r_2 \rangle$ delineates a finite epoch of v. Assume i completes round r_2. Then $exec(r_1, r_2, i)_v$ is an $[r_1, r_2]$-execution of v, and the calculated-last state of $exec(r_1, r_2, i)_v$ is $\ell state(exec(r_1, r_2, i)_v)$.*

Proof. The proof follows by inspection of calculate-state, *Figure* $10 - 11$. We observe that the calculate-state function calls do-recv and do-bcast in an alternating fashion, thus producing a broadcast-incomplete execution fragment of v. That is, the calculate-state function constructs the $[r_1, r_2]$-execution one round at a time, beginning with the empty execution and adding a one round execution fragment for each round from round r_1 up to round r_2. We then notice that the initial state of the execution is the the initial state of v, concluding that the resulting fragment is in fact an execution.

In more detail, calculate-state operates as follows. Initially, we know that *temp-state* = *last-good-state* = $start_v$ (as per Invariant 11.7.9), the state of the virtual node after an empty execution. Also, notice that initially *temp-rnd* = $E(v).rnd_i = r_2$ (again, by Lemma 11.2.4). By Corollary 11.4.5, *temp-last-reset* = $E(v).last\text{-}reset_i = r_1 - 1$.

calculate-state begins by constructing a round $[r_1, r_1]$-execution by calling do-recv, as per Lemma 11.7.4, where *temp-state* is the initial state of v. It proceeds iteratively from $r = r_1 + 1$ up to $r = r_2$. At each step, *exec* maintains the inductively constructed execution, and *temp-state* maintains the last state of *exec*.

For each $r \in [r_1 + 1, r_2]$, the protocol constructs a $[r, r]$-execution fragment for v, beginning in the last state of the inductively constructed execution, *temp-state*. calculate-state invokes do-bcast, followed by do-recv, creating a broadcast-complete execution fragment, as per Lemma 11.7.3. (At each step, the last event in the inductively constructed execution is a recv event.) This fragment is then appended to the inductively constructed execution, as per Lemma 11.7.5. When $r = r_2$, the construction is complete. It follows immediately that *temp-state* is the last state of the execution *exec*, as required.

Finally, notice that the initial state of $exec = start_v$, the initial state of v, and hence the fragment *exec* is an execution.

□

11.7.3 Only One Good $[r_1, r_2]$-Execution

In this section, we show a result that is important in proving that the agreement protocols converge to a single execution when the system is "well-behaved." Specifically, we show that in a good round r, every node that completes round r calculates the same execution.

Lemma 11.7.11. *Assume $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v. If round r_2 is a good round, then for every pair $i, j \in I_B$ that complete round r_2 for virtual node v: $exec(r_1, r_2, i) = exec(r_1, r_2, j)$.*

Proof. Each *exec* is calculated in a two-step process by first calling calculate-status to determine $status_i$ and $status_j$, and then by calling calculate-state. We argue that the input parameters to calculate-state are identical, with only one exception: $E(v).ballot[r]_i$ and $E(v).ballot[r]_j$ can differ in red rounds. We argue that the calculate-status function does not examine the *ballot* array for red rounds, and hence the calculations proceed identically despite this difference.

In more detail, consider the parameters to calculate-state:

- *temp-rnd*: By Corollary 11.2.6, $temp\text{-}rnd_i = temp\text{-}rnd_j$.
- *temp-status*: By Corollary 11.6.7, for every $r \in [r_1, r2]$, $temp\text{-}status[r]_i = temp\text{-}status[r]_j$. The calculate-state function does not access the *temp-status* array outside this range.
- *temp-ballot*: For $r \in [r_1, r_2]$, if $temp\text{-}status[r]_i =$ green, then $temp\text{-}ballot[r]_i = temp\text{-}ballot[r]_j$. This claim follows because if $temp\text{-}status[r]_i =$ green, then we know by the contrapositive of Lemma 11.6.8 that virtual round r is not red; by Corollary 11.5.20, we know that if virtual round r is not red, then the ballot arrays are equal.

- *temp-last-reset*: By Corollary 11.4.5, $\textit{temp-last-reset}_i = \textit{temp-last-reset}_j = r_1 - 1$.

- *temp-state*: By Invariant 11.7.9, $\textit{temp-state}_i = \textit{temp-state}_j = \textit{start}_v$.

We conclude the proof by noticing that the calculate-state function only accesses *temp-ballot*$[r]$, for some $r > 0$, if *temp-status*$[r]$ = green (see line 545, Figure 10-11). Thus the two calculations proceed identically, and produce the same execution, as desired. □

We get the following immediate corollary, restating Lemma 11.7.11 in terms of the set *execs*:

Corollary 11.7.12. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v, and round r_2 is green for v. Then $|\mathsf{execs}(r_1, r_2)_v| = 1$.*

Proof. By Corollary 11.5.7, we know that round r_2 is a good round. By Lemma 11.7.11, if i and j are participating in round r_2 and round r_2 is good, then $\textit{exec}(r_1, r_2, i) = \textit{exec}(r_1, r_2, j)$, concluding the proof. □

11.7.4 Message Self-Delivery

Later in Sections 11.11 and 11.12, we prove a variety of properties of the executions we have constructed. These later properties, however, involve interactions among different virtual nodes and clients. Here we prove one simple property involving an individual virtual node: each execution satisfies the self-delivery property. That is, if a virtual node v broadcasts a message m in a virtual round r, then it also receives message m in round r.

Lemma 11.7.13. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$. Assume $\gamma \in \textit{execs}(r_1, r_2)_v$. If for some $m \in \textit{msgs}_V$, a round r_2 $\mathsf{bcast}(m, \cdot)_{v,*}$ occurs in γ, then a round r_2 $\mathsf{recv}(M, \ldots)_{v,*}$ event occurs in γ in which $m \in M$.*

Proof. This lemma follows immediately from the calculate-state function. Fix i such that $\gamma = \textit{exec}(r_1, r_2, i)$, which is calculated by calling calculate-state at the end of round r_2. Notice that if a bcast event occurs in γ, it is added by a call to do-bcast in either line 554 or line 564. In both cases, the do-bcast function returns the message broadcast, *temp-msg*. When constructing round r_2, *temp-msg* $= m$. In both cases, the *temp-msg* message is added to the *inM* set, which is then used as a parameter to do-recv, thus ensures that m is in the set of messages received. □

11.7.5 Every Execution Extends an Earlier Execution

A key property of an execution $\textit{exec}(r_1, r_2, i)_v$ is that if $r_2 > r_1$ and if i completes round r_2, then we can conclude that $\textit{exec}(r_1, r_2, i)_v$ extends execution $\textit{exec}(r_1, r_2 - 1, j)_v$, for some j that completes round $r_2 - 1$. As a result of this property, we can

construction executions of v through successive extension: if we take a sequence of sets of executions:

$$\langle execs(r_1, r_1)_v, execs(r_1, r_1 + 1)_v, \ldots, execs(r_1, r_2)_v \rangle \ ,$$

then we can conclude that every execution $\gamma \in execs(r_1, r_2)_v$ has a prefix in each of the sets in the sequence.

In this section, we prove two slightly stronger lemmas that identify exactly which execution is extended, based on the color that i designates for round r_2. We then conclude with the main result of this section, Lemma 11.7.19, which shows the key property described above.

First, we consider the case where a node $j \in I_B$ designates round r_2 to be either green or yellow. As a result, we can conclude that round r_2 is not a red round, and hence has a unique proposer (Corollary 11.5.15), which we designate i. In this case, we claim that the execution constructed by node j is an extension of the execution constructed by node i.

Lemma 11.7.14. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$ and that $r_2 > r_1$. Assume $j \in I_B$ completes round r_2 and has $E(v).round\text{-}status[r]_j \in \{\mathsf{green}, \mathsf{yellow}\}$ at the end of r_2. Moreover, assume that node $i \in I_B$ is the round r_2 proposer. (See Definition 11.5.17; as result of the existence of i we know that such an i exists.) Then $exec(r_1, r_2, j)_v$ is a one-round extension of $exec(r_1, r_2 - 1, i)$.*

Proof. Let $\gamma_i = exec(r_1, r_2 - 1, i)$ and $\gamma_j = exec(r_1, r_2, j)$. Our goal is to show that γ_i is a prefix of γ_j. The proof proceeds in three parts. First, we recall the construction of γ_i and γ_j via the calculate-status and calculate-state functions. Second, we examine the calculate-status function and claim that i and j calculate the same round status for each of the rounds in the range $[r_1, r_2 - 1]$. This second part of the proof closely resembles the proof for Lemma 11.6.1. Third, we examine the calculate-state function and claim that as a result, both constructions produce the same execution prefix from round r_1 through round $r_2 - 1$. The main fact used here is that the invocations to calculate-state differ (primarily) in the *ballot* data structure; however calculate-state examines the round r $ballot[r]$ only when round r is not red, and hence by Corollary 11.5.18, the ballots for i and j are equal.

Part 1: Recall the construction of γ_i and γ_j. The two executions γ_i and γ_j are each constructed in two steps: first by calling calculate-status, and then by calling calculate-state. Specifically, for $\gamma_i = exec(r_1, r_2 - 1, i)$:

- $status_i =$calculate-status $($ $E(v).rnd_i - 1$, $E(v).prev\text{-}rnd_i$, $E(v).ballot_i$, $E(v).last\text{-}reset_i)$,
- $\langle s, e \rangle =$calculate-state $($ $E(v).rnd_i - 1$, $status_i$, $E(v).ballot_i$, $E(v).last\text{-}good\text{-}state_i$, $E(v).last\text{-}reset_i)_v$.

In this case, each variable is referenced at the end of virtual round $r_2 - 1$. Similarly, for $\gamma_j = exec(r_1, r_2, j)$:

- $status_j$ = calculate-status ($E(v).rnd_j$, $E(v).prev\text{-}rnd_j$, $E(v).ballot_j$, $E(v).last\text{-}reset_j$) ,
- $\langle$ s,e $\rangle$ = calculate-state ($E(v).rnd_j$, $status_j$, $E(v).ballot_j$, $E(v).last\text{-}good\text{-}state_j$, $E(v).last\text{-}reset_j$)$_v$.

In this case, each variable is referenced at the end of virtual round r_2.

Part 2: Both nodes calculate the same round status. Our first goal is to prove the following claim: for all $r : r_1 \leq r \leq r_2 - 1$, $status[r]_i = status[r]_j$. That is, for the rounds in the range $[r_1, r_2 - 1]$, both nodes i and j calculate the same round status. This part is the main technical piece of the lemma, and closely resembles the proof of Lemma 11.6.1.

The calculate-status calculation consists primarily of a single loop that decrements r in each iteration and fixes $temp\text{-}status[r]$. We argue that at each iteration of the loop, both i and j calculate the same status for $temp\text{-}status[r]$. We proceed line-by-line through the loop, showing that certain invariants are maintained. The main technical steps involve applying Corollary 11.5.18 to conclude that both calculations have the same ballot for non-red rounds, and applying Lemma 11.5.12 to conclude that certains rounds are not red.

In more detail, we compare the two calculations in the following manner: we examine the initial state of the calculation for $status_i$, and the state of the calculation for $status_j$ after the first iteration of the loop. We show that in this situation, a set of invariants hold. Moreover, from then onwards, after each iteration of the loop, the invariants are maintained. Throughout, we are comparing iteration k of the calculation for i with iteration $k + 1$ of the calculation for j. We consider the following invariants:

1. $r_i = r_j$,
2. $p_i = p_j$,
3. $temp\text{-}last\text{-}reset_i = temp\text{-}last\text{-}reset_j$.
4. Either v is down in round p_i or p_i is not a red round; the same holds, naturally, for p_j.
5. For all $r \in [r_i, r_2 - 1]$, $temp\text{-}status[r]_i = temp\text{-}status[r]_j$.

The subscript i refers to the variables from the calculation of $status_i$, while the subscript j refers to the variables from the calculation of $status_j$.

Consider the initial state of the $status_i$ calculation and the state of the $status_j$ calculation after the first iteration through the calculate-status loop:

1. Round r counter: Initially, $r_i = r_2 = \textit{temp-rnd}_i + 1$ (by Lemma 11.2.4,as usual); after the first iteration $r_j = r_2$ (by Lemma 11.2.4), as r_j has been decremented once from $r_2 + 1$. Thus $r_i = r_j$.

2. Previous round pointer p: Initially, $p_i = E(v).\textit{prev-rnd}_i$ at the end of round $r_2 - 1$; we now argue that p_j after the first iteration of the calculate-status loop is also equal to $E(v).\textit{prev-rnd}_i$ at the end of round $r_2 - 1$.

   Initially, $\textit{temp-prev}_j = \textit{temp-rnd}_j = r_2$ by Lemma 11.6.4, since round r_2 is a good round. Thus initially, $p_j = r_2$. As discussed above, initially, $r_j = r_2 + 1$, and r_j is immediately decremented to equal r_2 (line 506, Figure 10-10). Next, in line 507, p_j is compared to r_j. Since $p_j = r_j$, the first branch of the if statement is executed, updating $p_j = \textit{ballot}[r_2].\textit{prev-rnd}$. It thus remains to show that $\textit{ballot}[r_2].\textit{prev-rnd} = p_i$.

   Recall that i is the unique proposer of a ballot in round r_2. (By Corollary 11.5.15 we know that there is only one proposer since round r_2 is good.) Thus the ballot which is broadcast in the ballot phase for v in r_2 (whether it is scheduled or unscheduled) contains $E(v).\textit{prev-rnd}_i$ at the beginning of virtual round r_2, i.e., the end of virtual round $r_2 - 1$. Therefore, $E(v).\textit{ballot}[r_2].\textit{prev-rnd}_j$ at the end of round r_2 is equal to $E(v).\textit{prev-rnd}_i$ at the beginning of round r_2, by which we conclude that p_i, initially, is equal to p_j, after the first iteration of the loop.

3. Last good round *temp-last-reset*: In both calculation, $\textit{temp-last-reset} = r_1 - 1$ both initially and throughout the calculation, as per Corollary 11.4.5.

4. Either v is down in round p_i, or round p_i is not red: By Lemma 11.5.11, either v is down in $E(v).\textit{prev-rnd}_i$ or the round is not red, leading to the initial conclusion about p_i. Since after the first iteration, $p_j = p_i$ initially, the claim holds also for p_j.

5. Temporary status array *temp-status*: $r_i = r_2 > r_2 - 1$, hence for all $r \in [r_i, r_2 - 1]$, the claim about the status array is trivially true.

We argue that although the *temp-ballot* arrays may differ, in fact the calculations proceed in an identical manner, and thus the invariants are maintained throughout the calculation. The argument is nearly identical to that from Lemma 11.6.1.

The first step of the loop is to decrement r_i and r_j. Thus, after the first step (and hence at the end of the loop), $r_i = r_j$, maintaining Invariant 1. We can also now conclude that $r_1 \leq r_i \leq r_2 - 1$, and thus v is up in round r_i (and round r_j).

Next, after r_i and r_j are decremented, the calculation branches, based on whether $r_i = p_i$, and whether $r_j = p_j$. Since $r_i = r_j$ and $p_i = p_j$, both calculations proceed through the same branch of the if clause on line 507.

If they choose the first branch, both p_i and p_j are updated according to the $\textit{temp-ballot}[p_i].\textit{prev-rnd}$ data structure. Since $p_i = r_i$, we can conclude that v is up in round p_i, and hence that $\textit{temp-ballot}[p_i] = \textit{temp-ballot}[p_j]$ by Corollary 11.5.18.

According to Lemma 11.5.12, if $r = E(v).ballot[p_i].prev\text{-}rnd_k$, then either round r is not a red round or v is down in round r; thus setting $p_i = r$ maintains Invariant 4 on p_i. Moreover, since prior to updating p_i and p_j, $temp\text{-}ballot[p_i] = temp\text{-}ballot[p_j]$, we can conclude that after the update $p_i = p_j$ at the end of the loop, maintaining Invariant 2.

The variable *temp-last-reset* is not modified in either case, and thus Invariant 3 is maintained.

Finally, $temp\text{-}status_i$ and $temp\text{-}status_j$ are updated depending on the branch chosen in the loop; as argued above, both calculations choose the same branch in the loop, and hence $temp\text{-}status[r_i]_i = temp\text{-}status[r_j]_j$. By induction, we already know that for $r \in [r_i+1, r_2-1]$, $temp\text{-}status[r]_i = temp\text{-}status[r]_i$; thus, we have maintained Invariant 5.

Part 3: Examining calculate-state *and the construction of* γ_i *and* γ_j. We can now conclude the proof by examining the calculate-state function. Specifically, we want to examine the two executions γ_i and γ_j constructed by the following:

- $\langle \cdot, \gamma_i \rangle \leftarrow calculate\text{-}state(r_2 - 1, status_i, E(v).ballot_i, r_1 - 1, start_v)_v$.
- $\langle \cdot, \gamma_j \rangle \leftarrow calculate\text{-}state(r_2, status_j, E(v).ballot_j, r_1 - 1, start_v)_v$.

There are three ways in which these invocations differ. First, γ_i begins with $temp\text{-}rnd_i = r_2 - 1$, while γ_j begins with $temp\text{-}rnd = r_2$. Since *temp-rnd* controls the number of times the loop is repeated, this indicates that γ_j is one round longer than γ_i. The second difference is in the *status* array: we have argued above that for all $r : r_1 \leq r \leq r_2 - 1$, $status_i = status_j$; for round outside this range, the array may differ. Third, the *temp-ballot* arrays may differ. As before, if some virtual round $r \in [r_1, r_2 - 1]$ is not red, then we know by Corollary 11.5.20 that $temp\text{-}ballot[r]_i = temp\text{-}ballot[r]_j$. We argue that despite these differences, from round r_1 through round $r_2 - 1$, both calculations constructs the same $[r_1, r_2 - 1$-execution, and hence γ_j is a one-round extension of γ_i.

In more detail, notice that the calculate-state function accesses the ballot array only when $temp\text{-}status[r] =$ green. By Lemma 11.6.8, however, we know that $temp\text{-}status[r]$ is green only when virtual round r is not red, as *temp-status* is a result of the calculate-status calculation. Thus, whenever the two calculations access the ballot array, the value are the same.

Thus, in each iteration of the for loop, beginning with $r = r_1$ and up until $r = r_2 - 1$, the two invocations proceed in the same manner: first, calculating *inCM* based on whether v is scheduled in round r; branching, based on whether $temp\text{-}status[r]$ is green or red; if the status is green, they both construct a round based on the ballot data structures, which are the same; if the status is red, they both construct a round that does not depend on the ballot data structure.

Thus, when $r = r_2 - 1$, the two executions are the same. Execution γ_j goes on to append an additional round, and so we conclude that γ_i is always a prefix of γ_j, as required. □

We get the following immediate corollary when round r_2 is good:

Corollary 11.7.15. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$ and that $r_2 > r_1$. If round r_2 is good, and $i \in I_B$ is the round r_2 proposer, then for every $j \in I_B$ that completes round r_2, $exec(r_1, r_2, j)_v$ is an extension of $exec(r_1, r_2 - 1, i)$.*

Proof. By Lemma 11.5.4, if round r_2 is good, then $E(v).round\text{-}status[r_2]_j \in \{\mathsf{green}, \mathsf{yellow}\}$, and the claim follows by Lemma 11.7.14. □

We have just examined the case where a node j (locally) designates a round r to be green or yellow, and the case where a round r is good. We now proceed to the case where a node j (locally) designates a round to be orange or red, and the case where a round r is bad. Specifically, we want to show that for every $j \in I_B$ that completes round r_2, if j designates the round to be orange or red, then there exists some $i \in I_B$ where $exec(r_1, r_2, j)_v$ extends $exec(r_1, r_2 - 1, i)_v$. (In the case where j locally designated the round as green or yellow, i was the unique proposer.)

In order to prove this claim, we consider two cases separately: first, the case where j does not join the virtual node in round r_2, i.e., the case where j participates in round r_2 (Lemma 11.7.16); second, the case where j joins v in round r (Lemma 11.7.17). In this second case, we apply our previous analysis of the join protocol (Lemma 11.5.3) to show that the desired conclusion holds.

In the following lemma, we assume that a node j (locally) designates a round as orange or red, and assume that j participates in round r_2 and does not join the emulation of v in round r_2. We then claim that node j extends its own round $r_2 - 1$ execution. That is, $exec(r_1, r_2, j)_v$ extends $exec(r_1, r_2 - 1, j)_v$. This result is intuitively what one might expect: in a bad round, it is not possible for j to extend any other node's execution of v, as communication has been unreliable; thus the only safe possibility is for j to extend j's own execution of v.

Lemma 11.7.16. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$ and that $r_2 > r_1$. Moreover, assume that $j \in I_B$ participates in round r_2 and does not join v in round r_2. If $E(v).round\text{-}status[r_2]_j \in \{\mathsf{red}, \mathsf{orange}\}$ at the end of the last agreement phase, then execution $exec(r_1, r_2, j)_v$ extends execution $exec(r_1, r_2 - 1, j)_v$.*

Proof. Recall that each execution is calculated in two steps: first, determining the round status array (calculate-status), and second, constructing the actual execution (calculate-state). We are comparing two such calculations for node j at the end of round $r_2 - 1$ and the end of round r_2. The main difficulty in the proof is showing that the round status array for the two different calculations is the same for every $r \in [r_1, r_2 - 1]$. In fact, since the round r_2 was (locally) designated by j to be red or orange, we conclude that the $E(v).prev\text{-}rnd_j$ pointer was unmodified in round r_2, and as a result, the status calculation is also identical. As a result, the inputs to the calculate-state calculation are identical (for rounds in the range $[r_1, r_2 - 1]$); for example, for every round $r \in [r_1, r_2 - 1]$, $ballot[r]$ is the same in both cases. Thus we conclude the the calculations proceed identically, leading to the desired result.

Let:

$$s_A = \mathsf{calculate\text{-}status}(E(v).rnd_j, E(v).prev\text{-}rnd_j, E(v).ballot_j, E(v).last\text{-}reset_j)$$

at the end of round $r_2 - 1$, and let:

$$s_B = \textsf{calculate-status}(E(v).rnd_j, E(v).prev\text{-}rnd_j, E(v).ballot_j, E(v).last\text{-}reset_j)$$

at the end of round r_2. Notice that s_A is the status array used in the calculation of $exec(r_1, r_2-1, j)_v$, while s_B is the status array used in the calculation of $exec(r_1, r_2, j)_v$.

Part 1: In the first part of the proof, we show the following: for all $r \in [r_1, r_2-1]$, $s[r]_A = s[r]_B$. As in Lemma 11.7.14, we compare the initial state of the s_A calculation to the state of the s_B calculation after the first iteration of the loop. We claim that in this situation, and after each subsequent iteration of the loop, the local state calculated is equivalent:

1. $r_A = r_B$,
2. $p_A = p_B$,
3. $temp\text{-}last\text{-}reset_A = temp\text{-}last\text{-}reset_B$,
4. for $r \in [r_1, r_2 - 1]$, $temp\text{-}ballots[r]_A = temp\text{-}ballots[r]_B$, and
5. for $r \in [r_1, r_2 - 1]$, $temp\text{-}status[r]_A = temp\text{-}status[r]_B$.

First we compare the initial state of calculation s_A to the state of s_B after the first iteration of the loop:

- Round r counter: Initially, $r_A = temp\text{-}rnd_A + 1 = r_2$ (by Lemma 11.2.4,as usual). Initially, $r_B = temp\text{-}rnd_B + 1 = r_2 + 1$, and hence after the first iteration of calculation s_B, $r_B = r_2$ (by Lemma 11.2.4), as r_j has been decremented once from $r_2 + 1$. Thus $r_A = r_B$.

- Previous round pointer p: We first argue that $E(v).prev\text{-}rnd_j$ is not modified during virtual round r_2. Prior to the last agreement phase, $E(v).prev\text{-}rnd_j$ is not modified, since since $E(v).round\text{-}status[r_2]_j$ is red or orange. After the last agreement phase, $E(v).prev\text{-}rnd_j$ is not modified, since j does not join or reset v in round r_2. Thus, $E(v).prev\text{-}rnd_j$ is unchanged in virtual round r_2. From this we conclude that initially $p_A = p_B$.

  Next, we argue that p_B is not modified during the first iteration of the calculation of s_B. In particular, p_B is only modified if $r_B = p_B$. Yet we have just argued that p_B is not modified in virtual round r_2; at the beginning of virtual round r_2, $p_B < r_2$, by Lemma 11.4.1. Thus, $p_B \neq r_B$, and after the first iteration $p_B = p_A$ initially.

- Last good round *temp-last-reset*: In both calculation, $temp\text{-}last\text{-}reset = r_1 - 1$ both initially and throughout the calculation, as per Corollary 11.4.5.

- Temporary status array *temp-status*: Initially, for all $r \leq r_2$, $temp\text{-}status[r]_A = temp\text{-}status[r]_B = \perp$. In the first iteration, the $status_B$ calculation updates only $temp\text{-}status[r_2]$; thus for all $r < r_2$, $temp\text{-}status[r]_j = \perp$.

- Temporary ballot array *temp-ballot*: During virtual round r_2, for every $r < r_2$, $E(v).ballot[r]_j$ is unmodified: prior to the last agreement phase, this follows immediately; after the last agreement phase, this follows since j does not join or reset v. Thus, for every $r \in [r_1, r_2 - 1]$, $E(v).ballot[r]_j$ is the same at the beginning and end of round r_2, and hence initially, $\textit{temp-ballot}[r]_A = \textit{temp-ballot}[r]_B$. Moreover, $\textit{temp-ballot}[r]$ is never modified during the calculate-status calculation, and hence they remain equal after the first iteration of the calculation of s_B.

Since the initial state of calculation s_A and the state of calculation s_B after the first iteration are equivalent, except for array values outside the range $[r_1, r_2 - 1]$, and since neither calculation accesses an array entry outside of the range $[r_1, r_2 - 1]$, we conclude that the calculations proceed identically in both instances, and as a result, when the calculation terminates, for every $r \in [r_1, r_2 - 1]$, $s[r]_A = s[r]_B$.

Part 2: The proof concludes by examining the calculate-state function. All the input parameters are equal in the two calculations, with the following exceptions: (1) array entries outside the range $[r_1, r_2 - 1]$, and (2) initially, $\textit{temp-state}_A = r_2 - 1$ and $\textit{temp-state}_B = r_2$. Both calculations thus execute the same **for** loop, beginning at round r_1, and proceeding to construct the execution round by round. Since the *ballot* and *status* array entries are the same, the execution constructed is the same, through the end of round $r_2 - 1$. At this point, the calcuation of $\textit{exec}(r_1, r_2 - 1, j)_v$ returns, while the calculation of $\textit{exec}(r_1, r_2, j)_v$ continues, ensuring that the former is a prefix of the latter. □

In Lemma 11.7.16, we assume that node j does not join virtual node v in round r_2. We now consider the more general case where j may join virtual node in round r_2; our only assumption about node j is that it completes round r_2. If node j joins the emulation in round r_2, we can no longer conclude that j extends its own prior execution for v, as j has no prior execution to extend! Instead, we argue that j extends the execution of some other node i, specifically, the node i that sent the join-ack message that allowed j to complete the join protocol.

Lemma 11.7.17. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$ and that $r_2 > r_1$. Moreover, assume that $j \in I_B$ completes round r_2. If $E(v).ballot[r_2]_j \in \{\text{red}, \text{orange}\}$, then there exists some $i \in I_B$ that participates in virtual round r_2 such that execution $\textit{exec}(r_1, r_2, j)_v$ extends execution $\textit{exec}(r_1, r_2 - 1, i)_v$.*

Proof. If j does not join v in round r_2, then the conclusion follows immediately from Lemma 11.7.16: in this case, j participates in round r_2, and hence we choose $i = j$. Thus it remains to consider the case where j joins v in round r_2.

If j joins v in round r_2, then j receives a message from some node $i \in I_B$ during the join-ack phase. It is straightforward to see that i does not join v in round r_2, as i sends information in the join-ack phase; similarly, it is clear that i participates in round r_2. By Lemma 11.7.16, we conclude that $\textit{exec}(r_1, r_2 - 1, i)_v$ at the end of round $r_2 - 1$ is a prefix of $\textit{exec}(r_1, r_2, i)_v$ at the end of round r_2.

After j joins, all the relevant state variables of j are equal to those of i. Thus their calculation of the executions proceed identically, that is, $exec(r_1, r_2, i)_v = exec(r_1, r_j, j)_v$ at the end of round r_2, which implies the desired result. □

We get the following immediate corollary which states that when a round r_2 is bad, then every calculated execution extends some previous execution:

Corollary 11.7.18. *Assume that $\langle r_1, r_2\rangle$ delineates a finite epoch of virtual node $v \in I_V$ and that $r_2 > r_1$. Moreover, assume that $i \in I_B$ completes round r_2. If round r_2 is bad, then there exists some $j \in I_B$ that participates in virtual round r_2 such that execution $exec(r_1, r_2, i)_v$ extends execution $exec(r_1, r_2 - 1, j)_v$.*

Proof. By Lemma 11.5.4, we know that $E(v).round\text{-}status[r_2]_i$ is either red or orange at the end of round r_2, and the claim then follows by Lemma 11.7.17 □

The following lemma, then, is a key lemma which shows that every execution is an extension of a previous execution. Recall that Lemma 11.7.14 addresses the situation where a node $j \in I_B$ designated a round to be green or yellow; Lemma 11.7.17 addresses the situation where a node $j \in I_B$ desiganted a round to be orange or red. Thus the following lemma combines these two results, showing that regardless of the color designation, if j completes round r_2, then its view of virtual node v's $[r_1, r_2]$-execution is a one-round extension of some other node's view of v's $[r_1, r_2 - 1]$-execution. Moreover, we argue that this other node does not fail or depart the emulation prior to the scheduled-ballot phase.

Lemma 11.7.19. *Assume $\langle r_1, r_2\rangle$ delineates a finite epoch of virtual node v, and $r_2 > r_1$. Then for every $j \in I_B$ that completes round r_2, there exists some $i \in I_B$ that begins round r_2 and does not fail or set $E(v).joined_i =$ false prior to the scheduled-ballot phase bcast event, such that $exec(r_1, r_2, j)_v$ is an extension of $exec(r_1, r_2 - 1, i)_v$.*

Proof. There are two cases: either status $E(v).round\text{-}status[r_2]_j \in \{\text{green}, \text{yellow}\}$, or status $E(v).round\text{-}status[r_2]_j \in \{\text{red}, \text{orange}\}$. In the former case, the result follows from Lemma 11.7.14, and the fact that the round r_2 proposer does not fail or set $E(v).joined_i =$ false prior to broadcasting the ballot; in the latter case, the result follows from Lemma 11.7.17. □

Lemma 11.7.19 shows that every execution is a one-round extension of a previous execution. We now apply this argument inductively to prove the same result about multi-round extensions. That is, we show that for every $r \in [r_1, r_2]$, every execution constructed for rounds $[r_1, r_2]$ has as a prefix some execution constructed for rounds $[r_1, r]$.

Lemma 11.7.20. *Assume $\langle r_1, r_2\rangle$ delineates a finite epoch of virtual node v. Let $r \in [r_1, r_2]$ be a virtual round. Then if some $i \in I_B$ completes round r_2, there exists some $j \in I_B$ that completes round r such that $exec(r_1, r_2, i)_v$ is an extension of $exec(r_1, r, j)_v$. Moreover, if $r < r_2$, then j does not fail or set $E(v).joined_j =$ false prior to the scheduled-ballot phase bcast of round $r + 1$.*

Proof. If $r = r_2$, the claim follows immediately, where $j = i$. If $r_2 > r$, the claim follows by backwards induction, decrementing r' from r_2 down to $r + 1$, and at each stage showing that for every node $k \in I_B$ that completes round r', there exists some $k' \in I_B$ that completes round $r' - 1$ such that $exec(r_1, r', k)_v$ is an extension of $exec(r_1, r'-1, k)_v$. At each stage of the induction, this claim follows by Lemma 11.7.19. Thus by transitivity of extension, the claim follows. □

Thus, we can conclude from Lemma 11.7.20 that for every $r \in [r_1, r_2]$, for every $\gamma \in execs(r_1, r_2)_v$ there exists some $\gamma' \in execs(r_1, r)_v$ such that $\gamma' < \gamma$. We can state a somewhat stronger result in the case where some round $r \in [r_1 + 1, r_2]$ is good: we know that if round r is good, then γ' is the execution for $[r_1, r - 1]$ calculated by the (unique) proposer from round r. Lemma 11.7.21 extends Lemma 11.7.20 for this specific case:

Lemma 11.7.21. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch for virtual node $v \in I_V$ and that $r_2 > r_1$. Moreover, assume that virtual round $r \in [r_1 + 1, r_2]$ is good and that node $i \in I_B$ is the round r proposer. (See Definition 11.5.17.) For every $j \in I_B$ that completes round r_2, $exec(r_1, r_2, j)_v$ is an extension of $exec(r_1, r - 1, i)$.*

Proof. We first invoke Lemma 11.7.20 to show that for every $j \in I_B$ that completes round r_2, there exists some $k \in I_B$ that completes round r such that $exec(r_1, r_2, j)_v$ is an extension of $exec(r_1, r, k)_v$. Since k completes round r, we then invoke Corollary 11.7.15 to show that $exec(r_1, r, k)_v$ is an extension of $exec(r_1, r - 1, i)$. The claim then follows by the transitivity of extension. □

11.8 Eventually All Green

In this section, we argue that eventually every round is green for a non-failed virtual node. Recall (from Chapter 8) that the following three components of a basic system guarantee some eventually good behavior: the basic broadcast service guarantees eventual collision freedom; the collision detector guarantees eventual accuracy; and the regional contention managers guarantee eventual ℓ-regional fairness. In this section, we consider what happens when all the eventuality conditions hold. We begin by identifying the earliest round, r_{gst}, in which all the conditions hold:

Definition 11.8.1. *Let r_{gst} be the maximum of the following values:*

- *$r_{acc}(\alpha)$, as hypothesized to exist by Lemma 8.1.42;*
- *r_{cf}, as assumed to exist by Definition 8.1.56;*
- *For every CM_v, $v \in I_V$, round r_{cm} as assumed to exist by Definition 8.1.54.*

The main result of this section is Lemma 11.8.12, which shows that for every round $r \geq r_{gst}$, for every $v \in I_V$, if virtual node v is up in round r, then round r is green for v. This fact implies that the agreement protocol has completed successfully with all the replicas agreeing on the execution for the virtual node. We also show a slightly

stronger result, Lemma 11.8.13, which shows that every nearby node designates a round as green. (Recall that the definition of a round being green requires only that one node designate a round as green.)

The main structure of the argument in this section proceeds as follows. Consider, for the purpose of this overview, the case in which some virtual node v is scheduled for some virtual round r. (The unscheduled case is nearly identical.) We first show that in this case, there is exactly one ballot b broadcast for v in the scheduled-ballot phase. Moreover, every node near to v receives ballot b, and moreover does not detect any collisions. Thus, none of the nodes near to v broadcast in the scheduled-veto-1 phase, and as a result, none of the nodes near to v broadcast in the scheduled-veto-2 phase. As a result, the round is designated green by every node that is near to v at the beginning of round r.

Thus the key argument in the proof is presented in Section 11.8.1 in which we conclude that if a virtual node v is up, then after round r_{gst}: exactly one ballot is broadcast by a replica of v, exactly one ballot is received by replicas of v, and no collisions are detected in the ballot phase. This result depends on the stabilization of the contention managers, the collision detector, and the broadcast service: the contention manager ensures that exactly one ballot is broadcast; the broadcast service and eventual collision freedom ensure that the ballot is received; the eventual accuracy of the collision detector ensures that no spurious collision are detected.

The argument then divides into two (nearly) identical cases, depending on whether the virtual node v is scheduled or unscheduled. We discuss the scheduled agreement instance in Section 11.8.2, and the unscheduled agreement instance in Section 11.8.3. Finally, in Section 11.8.4, we assemble the various cases and prove the main conclusion of this section, Lemma 11.8.12.

11.8.1 Exactly One Proposer per Virtual Round

In this section we show that after r_{gst}, there is exactly one proposer for virtual node v that broadcasts a message during a ballot phase, and hence each node $i \in I_B$ that is near a virtual node receives exactly one ballot—and detects no collisions—in the ballot phase of each virtual round $\geq r_{gst}$.

We begin the section with two lemmas related to the contention manager: Lemma 11.8.2 shows that the emulator satisfies ℓ-restricted contention for each of the regional contention managers, while Lemma 11.8.3 shows that the regional contention managers select exactly one emulator to be active for each virtual round $\geq r_{gst}$. Both of these lemmas involve a careful examination of the properties of a regionally-fair contention manager (Definition 8.1.54). This section is one of two places in this paper where the properties of the regional contention manager are exploited. (The other place is Section 11.16, where we analyze the join protocol to show when the virtual node is failed and when it is not failed.)

Lemma 11.8.3 further shows that there is exactly one proposer for each virtual round $\geq r_{gst}$. This lemma may be reminiscent of Lemma 11.5.14, which shows that in a non-red round there is a unique proposer. In this case, however, we have a different hypothesis: instead of assuming that round r is non-red, we assume that

round $r \geq r_{gst}$; we later use this lemma to conclude that the round is non-red—in fact, that the round is green.

We are then able to draw some conclusions about the messages broadcast and received by replicas of v during ballot rounds. Lemma 11.8.4 show that for every node i that is near a virtual node, there is exactly one message broadcast by a node near to i during a ballot phase. This lemma depends primarily on the *schedule* to show that no two virtual nodes are too close together. Finally, Lemma 11.8.5 concludes that for every node i that is near a virtual node, i does not detect a collision during a ballot phase and i receives exactly one ballot in the ballot phase. This lemma depends on eventual collision freedom to show that the single message broadcast is received, and eventual accuracy to show that no collision is detected.

We now begin with an examination of the regional contention managers. Recall that the guarantees of a regional contention manager depend on the execution α satisfying ℓ-restricted contention for some location ℓ; if α does not satisfy ℓ-restricted contention, then the regional contention managers may not reduce contention. Thus, our first claim in this section is that α in fact satisfies ℓ-restricted contention for all $v \in V$. The proof of this fact is relatively straightforward: a port contends only if it has already joined a virtual node $v \in I_V$, and a joined port is always within distance $R_B/4$ of v.

Lemma 11.8.2. *For every $v \in I_V$, for $\ell = loc(v)_V$, execution α satisfies ℓ-restricted contention for CM_v.*

Proof. The broadcast service has two types of ports: those of the form $\langle i, 0\rangle$, for $i \in I_B$, and those of the form $\langle i, v\rangle$, for $i \in I_B$ and $v \in I_V$. For every $i \in I_B$, port $\langle i, 0\rangle$ never contends for CM_v: this is ensured by the preconditions of the multiplexer bcast action which only allow $cm = \mathsf{global}$ or $\mathsf{cm} = \perp$ (see lines 695–698, Figure 10-13).

Consider port $\langle i, v\rangle$ for $i \in I_B$, $v \in I_V$, and consider some basic round $r_b > 0$. We show that if port $\langle i, v\rangle$ contends in round r_b, then at the beginning of round $r_b - 1$, node i is within distance $R_B/4$ of $loc(v)$. The preconditions of the round $r_b - 1$ $\mathsf{bcast}_{i,v}$ transition ensure that $cm = v'$ only if $v = v'$ and $E(v).joined_i = \mathsf{true}$ (lines 164–167, Figure 10-4). Moreover, $E(v).joined_i = \mathsf{true}$ at the beginning of basic round $r_b - 1$ only if the preceding round $r_b - 2$ $\mathsf{recv}(\cdot, \cdot, \cdot, loc)_{i,v}$ event specified a location loc within distance $R_B/4$ of $loc(v)_V$. Since the round $r_b - 1$ bcast event occurs at the same time as the round $r_b - 2$ recv event, we can conclude that i is within distance $R_B/4$ of $loc(v)_V$ when the $\mathsf{bcast}_{i,v}$ event occurs, as required. □

Notice that the emulator uses the information from the regional contention managers in only two phases: in the client phase, to set the *roundCM* variable (which is used in the vn and scheduled-ballot phases), and in the unscheduled-ballot phase, to determine whether to send a ballot. The first conclusion of Lemma 11.8.3, then, is that after stabilization *at most* one node has $roundCM = \mathsf{active}$ after the client phase. This conclusion follows almost immediately from the fact that α satisfies ℓ-restricted contention. The second conclusion is that, as a result, there is at most one $k \in I_B$ such that port $\langle k, v\rangle$ broadcasts in the scheduled-ballot phase.

The third and fifth conclusions of Lemma 11.8.3 depend on a further assumption about the availability of a node to activate, and conclude that, in fact, there is *exactly one* node that is advised by the contention manager to be active. Recall (from Definition 8.1.54) that a regional contention manager advises a unique node to be active in basic round r_b only when there is some non-failed node that contends for r_b and is nearby and non-failed in rounds r_b and $r_b + 1$. Thus the second conclusion of Lemma 11.8.3 is that when such a node exists, there is exactly one node with $roundCM = \mathsf{active}$. The third conclusion considers the unscheduled-ballot phase, and shows that exactly one node is advised to be active. The fourth and sixth conclusions of Lemma 11.8.3 follow immediately from the third and fifth conclusions, respectively, showing that there is exactly one $k \in I_B$ such that port $\langle k, v\rangle$ broadcasts during the scheduled-ballot phase or unscheduled-ballot phase, respectively.

Lemma 11.8.3. *Let $r \geq r_{gst}$ be a virtual round, and $v \in I_V$ be a virtual node. Then:*

1. *There exists is at most one node $j \in I_B$ that has $E(v).roundCM_j = \mathsf{active}$ at the end of the client phase of round r.*

2. *There exists at most one $j \in I_B$ such that port $\langle j, v\rangle$ broadcasts a message $m \neq \perp$ in the scheduled-ballot phase.*

3. *If there exists some $i \in I_B$ that begins round r and has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the end of the scheduled-ballot phase of round r, then there exists exactly one node $j \in I_B$ that has $E(v).roundCM = \mathsf{active}$ at the end of the vn phase of round r; moreover, j begins round r for v and remains joined and not failed through the beginning of the scheduled-ballot phase, that is, through the $\mathsf{bcast}_{j,v}$ event in the scheduled-ballot phase.*

4. *If there exists some $i \in I_B$ that begins round r and has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the end of the scheduled-ballot phase of round r, and if v is scheduled for round r, then there is exactly one $j \in I_B$ such that port $\langle j, v\rangle$ broadcasts a message $m \neq \perp$ in the scheduled-ballot phase.*

5. *Let $r_a \in [0, \mathsf{SMAX} - 1]$, and assume that v is unscheduled for round r and $v \in schedule[r_a]$. If there exists some $i \in I_B$ that begins virtual round r and has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the end of round $r_a + 2$, then there exists exactly one $j \in I_B$ for which a $\mathsf{recv}(\cdot, \cdot, cm, \cdot)_{j,v}$ event occurs in the basic round r_a in the unscheduled-ballot phase of round r where $cm = \mathsf{active}$. Moreover, j begins round r for v and remains joined and not failed through the end of basic round $r_a + 1$ in the unscheduled-ballot phase.*

6. *Let $r_a \in [0, \mathsf{SMAX} - 1]$, and assume that v is unscheduled for round r and $v \in schedule[r_a]$. If there exists an $i \in I_B$ that begins virtual round r and has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the end of round $r_a + 2$, and if v is not scheduled for round r, then there is exactly one $j \in I_B$ such that port $\langle j, v\rangle$ broadcasts a message $m \neq \perp$ in round $r_a + 1$ of the unscheduled-ballot phase of r.*

Proof. The main argument of the proof proceeds as follows: Conclusion 1 follows from the fact that CM_v advises at most one port $\langle j, v\rangle$, for $j \in I_B$, to be active according to Definition 8.1.54, Part 1. It remains to show that other contention managers do not advise some other port $\langle k, v\rangle$, $k \neq j$, to be active, thus violating the claim that port $\langle j, v\rangle$ is the only port that may be advised to be active. To show this claim, we rely on the fact that all the contention managers are conservative: every port $\langle k, v\rangle$ either contends for contention manager CM_v, or does not contend for any contention manager, which leads to the desired result. Conclusion 2 is an immediate corollary of Conclusion 1.

Conclusions 3 and 5 follow from Definition 8.1.54, Part 2: since a node exists satisfying the hypothesis of Definition 8.1.54, Part 2, we conclude that there is in fact some $j \in I_B$ such that the contention manager CM_v advises $\langle j, v\rangle$ to be active. Combined with the results from Conclusion 1, this implies that the port $\langle j, v\rangle$ is the unique port advised by CM_v to be active. Conclusion 4 and 6 are immediate corollaries of Conclusion 3 and 5, respectively. We now proceed in more detail.

Conclusion 1. Definition 8.1.54, Part 1, states that if α satisfies ℓ-restricted contention for $\ell = loc(v)$, then there exists some round r_{cm} such that for every round $r \geq r_{cm}$, at most one port is advised by CM_v to be active in round r. We conclude from Lemma 11.8.2 that α satisfies ℓ-restricted contention, for $\ell = loc(v)_V$, and by assumption, $r_{gst} \geq r_{cm}$ posited by the definition. We thus conclude that in the client phase, there is at most one node $j \in I_B$ that the contention manager CM_v advises to be active in the vn phase.

It remains to show that no other contention manager advises a port in the client phase recv to be active during the vn phase. Consider some node $k \in I_B, k \neq j$. We can conclude that port $\langle k, v\rangle$ either contends for CM_V, or does not contend to be active in the vn phase, as per lines 164–167 in Figure 10-4. Since port $\langle k, v\rangle$ contends only for contention manager CM_v, and, since all the contention managers are conservative, we can conclude that for all $k \in I_B$, $k \neq j$, every contention manager advises port $\langle k, v\rangle$ to be passive in the vn phase of round r, resulting in $E(v).roundCM_k = \mathsf{passive}$.

We thus conclude that at the end of the client phase, there is at most one node $j \in I_B$ such that $E(v).roundCM_j = \mathsf{active}$.

Conclusion 2. A port $\langle k, v\rangle$, where $k \in I_B$, broadcasts a message $m \neq \perp$ in the scheduled-ballot phase only if $E(v).roundCM_k = \mathsf{active}$ at the end of the vn phase, and thus Conclusion 2 follows as an immediate result of Conclusion 1, along with the fact that *roundCM* is unmodified in the vn phase.

Conclusions 3 and 5. Choose basic round r_b to be either (1) the basic round associated with the client phase of round r, or (2) the basic round associated with round r_a of the unscheduled-ballot phase, depending on whether we are interested in arguing Conclusion 3 or 5, respectively. As per the lemma hypothesis, we assume that there exists an $i \in I_B$ that begins virtual round r and has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ at the end of round $r_b + 2$ (i.e., the scheduled-ballot phase of round r, in the case of Conclusion 3, or round $r_a + 2$, in the case of Conclusion 5).

It is immediately clear that port $\langle i, v\rangle \in \mathsf{near}(loc(v)_V, [r_b{+}1, r{+}b{+}2], bcast\text{-}ports_B)$, since if node i were farther than $R_B/4$ from $loc(v)_V$ at the beginning of either round

$r_b + 1$ or $r_b + 2$, then during the recv transition for round r_b or round $r_b + 1$ (which occur simultaneous with the beginning of the following rounds), $E(v).joined_i \leftarrow \mathsf{false}$. By assumption, we know that this is not the case.

It is also immediately clear that port $\langle i, v\rangle$ contends for CM_v in round r_b+1, since whenever $E(v).joined_i = \mathsf{true}$ at the beginning of basic round r_b, port $\langle i, v\rangle$ contends in round $r_b + 1$, according to lines 164–167, Figure 10-4.

We can therefore determine that node i satisfies the hypothesis of Definition 8.1.54, Part 2. Thus, there exists a port $p \in bcast\text{-}ports_B$ such that:

a. Port p contends for CM_v in round $r_b + 1$.

b. Port p does not fail prior to the end of round $r_b + 2$.

c. Port $p \in \mathsf{near}(loc(v)_V, [r_b + 1, r_b + 2], bcast\text{-}ports_B)$.

d. Contention manager CM_v advises port p to be active in round r_b.

Choose $j \in I_B$ such that $p = \langle j, v\rangle$. (We know that for all $j \in I_B$, $p \neq \langle j, 0\rangle$, as port $\langle j, 0\rangle$ never contends for the vn phase of a virtual round.) Thus we conclude that at the end of the client phase, $E(v).roundCM_j = \mathsf{active}$, in the case of Conclusion 3, and that there is a $\mathsf{recv}(\cdot, \cdot, \mathsf{active}, \cdot)_{j,v}$ event in round r_a, in the case of Conclusion 5. Combined with Conclusion 1, this yields the desired results.

Conclusion 4. Since v is scheduled, a port $\langle k, v\rangle$ broadcasts a message $m \neq \perp$ in the scheduled-ballot phase under the following conditions: (1) line 273: $E(v).scheduled_k = \mathsf{true}$; (2) line 274: $E(v).joined_k = \mathsf{true}$; (3) line 274: $E(v).roundCM_k = \mathsf{active}$. Conclusion 3 guarantees that there is exactly one such node.

Conclusion 6. Since v is unscheduled, a port $\langle k, v\rangle$ broadcasts a message $m \neq \perp$ in round $r_a + 1$ of the unscheduled-ballot phase under the following conditions: (1) line 378: $E(v).scheduled_j = \mathsf{false}$; (2) line 378: $E(v).joined_j = \mathsf{true}$; (3) line 388: $v \in schedule[r_a]$; (4) line 390: $cm = \mathsf{active}$. Conclusion 5 guarantees that there is exactly one such node. □

The following lemma draws further conclusions about the behavior of the emulator for v during the scheduled-ballot and unscheduled-ballot phase. Lemma 11.8.3 shows that only one port $\langle \cdot, v\rangle$ for virtual node v broadcasts a message. That is, it shows that among ports for a single virtual node, the regional contention manager is successful in reducing the contention. It remains to show that contention among ports for *different* virtual nodes is also sufficiently low.

Lemma 11.8.4, Conclusion 1, shows that in each of the ballot phases, for every node $i \in I_B$ that is near a virtual node, there is *at most* one broadcast by a node that may interfere with i, or none if i itself broadcasts a message. If some node is available, that is, joined and not failed, then Lemma 11.8.4, Conclusions 2 and 3, show that there is exactly one such broadcast in the appropriate ballot phase (i.e., the scheduled-ballot phase, if v is scheduled, and the unscheduled-ballot phase, otherwise). The proof depends in large part on Lemma 11.8.3 and the fact that the schedule is non-interfering.

Lemma 11.8.4. *Let $r \geq r_{gst}$ be a virtual round, let $v \in I_V$ be a virtual node, and let $i \in I_B$ be a node.*

1. *If i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the* scheduled-ballot *phase, then there exists at most one port $p \in$ bcast-ports$_B$ such that a* bcast$(m, \cdot)_p$ *event, $m \neq \perp$, occurs in the* scheduled-ballot *phase of round r where port p is within distance R'_B of i at the beginning of the phase.*

2. *If i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the* scheduled-ballot *phase, and if v is scheduled for round r, and if there exists some $k \in I_B$ that begins round r and remains joined and not failed through the end of the* scheduled-ballot *phase, then there exists exactly one port $p \in$ bcast-ports$_B$ such that a* bcast$(m, \cdot)_p$ *event, $m \neq \perp$, occurs in the* scheduled-ballot *phase of round r where port p is within distance R'_B of i at the beginning of the phase. Additionally, port p is within distance R_B of i at the beginning of the* scheduled-ballot *phase.*

3. *Choose $r_a \in [0, \mathsf{SMAX} - 1]$. If i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of round $r_a + 1$ of the* unscheduled-ballot *phase of virtual round r, and if v is unscheduled for round r, and if $v \in schedule[r_a]$, and if there exists some $j \in I_B$ that begins round r and remains joined and not failed through the end of basic round $r_a + 2$ in the* unscheduled-ballot *phase of round r, then there exists exactly one port $p \in$ bcast-ports$_B$ such that a* bcast$(m, \cdot)_p$ *event, $m \neq \perp$, occurs in round r_a of the* unscheduled-ballot *phase of round r where port p is within distance R'_B of i at the beginning of the round. Additionally, port p is within distance R_B of i at the beginning of round r_a in the* unscheduled-ballot *phase.*

Proof. For the purpose of this proof, we partition the broadcast ports into three sets:

1. $BP_1 = \{\langle j, v\rangle : j \in I_B\}$,
2. $BP_2 = \{\langle j, v'\rangle : j \in I_B, v' \in I_V, v' \neq v\}$,
3. $BP_3 = \{\langle j, 0\rangle : j \in I_B\}$.

Notice that these three sets are non-intersecting, and together contain all the ports in *bcast-ports*$_B$. (Recall that $\langle j, 0\rangle$ is a broadcast port for node j.)

We consider each of the three conclusions separately (though the arguments are similar). In each case, we consider which of the ports broadcast in the specified phase. We can conclude from Lemma 11.8.3 that there is at most one port—or exactly one port, depending on the case—in the set BP_1 that broadcasts in the ballot phase. We can conclude from the fact that the schedule is non-interfering that there is at most one port in the set BP_2 that broadcasts in the ballot phase—and that ports in this set broadcast only when no ports in the BP_1 set broadcast in the ballot phase. Finally, we show that no port in BP_3 ever broadcasts in a ballot phase. Thus, in each of the three cases, we achieve the desired conclusion.

Conclusion 1: Assume that i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the scheduled-ballot phase:

1. BP_1: Lemma 11.8.3, Part 2, shows that there is at most one $j \in I_B$ such that port $\langle j, v\rangle$ broadcasts a message $m \neq \perp$ in the scheduled-ballot phase. If v is unscheduled for round r, then there is no $j \in I_B$ such that port $\langle j, v\rangle$ broadcasts a message $m \neq \perp$ in the scheduled-ballot phase, since the $E(v).scheduled_j$ flag prevents a message $m \neq \perp$ from being broadcast in the scheduled-ballot phase by an emulator for an unscheduled virtual node (line 273).

2. BP_2: There are two subcases to consider, depending on whether v is scheduled or unscheduled.

   Assume that v is scheduled for round r. Consider some port $\langle k, v'\rangle \in BP_2$ where k is within distance R'_B of i and $v' \neq v$. Port $\langle k, v'\rangle$ broadcasts a message $\neq \perp$ in the scheduled-ballot phase only if k has $E(v').joined_k = \mathsf{true}$, and hence only if k is within distance $R_B/4$ of $loc(v')_V$. Thus, v' is within distance $R'_B + R_B/4 + 3R_B/4 < 2R'_V$ of v. (The first term R'_B refers to the distance of k from i; the second term refers to the distance of k from v'; the third term refers to the distance of i from v.) Since $v' \neq v$, and node v is scheduled, we can conclude by the non-interference guarantee of the *schedule* that v' is not scheduled for round r, and hence port $\langle k, v'\rangle$ does not broadcast a message $\neq \perp$ in the scheduled-ballot phase. From this argument we conclude that if v is scheduled for round r, no port in BP_2 that is within distance R'_B of i at the beginning of the scheduled-ballot phase broadcasts a message $\neq \perp$ in the scheduled-ballot phase.

   Assume that v is not scheduled for round r. We show that there is at most one port in the interference range of i that broadcasts a message $\neq \perp$ in the scheduled-ballot phase. Assume for the sake of contradiction that there are two distinct ports $\langle j', v'\rangle$ and $\langle j'', v''\rangle$, where $v', v'' \in I_V$ and $j', j'' \in I_B$, and both j and j' are both within distance R'_B of i at the beginning of the scheduled-ballot phase. Port $\langle j', v'\rangle$ broadcasts $m \neq \perp$ in the scheduled-ballot phase only if $E(v').joined_{j'} = \mathsf{true}$, and hence only j' is within distance $R_B/4$ of $loc(v')_V$; it also broadcasts in the scheduled-ballot phase only if v' is scheduled for round r. Similarly, port $\langle j'', v''\rangle$ broadcasts $m \neq \perp$ in the scheduled-ballot phase only if j'' is within distance $R_B/4$ of $loc(v'')_V$ and v'' is scheduled for round r. Since v' and v'' are both scheduled, either $v' = v''$ or the two ports are at least distance $2R'_V$ apart, due to the non-interference of the schedule.

   If $v' = v''$, then we conclude by Lemma 11.8.3, Part 1, that $j' = j''$, as there is at most one $j \in I_B$ such that port $\langle j, v'\rangle$ that broadcasts a message $m \neq \perp$ in the scheduled-ballot phase of round r. This contradicts our assumption that ports $\langle j', v'\rangle$ and $\langle j'', v''\rangle$ are distinct.

   Otherwise, if $v' \neq v''$, then v' and v'' are distance $> 2R'_V = R_B + 2R'_B$ apart. Since j' is within distance $R_B/4$ of $loc(v')_V$ and j'' is within distance $R_B/4$ of $loc(v'')_V$, we can conclude that j' and j'' are at distance $> R_B/2 + 2R'_B > 2R'_B$ apart. Thus it is impossible that i be within distance R'_B of both of them, contradicting our assumption.

From this argument, we see that if v is not scheduled, then there is at most one port $\langle j, v' \rangle \in BP_2$ within distance R'_B of i that broadcasts a message $\neq \perp$ in the scheduled-ballot phase.

3. BP_3: For all $k \in I_B$, port $\langle k, 0 \rangle$ broadcasts only during the client phase, and never during the scheduled-ballot phase.

From these three facts, we conclude that there exists at most one port $p \in \textit{bcast-ports}_B$ such that a $\mathsf{bcast}(m, \cdot)_p$ event, $m \neq \perp$, occurs in the scheduled-ballot phase of round r where port p is within distance R'_B of i at the beginning of the phase: specifically, if v is scheduled, $p \in BP_1$ (if it exists); if v is unscheduled, $p \in BP_2$ (if it exists).
Conclusion 2: Assume that i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the scheduled-ballot phase, and that v is scheduled for round r, and that there exists some $k \in I_B$ that begins round r and remains joined and not failed through the end of the scheduled-ballot phase:

1. BP_1: Lemma 11.8.3, Part 4, shows that there is exactly one $j \in I_B$ such that port $\langle j, v \rangle$ broadcasts a message $m \neq \perp$ in the scheduled-ballot phase. We can also conclude that node j is within distance $R_B/4$ of $loc(v)_V$ at the beginning of the scheduled-ballot phase, as otherwise $E(v).joined_j \leftarrow \mathsf{false}$, at which point port $\langle j, v \rangle$ does not broadcast a message $\neq \perp$.

2. BP_2: Since v is scheduled, the argument here is identical to the scheduled subcase of Conclusion 1. We repeat the argument here: Consider some port $\langle k, v' \rangle \in BP_2$ where k is within distance R'_B of i and $v' \neq v$. Port $\langle k, v' \rangle$ broadcasts a message $\neq \perp$ in the scheduled-ballot phase only if k has $E(v').joined_k = \mathsf{true}$, and hence only if k is within distance $R_B/4$ of $loc(v')_V$. Thus, v' is within distance $R'_B + R_B/4 + 3R_B/4 < 2R'_V$ of v. (The first term R'_B refers to the distance of k from i; the second term refers to the distance of k from v'; the third term refers to the distance of i from v.) Since $v' \neq v$, and node v is scheduled, we can conclude by the non-interference guarantee of the *schedule* that v' is not scheduled for round r, and hence port $\langle k, v' \rangle$ does not broadcast a message $\neq \perp$ in the scheduled-ballot phase. From this argument we conclude that no port in BP_2 that is within distance R'_B of i at the beginning of the scheduled-ballot phase broadcasts a message $\neq \perp$ in the scheduled-ballot phase.

3. BP_3: For all $k \in I_B$, port $\langle k, 0 \rangle$ broadcasts only during the client phase, and never during the scheduled-ballot phase.

From these three facts, we conclude that here exists exactly one port $p \in \textit{bcast-ports}_B$ such that a $\mathsf{bcast}(m, \cdot)_p$ event, $m \neq \perp$, occurs in the scheduled-ballot phase of round r where port p is within distance R'_B of i at the beginning of the phase. Additionally, port $p \in BP_1$ and is within distance $R_B/4$ of v at the beginning of the scheduled-ballot phase; since i is within distance $3R_B/4$ of v, we can conclude that port p is within distance R_B of i.
Conclusion 3: Assume i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of round $r_a + 1$ of the unscheduled-ballot phase of virtual round r, and that v is unscheduled

for round r, and that $v \in \mathit{schedule}[r_a]$, and that there exists some $i \in I_B$ that begins round r and remains joined and not failed through the beginning of basic round r_a+2 in the unscheduled-ballot phase of round r:

1. Since v is unscheduled and $v \in \mathit{schedule}[r_a]$, then Lemma 11.8.3, Part 6, shows that there exists exactly one $j \in I_B$ such that port $\langle j, v\rangle$ broadcasts a message $m \neq \perp$ in round $r_a + 1$ of the unscheduled-ballot phase. Also, as argued in the previous cases, node j is within distance $R_B/4$ of $\mathit{loc}(v)_V$ at the beginning of round r_a of the scheduled-ballot phase, as otherwise port $\langle j, v\rangle$ would not have broadcast a message in the ballot phase. Thus we conclude that node j is within distance R_B of node i.

2. Since v is unscheduled, the argument here is not entirely identical to—but essentially the same as—the scheduled subcase of Conclusion 1. We repeat the argument here: Consider some port $\langle k, v'\rangle \in BP_2$ where k is within distance R'_B of i at the beginning of round r_a in the unscheduled-ballot phase and $v' \neq v$. Port $\langle k, v'\rangle$ broadcasts a message $\neq \perp$ in the unscheduled-ballot phase only if k has $E(v').\mathit{joined}_k = \mathsf{true}$, and hence only if k is within distance $R_B/4$ of $\mathit{loc}(v')_V$ at the beginning of round r_a in the unscheduled-ballot phase. Thus, v' is within distance $R'_B + R_B/4 + 3R_B/4 < 2R'_V$ of v. (The first term R'_B refers to the distance of k from i; the second term refers to the distance of k from v'; the third term refers to the distance of i from v.) Since $v' \neq v$, and $v \in \mathit{schedule}[r_a]$, we can conclude by the non-interference guarantee of the *schedule* that $v' \notin \mathit{schedule}[r_a]$, and hence port $\langle k, v'\rangle$ does not broadcast a message $\neq \perp$ in round r_a+1 of the unscheduled-ballot phase. From this argument we conclude that no port in BP_2 that is within distance R'_B of i at the beginning of round $r_a + 1$ in the unscheduled-ballot phase broadcasts a message $\neq \perp$ in round $r_a + 1$ of the unscheduled-ballot phase.

3. BP_3: For all $k \in I_B$, port $\langle k, 0\rangle$ broadcasts only during the client phase, and never during the unscheduled-ballot phase.

Putting these three conclusions together, we can conclude that there exists exactly one port $p \in \mathit{bcast\text{-}ports}_B$ such that a $\mathsf{bcast}(m, \cdot)_p$ event, $m \neq \perp$, occurs in round r_a+1 of the unscheduled-ballot phase of round r where port p is within distance R'_B of i at the beginning of the basic round. Additionally, port $p \in BP_1$, and thus is within distance R_B of i at the beginning of round $r_a + 1$ in the unscheduled-ballot phase. □

Lemma 11.8.4 specifies which messages are broadcast in a ballot phase. We now proceed in Lemma 11.8.5 to examine which messages are received in a ballot phase. Since only one message is broadcast by a nearby node in the ballot phase (as per Lemma 11.8.4), we invoke eventual collision freedom to show that the message is received; we invoke eventual accuracy to conclude that no collisions are detected. Using Lemma 11.8.5 we will be able to easily conclude that after r_{gst}, the various ballot phases proceed "cleanly," which is a prerequisite for a green round.

The conclusions of Lemma 11.8.5 divide into two (nearly identical) claims, the first with respect to the scheduled-ballot phase, the second with respect to the unscheduled-ballot phase.

Lemma 11.8.5. *Let $r \geq r_{gst}$ be a virtual round, and let $v, v' \in I_V$ be virtual nodes. Let $i \in I_B$ be a node.*

1. *Assume that a $\mathsf{recv}(allM, cd, \ldots)_{i,v}$ event occurs in the* scheduled-ballot *phase of round r in α, and that virtual node v' is scheduled. Assume that node i is within distance $3R_B/4$ of $loc(v')_V$. Then $cd = \mathsf{null}$ and $|\{\langle \mathsf{vn}, v', \cdot \rangle \in allM\}| \leq 1$.*

   If there exists some $j \in I_B$ that begins round r for v' and remains joined and not failed through the end of the scheduled-ballot *phase, then $|\{\langle \mathsf{vn}, v', \cdot \rangle \in allM\}| = 1$.*

2. *Let $r_a \in [0, \mathsf{SMAX} - 1]$, and assume that a $\mathsf{recv}(allM, cd, \ldots)_{i,v}$ event occurs in round $r_a + 1$ of the* unscheduled-ballot *phase of round r in α. Assume that virtual node v' is unscheduled, and that $v' \in schedule[r_a]$. Assume that node i is within distance $3R_B/4$ of $loc(v')_V$. If there exists some $j \in I_B$ that begins round r for v' and remains joined and not failed through the beginning of round $r_a + 2$, then $cd = \mathsf{null}$ and $|\{\langle \mathsf{vn}, v', \cdot \rangle \in allM\}| = 1$.*

Proof. We consider the two parts separately, first examining the scheduled-ballot phase, and then examining the unscheduled-ballot phase. In each case, the argument proceeds roughly according to the following three steps: (1) According to Lemma 11.8.4, there is at most one nearby port p that broadcasts a message m in the ballot phase; (2) Since the basic broadcast service guarantees eventual collision freedom and $r \geq r_{gst}$, port $\langle i, v \rangle$ receives message m, if it is broadcast by port p; (3) Since every message broadcast by a nearby or interfering node is received, eventual accuracy guarantees that $cd = \mathsf{null}$. We now proceed in more detail.

Part 1: First, we consider the scheduled-ballot phase. By Lemma 11.8.4, Part 1, there exists at most one port $p \in bcast\text{-}ports_B$ such that a $\mathsf{bcast}(m, \cdot)_p$ event, $m \neq \perp$, occurs in the scheduled-ballot phase of round r where port p is within distance R'_B of i at the beginning of the phase. By Lemma 11.8.4, Part 2, if there exists some $j \in I_B$ that begins round r for v' and remains joined and not failed through the end of the scheduled-ballot phase, then there is exactly one such port p, and in addition port p is within distance R_B of i at the beginning of the phase.

In the case where some port p broadcasts a message, we now consider the eventual collision freedom property of the basic broadcast service (Definition 8.1.56). Notice that all the required conditions hold: (1) port p broadcasts a message, (2) port p is within distance R_B of port $\langle i, v \rangle$ at the beginning of the scheduled-ballot phase, and (3) no port $k \neq p$ within distance R'_B of $\langle i, v \rangle$ at the beginning of the scheduled-ballot phase broadcasts a message. Thus, since $r \geq r_{gst}$, we know that the basic round associated with the scheduled-ballot phase $\geq r_{cf}$, and thus we conclude that $m \in allM$, the set of messages received by port $\langle i, v \rangle$ in the scheduled-ballot phase.

Next, we argue that $cd = \mathsf{null}$ because of the eventual accuracy of the basic broadcast service. In particular, Definition 8.1.41 and Lemma 8.1.42 state that if

there is a $\mathsf{recv}(allM, \pm, \ldots)_{i,v}$ event in the scheduled-ballot phase, then there exists a port $p' \in bcast\text{-}ports_B$ and $m \in msgs_B$ such that:

1. There exists a $\mathsf{bcast}(m, \ldots)_{p'}$ event in the scheduled-ballot phase.
2. Port p' is within distance R'_B of $\langle i, v\rangle$ at the beginning of the scheduled-ballot phase.
3. Message $m \notin allM$.

If no port p broadcast a message, then we conclude immediately that $cd = \mathsf{null}$. Otherwise there exists exactly one port $p \in bcast\text{-}ports_B$, where p is within distance R'_B of $\langle i, v\rangle$ at the beginning of the phase, that broadcasts a message $m \neq \perp$. And we have already concluded that $m \in allM$. Thus no such port p' can exist, and hence we can conclude that $cd = \mathsf{null}$.

Finally, we conclude that $|\{\langle \mathsf{vn}, v', \cdot\rangle \in allM\}| = 1$: only one message is sent by a port within distance R'_B (i.e., port p), and no port of distance greater than R'_B sends a message of the form $\langle \mathsf{vn}, v', \cdot\rangle$. In particular, a port sends a message of the form $\langle \mathsf{vn}, v', \cdot\rangle$ only if it is within distance $R_B/4$ of $loc(v')_V$, and hence only if it is within distance R_B of i. (Notice that there may be messages of the form $\langle \mathsf{vn}, v'', \cdot\rangle$ in $allM$, where $v'' \neq v'$; the broadcast service may deliver messages sent from ports that are quite distant. These messages are discarded by the emulator.)

Part 2: The second part is quite similar to the first part, though we omit the claim that *at most* one node broadcasts, instead focusing on the case where *exactly* one node broadcasts a message. Consider round r_a of the unscheduled-ballot phase. By Lemma 11.8.4, Part 3, there exists exactly one port $p \in bcast\text{-}ports_B$ such that a $\mathsf{bcast}(m, \cdot)_p$ event, $m \neq \perp$, occurs in basic round $r_a + 1$ in the unscheduled-ballot phase of round r where port p is within distance R'_B of i at the beginning of the basic round, and in addition port p is within distance R_B of i at the beginning of the basic round.

We now consider the eventual collision freedom property of the basic broadcast service (Definition 8.1.56). Notice that all the required conditions hold: (1) port p broadcasts a message, (2) port p is within distance R_B of port $\langle i, v\rangle$ at the beginning of round $r_a + 1$ in the unscheduled-ballot phase, and (3) no port $k \neq p$ within distance R'_B of $\langle i, v\rangle$ at the beginning of round $r_a + 1$ in the unscheduled-ballot phase broadcasts a message. Thus, since $r \geq r_{gst}$, we know that the basic round associated with round $r_a + 1$ of the unscheduled-ballot phase is $\geq r_{cf}$, and thus we conclude that $m \in allM$, the set of messages received by port $\langle i, v\rangle$ in round $r_a + 1$ in the unscheduled-ballot phase.

Next, we argue that $cd = \mathsf{null}$ because of the eventual accuracy of the basic broadcast service. In particular, Definition 8.1.41 and Lemma 8.1.42 state that if there is a $\mathsf{recv}(allM, \pm, \ldots)_{i,v}$ event in round $r_a + 1$ of the unscheduled-ballot phase, then there exists a port $p' \in bcast\text{-}ports_B$ and $m \in msgs_B$ such that:

1. There exists a $\mathsf{bcast}(m, \ldots)_j$ event in the scheduled-ballot phase.
2. Port p' is within distance R'_B of $\langle i, v\rangle$ at the beginning of round $r_a + 1$ of the unscheduled-ballot phase.

3. Message $m \notin allM$.

However, we have already concluded that there exists exactly one port $p \in bcast\text{-}ports_B$, where p is within distance R'_B of $\langle i, v\rangle$ at the beginning of the basic round, that broadcasts a message $m \neq \perp$. And we have already concluded that $m \in allM$. Thus no such port j can exist, and hence we can conclude that $cd = \mathsf{null}$.

Finally, we argue that $|\{\langle \mathsf{vn}, v', \cdot\rangle \in allM\}| = 1$, since only one such message is sent by a port within distance R'_B (i.e., port p), and no port of distance greater than R'_B sends a message of the form $\langle \mathsf{vn}, v', \cdot\rangle$. In particular, a port sends a message of the form $\langle \mathsf{vn}, v', \cdot\rangle$ only if it is within distance $R_B/4$ of $loc(v')_V$, and hence only if it is within distance R_B of i. □

In the next two sections, we apply Lemma 11.8.5 to show that the scheduled and unscheduled ballot rounds proceed cleanly, which results in the conclusion that for every virtual node v that is up, for every virtual round $r \geq r_{gst}$, round r is a green round for v.

11.8.2 Scheduled Virtual Nodes

In this section, we analyze the scheduled agreement instance, and show that after stabilization, i.e., for every $r \geq r_{gst}$, every node $j \in I_B$ that has not failed at the end of the agreement instance designates virtual round r as green.

We examine each phase of the scheduled agreement instance, and show that throughout, the variable $E(v).scheduled\text{-}status$ remains set to the default value of $\perp$. Thus, at the end of the scheduled-veto-2 phase, when the scheduled agreement instance is complete, each participating node designates the round as green. Lemma 11.8.6 analyzes the scheduled-ballot phase; Lemma 11.8.7 analyzes the scheduled-veto-1 phase; and Lemma 11.8.8 analyzes the scheduled-veto-2 phase, stating the final result. Each lemma examines the possible ways in which $E(v).scheduled\text{-}status$ can be modified, and shows that none of them occur.

First, we show that for all $i \in I_B$, for all $v \in I_V$, at the end of the scheduled-ballot phase for some virtual round $r \geq r_{gst}$, $E(v).scheduled\text{-}status_i = \perp$. In addition, we show that $E(v).outgoing\text{-}msg_i = \perp$. Together, these facts indicate that the ballot round was "successful," thus preventing vetoes in the following two rounds. Notice that the lemma makes no assumptions on whether v is up or down, or whether node i is near to virtual node v. It also makes no assumptions as to whether v is scheduled or unscheduled for round r.

Lemma 11.8.6. *Let $v \in I_V$ be a virtual node, $r \geq r_{gst}$ a virtual round, and $i \in I_B$ be a node that does not fail prior to the end of the* scheduled-ballot *phase of round r. Then at the end of the* scheduled-ballot *phase, $E(v).scheduled\text{-}status_i = \perp$ and $E(v).outgoing\text{-}msg_i = \perp$.*

Proof. Consider the $\mathsf{recv}(allM, cd, \ldots)_{i,v}$ event in the scheduled-ballot phase of round r (which takes place since i is non-failed). Initially (line 303), $E(v).scheduled\text{-}status_i \leftarrow \perp$. In this proof, we examine the various lines in which $E(v).scheduled\text{-}status_i$ is

modified, and argue that Lemma 11.8.5 implies that $E(v).\mathit{scheduled\text{-}status}_i$ remains $\perp$ throughout. The conclusion for $E(v).\mathit{outgoing\text{-}msg}_i$ follows immediately, since $E(v).\mathit{outgoing\text{-}msg}_i$ is modified only if $E(v).\mathit{scheduled\text{-}status}_i \neq \perp$.

First, consider the case where i is farther than $3R_B/4$ from $loc(v)_V$ when the recv event occurs. Then $E(v).\mathit{scheduled\text{-}status}_i$ is not modified from $\perp$ throughout the recv transition for the scheduled-ballot phase (see the **if** condition on line 304).

Second, consider the case where there exists no $v' \in I_V$ such that v' is scheduled and i is within distance $3R_B/4$ of $loc(v)_V$. Then again, $E(v).\mathit{scheduled\text{-}status}_i$ is not modified from $\perp$ throughout the recv transition (see the **else if** condition on line 306).

Thus, assume that i is within distance $3R_B/4$ of $loc(v')_V$, for some $v' \in I_V$ where v' is scheduled for round r. By Lemma 11.8.5, Part 1, we conclude that $cd = \mathsf{null}$ and that if $M' = \{\langle \mathsf{vn}, v', \cdot\rangle \in \mathit{allM}\}$, then $|M'| \leq 1$. If $v' = v$ and $E(v).\mathit{joined}_i = \mathsf{true}$, then we assert that there exists some $j \in I_B$ that begins round r for v' and remains joined and not failed through the end of the scheduled-ballot phase, that is, node i itself. In this case, again by lemrefexactlyone3, Part 1, we conclude that $|M'| = 1$.

We now consider each of the lines during the recv transition in the scheduled-ballot phase where $E(v).\mathit{scheduled\text{-}status}_i$ is modified, in the order the lines are executed during the transition:

- line 303: In this line, $E(v).\mathit{scheduled\text{-}status}_i$ is initialized to $\perp$, as desired. We will argue that it is not changed during the rest of the recv transition.
- line 313: This line is only executed if $cd = \pm$, which we have already precluded above (by Lemma 11.8.5), or if $|M'| > 1$, which we have already precluded above (by Lemma 11.8.5).
- line 319: This line is only executed if $v = v'$, $E(v).\mathit{joined}_i = \mathsf{true}$, and $\not\exists \langle \mathsf{vn}, v, b\rangle \in \mathit{allM}$, i.e., $|M'| = 0$, which we have already precluded above (by Lemma 11.8.5).

Thus, when the recv transition completes, $E(v).\mathit{scheduled\text{-}status}_i = \perp$, which immediately implies that $E(v).\mathit{outgoing\text{-}msg}_= \perp$ (line 321). □

Next, we show that at the end of the scheduled-veto-1 phase, $\mathit{scheduled\text{-}status} = \perp$. In addition, we show that $\mathit{outgoing\text{-}msg} = \perp$. Together, these facts indicate that the first veto round was successful, that is, entirely silent.

Lemma 11.8.7. *Let $v \in I_V$ be a virtual node, $r \geq r_{gst}$ a virtual round, and $i \in I_B$ a node that does not fail prior to the end of the* scheduled-veto-1 *phase of round r. Then at the end of the* scheduled-veto-1 *phase, $E(v).\mathit{scheduled\text{-}status}_i = \perp$ and $E(v).\mathit{outgoing\text{-}msg}_i = \perp$.*

Proof. By Lemma 11.8.6, we conclude two facts:

- $E(v).\mathit{scheduled\text{-}status}_i = \perp$ at the beginning of the scheduled-veto-1 phase.
- For every $v' \in I_V$, for every $j \in I_B$ that has $E(v').\mathit{failed}_j = \mathsf{false}$ at the beginning of the scheduled-veto-1 phase, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$.

There are two situations during the scheduled-veto-1 phase of round r that can result in $E(v).\mathit{scheduled\text{-}status}_i$ being set $\neq \perp$: port $\langle i, v\rangle$ receives a message of the form $\langle \mathsf{vn}, v, \mathsf{veto}\rangle$, or port $\langle i, v\rangle$ detects a collision. That is, $E(v).\mathit{scheduled\text{-}status}_i$ is modified if during the $\mathsf{recv}(\mathit{allM}, \mathit{cd}, \ldots)_{i,v}$ event for the scheduled-veto-1 phase of round r, either $\mathit{allM} \neq \emptyset$ or $\mathit{cd} = \pm$.

Since for every non-failed $j \in I_B$, for every $v' \in I_V$, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$, we can conclude (by the integrity of the basic broadcast service, Lemma 8.1.35), that i does not receive a veto message in the scheduled-veto-1 phase.

We can also conclude by the eventual accuracy of the basic broadcast service (Lemma 8.1.42) that since no non-$\perp$ messages are broadcast in the scheduled-veto-1 phase, no collisions are detected by port $\langle i, v\rangle$.

Thus, at the end of the scheduled-veto-1 phase, $E(v).\mathit{scheduled\text{-}status}_i = \perp$. As a result, it follows immediately that $E(v).\mathit{outgoing\text{-}msg}_i = \perp$ (see line 341). □

Finally, we show that at the end of the scheduled-veto-2 phase, *scheduled-status* $= \perp$, and draw two further conclusions as a result. Consider some virtual node $v \in I_V$ and some node $i \in I_B$. The first conclusion of Lemma 11.8.8 deals with unscheduled virtual nodes: $E(v).\mathit{scheduled\text{-}status}_i = \perp$ at the end of the scheduled-veto-2 phase. This conclusion is useful when arguing about the ballot proposed in the unscheduled agreement instance, which depends on which messages were broadcast (or not broadcast) by v in the scheduled agreement instance. The second conclusion deals with scheduled virtual nodes and all nodes $i \in I_B$, and states that the round status is either $\perp$ or green. The third conclusion also deals with scheduled virtual nodes, but limits its attention to nodes $i \in I_B$ that are "near" to v. In this case, the round status is green. Thus, if any such node exists, we can conclude that round r is green for v.

Lemma 11.8.8. *Let $v \in I_V$ be a virtual node, $r \geq r_{gst}$ a virtual round, and $i \in I_B$ a node that does not fail prior to the end of the* scheduled-veto-2 *phase of round r.*

1. *If v is unscheduled in round r, then $E(v).\mathit{scheduled\text{-}status}_i = \perp$ at the end of the* scheduled-veto-2 *phase.*

2. *If v is scheduled in round r, then at the end of the* scheduled-veto-2 *phase, $E(v).\mathit{round\text{-}status}[r]_i \in \{\mathsf{green}, \perp\}$.*

3. *If v is scheduled in round r and i is within distance $3R_B/4$ of $\mathit{loc}(v)_V$ at the beginning of the* scheduled-veto-2 *phase, then at the end of the* scheduled-veto-2 *phase, $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$.*

Proof. The proof of this lemma is quite similar to that of Lemma 11.8.7. First, we apply Lemma 11.8.7 to conclude the following two facts:

- $E(v).\mathit{scheduled\text{-}status}_i = \perp$ at the beginning of the scheduled-veto-2 phase.
- For every $v' \in I_V$, for every $j \in I_B$ that has $E(v').\mathit{failed}_j = \mathsf{false}$ at the beginning of the scheduled-veto-2 phase, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$.

There are two situations during the scheduled-veto-2 phase of round r that can result in $E(v).\mathit{scheduled\text{-}status}_i$ being set $\neq \perp$: port $\langle i, v\rangle$ receives a message of the form $\langle \mathsf{vn}, v, \mathsf{veto}\rangle$, or port $\langle i, v\rangle$ detects a collision. That is, $E(v).\mathit{scheduled\text{-}status}_i$ is modified if during the $\mathsf{recv}(\mathit{allM}, \mathit{cd}, \ldots)_{i,v}$ event for the scheduled-veto-2 phase of round r, either $\mathit{allM} \neq \emptyset$ or $\mathit{cd} = \pm$.

Since for every non-failed $j \in I_B$, for every $v' \in I_V$, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$, we can conclude (by the integrity of the basic broadcast service, Lemma 8.1.35), that i does not receive a veto message in the scheduled-veto-2 phase.

We can also conclude by the eventual accuracy of the basic broadcast service (Lemma 8.1.42) that since no non-$\perp$ messages are broadcast in the scheduled-veto-2 phase, no collisions are detected by port $\langle i, v\rangle$.

We conclude, then, that at the end of the scheduled-veto-2 phase, $E(v).\mathit{scheduled\text{-}status}_i = \perp$. This immediately implies all three conclusions: if v is unscheduled for round r, then $E(v).\mathit{scheduled\text{-}status}_i = \perp$; if v is scheduled, then $E(v).\mathit{round\text{-}status}[\mathit{rnd}]_i$ gets either green or $E(v).\mathit{scheduled\text{-}status}_i$ (i.e., $\perp$); if v is scheduled *and* within distance $R_B \leq 3r_B/4$ of $\mathit{loc}(v)_V$, then we conclude that $E(v).\mathit{round\text{-}status}[r]_i$ is set to green, as desired, which concludes the proof. □

11.8.3 Unscheduled Virtual Nodes

In this section, we analyze the unscheduled agreement instance, and show that after stabilization, i.e., for every $r \geq r_{gst}$, every node $j \in I_B$ that has not failed at the end of the agreement instance designates virtual round r as green. The proof in this section is nearly identical to that in Section 11.8.2, with the exception that we are interested only in virtual nodes that are unscheduled.

We examine each phase of the unscheduled agreement instance, and show that throughout, the status $E(v).\mathit{round\text{-}status}[r]$ remains set to the default value of $\perp$. Thus, at the end of the unscheduled-veto-2 phase, each participating node designates the round as green. Lemma 11.8.9 analyzes the unscheduled-ballot phase; Lemma 11.8.10 analyzes the unscheduled-veto-1 phase; and Lemma 11.8.11 analyzes the unscheduled-veto-2 phase, stating the final result. Each lemma examines the possible ways in which $E(v).\mathit{round\text{-}status}[r]$ can be modified, and shows that none of them occur.

First, we show that for all $i \in I_B$, for all $v \in I_V$, at the end of the unscheduled-ballot phase for some virtual round $r \geq r_{gst}$, $E(v).\mathit{round\text{-}status}[r]_i = \perp$. In addition, we show that $E(v).\mathit{outgoing\text{-}msg}_i = \perp$. Together, these facts indicate that the ballot round was "successful," thus preventing vetoes in the following two phases. Notice that the lemma makes no assumptions on whether v is up or down, or whether node i is near to virtual node v. It also makes no assumptions as to whether v is scheduled or unscheduled for round r.

Lemma 11.8.9. *Let $v \in I_V$ be a virtual node, $r \geq r_{gst}$ a virtual round, and $i \in I_B$ be a node that does not fail prior to the end of the* unscheduled-ballot *phase of round r. Then at the end of the* unscheduled-ballot *phase, $E(v).\mathit{round\text{-}status}[r]_i = \perp$ and $E(v).\mathit{outgoing\text{-}msg}_i = \perp$.*

Proof. Initially, at the beginning of the unscheduled-ballot phase, $E(v).\mathit{round\text{-}status}[r]_i = \perp$. Our goal is to show that it remains as such through the end of the unscheduled-ballot phase. In this proof we examine the various lines in which $E(v).\mathit{round\text{-}status}[r]_i$ is modified, and argue that Lemma 11.8.5 implies that $E(v).\mathit{round\text{-}status}[r]_i$ remains $\perp$ throughout. The conclusion for $E(v).\mathit{outgoing\text{-}msg}_i$ follows immediately, since $E(v).\mathit{outgoing\text{-}msg}_i$ is modified only if $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{red}$ (line 393).

Notice that for every basic round $r_a{+}1$ in which $v \notin \mathit{schedule}[r_a]$, $E(v).\mathit{round\text{-}status}[r]_i$ is not modified. Thus we consider basic round $r_a{+}1$ such that $v \in \mathit{schedule}[r_a]$. (There is exactly one such round by the completeness of the schedule.)

Consider the $\mathsf{recv}(\mathit{allM}, \mathit{cd}, \ldots)_{i,v}$ event in round $r_a + 1$ of the unscheduled-ballot phase of round r (which takes place since i is non-failed).

First, consider the case where i is farther than $R_B/4$ from $\mathit{loc}(v)_V$ at the beginning of round $r_a + 1$ in the unscheduled-ballot phase. Then $E(v).\mathit{joined}_i = \mathsf{false}$, and as a result, $E(v).\mathit{round\text{-}status}[r]_i$ is not modified from $\perp$ throughout the recv transition (see the **if** condition on line 378).

Thus, for the rest of the proof, we assume that i is within distance $R_B/4$ of $\mathit{loc}(v)_V$ at the beginning of round $r_a + 1$ in the unscheduled-ballot phase, and that $E(v).\mathit{joined}_i = \mathsf{true}$ at the beginning of the basic round. By Lemma 11.8.5, Part 2, we conclude that $\mathit{cd} = \mathsf{null}$ and that if $M' = \{\langle \mathsf{vn}, v, b\rangle \in \mathit{allM}\}$, then $|M'| = 1$. (Notice that node i itself satisfies the hypothesis for Lemma 11.8.5.) We now consider each of the lines during the recv transition when $E(v).\mathit{round\text{-}status}[r]_i$ is modified, in the order the lines are executed during the transition:

- line 382: This line is only executed if $|M'| = 0$ of $|M'| > 1$, which we have already precluded.
- line 384: This line is only executed if $\mathit{cd} = \pm$, which we have already precluded.

Thus, at the end of the unscheduled-ballot phase, $E(v).\mathit{round\text{-}status}[r]_i$ remains unchanged from its initial value $\perp$. This immediately implies that $E(v).\mathit{outgoing\text{-}msg}_i = \perp$ (line 321). □

Next, we show that at the end of the unscheduled-veto-1 phase, $\mathit{round\text{-}status}[r] = \perp$. In addition, we show that $\mathit{outgoing\text{-}msg} = \perp$. Together, these facts indicate that the first veto round was successful, that is, entirely silent.

Lemma 11.8.10. *Let $v \in I_V$ be a virtual node, $r \geq r_{gst}$ a virtual round, and $i \in I_B$ a node that does not fail through the end of the* unscheduled-veto-1 *phase of round r. Then $E(v).\mathit{round\text{-}status}[r]_i = \perp$ and $E(v).\mathit{outgoing\text{-}msg}_i = \perp$ at the end of the* unscheduled-veto-1 *phase.*

Proof. The proof of this lemma is quite similar to that of Lemma 11.8.7. By Lemma 11.8.9, we conclude two facts:

- $E(v).\mathit{round\text{-}status}[r]_i = \perp$ at the beginning of the unscheduled-veto-1 phase.
- For every $v' \in I_V$, for every $j \in I_B$ that has $E(v').\mathit{failed}_j = \mathsf{false}$ at the beginning of the unscheduled-veto-1 phase, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$.

There are two situations during the unscheduled-veto-1 phase of round r that can result in $E(v).\mathit{round\text{-}status}[r]_i$ being set $\neq \perp$: port $\langle i, v\rangle$ receives a message of the form $\langle \mathsf{vn}, v, \mathsf{veto}\rangle$, or port $\langle i, v\rangle$ detects a collision. That is, $E(v).\mathit{round\text{-}status}[r]_i$ is modified if during the $\mathsf{recv}(\mathit{allM}, \mathit{cd}, \ldots)_{i,v}$ event for the unscheduled-veto-1 phase of round r, either $\mathit{allM} \neq \emptyset$ or $\mathit{cd} = \pm$.

Since for every non-failed $j \in I_B$, for every $v' \in I_V$, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$, we can conclude (by the integrity of the basic broadcast service, Lemma 8.1.35), that i does not receive a veto message in the unscheduled-veto-1 phase.

We can also conclude by the eventual accuracy of the basic broadcast service (Lemma 8.1.42) that since no non-$\perp$ messages are broadcast in the unscheduled-veto-1 phase, no collisions are detected by port $\langle i, v\rangle$.

Thus, at the end of the unscheduled-veto-1 phase, $E(v).\mathit{round\text{-}status}[r]_i = \perp$. As a result, it follows immediately that $E(v).\mathit{outgoing\text{-}msg}_i = \perp$ (see line 410). □

Finally, we show that at the end of the unscheduled-veto-2 phase, $\mathit{round\text{-}status}[r] =$ green. This lemma implies that in the case where v is unscheduled, round r is green.

Lemma 11.8.11. *Let $v \in I_V$ be a virtual node, $r \geq r_{gst}$ a virtual round, and $i \in I_B$ a node that does not fail through the end of the last agreement phase of round r.*

1. *If v is unscheduled for round r, then at the end of the last agreement phase, $E(v).\mathit{round\text{-}status}[r]_i \in \{\mathsf{green}, \perp\}$.*
2. *If v is unscheduled in round r, and if i is within distance $3R_B/4$ of $\mathit{loc}(v)_V$ at the beginning of the last agreement phase, then at the end of the last agreement phase, $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$.*

Proof. The proof of this lemma is quite similar to that of Lemma 11.8.8 and Lemma 11.8.10. First, we apply Lemma 11.8.10 to conclude the following two facts:

- $E(v).\mathit{round\text{-}status}[r]_i = \perp$ at the beginning of the unscheduled-veto-2 phase.
- For every $v' \in I_V$, for every $j \in I_B$ that has $E(v').\mathit{failed}_j = \mathsf{false}$ at the beginning of the unscheduled-veto-2 phase, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$.

There are two situations during the unscheduled-veto-2 phase of round r that can result in $E(v).\mathit{round\text{-}status}_i$ being set $\neq \perp$ prior to line 426: port $\langle i, v\rangle$ receives a message of the form $\langle \mathsf{vn}, v, \mathsf{veto}\rangle$, or port $\langle i, v\rangle$ detects a collision. That is, $E(v).\mathit{round\text{-}status}[r]_i$ is modified if during the $\mathsf{recv}(\mathit{allM}, \mathit{cd}, \ldots)_{i,v}$ event for the unscheduled-veto-2 phase of round r, either $\mathit{allM} \neq \emptyset$ or $\mathit{cd} = \pm$.

Since for every non-failed $j \in I_B$, for every $v' \in I_V$, $E(v').\mathit{outgoing\text{-}msg}_j = \perp$, we can conclude (by the integrity of the basic broadcast service, Lemma 8.1.35), that i does not receive a veto message in the unscheduled-veto-2 phase.

We can also conclude by the eventual accuracy of the basic broadcast service (Lemma 8.1.42) that since no non-$\perp$ messages are broadcast in the unscheduled-veto-2 phase, no collisions are detected by port $\langle i, v\rangle$.

Thus, on line 426, if node i is within distance $R_V < 3R_B/4$ of $\mathit{loc}(v)_V$, then $E(v).\mathit{round\text{-}status}[r]_i$ is assigned to green in line 426. Otherwise, $E(v).\mathit{round\text{-}status}[r]_i$ remains $\perp$ through the end of the last agreement phase. □

11.8.4 Eventually, All Rounds are Green

We conclude with the main result of this section, which shows that eventually all rounds are green:

Lemma 11.8.12. *For every virtual node $v \in I_V$, for every virtual round $r \geq r_{gst}$, if v is up in virtual round r, then round r is green for v.*

Proof. Since v is up in round r, by Lemma 11.3.10, there is some node $i \in I_B$ that begins round r for virtual node v and does not fail prior to the end of the unscheduled-veto-2 phase. Moreover, we can conclude that at the end of each basic round in round r, up until and including the unscheduled-veto-2 phase, node i is within distance $R_B/4$ of $loc(v)_V$ since $E(v).joined_i = \mathsf{true}$ throughout.

If v is scheduled, we apply Lemma 11.8.8 to conclude that $E(v).\textit{round-status}[r]_i = \mathsf{green}$ at the end of the last agreement phase. If v is not scheduled, we apply Lemma 11.8.11 to conclude that $E(v).\textit{round-status}[r]_i = \mathsf{green}$ at the end of the last agreement phase. Thus, by Definition 11.5.1, round r is green. □

We provide one secondary result: if a virtual node v is up in round r, and if $j \in I_B$ completes round r for v, then j designates round r as green. This result is somewhat stronger in that its conclusion refers to the designation of each node $j \in I_B$ that completes round r, rather than only one $j \in I_B$. Also, it consider the designation at the end of the virtual round, rather than at the end of the last agreement phase, and hence we must examine the join protocol, once again.

Lemma 11.8.13. *For every virtual node $v \in I_V$, for every virtual round $r \geq r_{gst}$, if v is up in virtual round r and if $j \in I_B$ completes round r for v, then $E(v).\textit{round-status}[r]_j = \mathsf{green}$ at the end of round r.*

Proof. Since j completes round r for v, and since v was not reset in round r (as v is up in round r), we can conclude that either i participates in round r, or i joins virtual node v in round r. In the former case, Lemma 11.8.8 and Lemma 11.8.11 imply the desired result. In the latter case, i must have received the round status from some other node $k \in I_B$ which itself participates in round r, again implying the desired result. We now proceed in more detail.

- Assume some node $k \in I_B$ participates in round r. We can conclude that at the end of each basic round in round r, up until and including the last agreement phase, node k is within distance $R_B/4$ of $loc(v)_V$; otherwise k would not remain joined.

  If v is scheduled, we apply Lemma 11.8.8 to conclude that $E(v).\textit{round-status}[r]_k = \mathsf{green}$ at the end of the last agreement phase. If v is not scheduled, we apply Lemma 11.8.11 to conclude that $E(v).\textit{round-status}[r]_k = \mathsf{green}$ at the end of the last agreement phase.

  From this we conclude that if j participates in round r and does not join v in round r, then at the end round r, $E(v).\textit{round-status}[r]_j = \mathsf{green}$: a node only changes *round-status* after the last agreement phase if it joins or resets the virtual node, and we have assumed that this does not happen.

- Assume, then, that j joins v in round r. Then in the join-ack phase, j receives a copy of the *round-status* array. This must have been sent by some node $k \in I_B$ (by the integrity of the basic broadcast service), and we can conclude from the previous argument that at the end of the last agreement phase, $E(v).\mathit{round\text{-}status}[r]_k = \mathsf{green}$. Thus, after receiving the join-ack message, round-status[r]j = green, and is not changed for the rest of the virtual round, completing our proof.

□

11.9 Defining an Execution of a Virtual Node

In this section, for each virtual node $v \in I_V$, we construct an execution γ_v. Later, we paste together the γ_v executions, along with the client executions, the contention manager executions, and the broadcast service execution to produce a single execution γ of the virtual infrastructure system.

We now present a brief description of the mechanism by which γ_v is constructed. A more careful description of the construction is presented in Figures 11-1 and 11-2. (During the construction, whenever two actions are inserted consecutively, assume that an intervening point trajectory is inserted into the execution, as required to ensure that the resulting sequence is an execution.)

The construction consists first of an initialization portion (Figure 11-1) which constructs all the virtual rounds prior to the first epoch in the updown sequence; it is followed (Figure 11-2) by a loop that iterates through epochs in the updown sequences. The variable r is a loop counter; the epoch under consideration is $\langle u_r, d_r \rangle$. In each iteration of the loop, the construction adds rounds $[u_r, u_{r+1}-1]$ to the execution: first, an execution for $\langle u_r, d_r \rangle$, and then a trajectory through the end of round $u_{r+1} - 1$. At the beginning of each loop, the following conditions hold: virtual node v is not failed (i.e., there are no preceding fail events without an intervening recover event), and $\ell\mathit{time}(\gamma_v)$ is the end of round $u_r - 1$. The main loop is broken into two major cases: either the epoch is finite, or the epoch is infinite.

Finite Epoch, Finite d_r: The main idea is to choose some node $j \in I_B$ and append the execution $\mathit{exec}(u_r, d_r, j)_v$ to γ_v. We know, due to the definition of an updown sequence, that v is down in round $d_r + 1$, and hence we need to end this execution with a fail event.

There are two subcases, however, that depend on whether any emulator for virtual node v broadcasts a message for v in round $d_r + 1$. We branch based on whether a message $\langle \mathsf{vn}, v, \cdot \rangle$ was broadcast in the vn and scheduled-ballot phases of round d_r. If so, then some emulator for virtual node v broadcast a message for v; other clients and virtual nodes may well have chosen to receive this message, and hence it is included in the execution for v prior to v failing. In this case, we choose j to be a node that broadcast this message in round $d_r + 1$. (If there is more than one, we choose any of them; in any case, if there is more than one such node, we can be sure that every

other emulator and client either received both or detected a collision.) We append $exec(u_r, d_r, j)_v$ to γ_v, extending the execution through the end of round d_r. We also call do-bcast to generate the round d_r+1 broadcast event, and append the appropriate execution fragment.

Otherwise, if no node broadcasts a $\langle \mathsf{vn}, v, \cdot \rangle$ message in the vn and scheduled-ballot phases of round $d_r + 1$, it is safe to continue constructing the execution under the premise that v failed prior to its round $d_r + 1$ bcast event. In this case, we simply choose an arbitrary execution fragment from $execs(u_r, d_r)_v$, which we know is know empty by Lemma 11.7.8; we append this execution fragment to γ_v.

In both subcases, we append a fail event, and then a time-passage trajectory. There are again two subcases depending on whether the virtual node ever recovers. If there exists some $\langle u_{r+1}, d_{r+1} \rangle$ in the updown sequence, then the time passage of the trajectory continues until some time ϵ after the end of round $u_{r+1} - 1$. The additional ϵ ensures that node v does not receive a round $u_{r+1} - 1$ recv event. Finally, we append a recover event, and a trajectory to complete the round.

If $\langle u_r, d_r \rangle$ is the last element in the updown sequence, a special case occurs: instead of adding a finite trajectory and a recover event, we simply append an infinite trajectory and terminate the construction of γ_v.

Infinite d_r: The second major case occurs when d_r is infinite. In this case, we need to construct an infinite number of virtual rounds. We break this up into two parts: first, we append an execution fragment from round u_r up until one round after the system stabilizes, i.e., $r_{gst} + 1$; then, we append one round at a time, inductively.

For the first part of this step, we choose r' to be the smallest virtual round such that every basic round in r' comes after r_{gst}. We know by Lemma 11.8.12 that after r_{gst}, every round is green, and hence good, by Corollary 11.5.7. Thus, there is only one execution in $execs(u_r, r')_v$ by Corollary 11.7.12. We append this execution to γ_v.

For the second step, we assume that we have constructed an execution up until round r', and show to extend it by one (virtual) round, thus incrementing r'. As before, we know that there is only one execution in $execs(u_r, r'+1)_v$, and moreover, we know that this execution extends the (sole) execution in $execs(u_r, r')_v$. Thus we simply add the difference between these two executions to γ_v, increment r', and continue. The end result is an infinite execution in which any fragment of γ_v representing rounds $[u_r, r']$ is equal to the (sole) execution in $execs(u_r, r')_v$.

11.10 Defining Other Component Executions

In this section, we define the other component executions for the virtual infrastructure system, specifically, the client executions and the contention manager executions. After showing that the executions constructed to this point are sufficient for showing the integrity of the virtual broadcast service (Section 11.11), we proceed in Section 11.12 we construct the (virtual) collision detection rule, and then in Section 11.13, we construct the virtual broadcast execution.

Figure 11-1: Initialization of γ_v.

Initialization:

- Execution γ_v begins with a point trajectory τ_0.
- Assume that $updown(v) = \langle u_0, d_0 \rangle, \langle u_1, d_1 \rangle, \langle u_2, d_2 \rangle, \ldots$.
- If $u_0 \neq 1$, then:
  - Append a fail_v event to γ_v.
  - Append a trajectory τ to γ_v, where $dom(\tau) = (u_i - 1) \cdot RndLength_V + \epsilon$, for some $\epsilon : 0 < \epsilon < RndLength_V$. (Recall that the state of a process is constant over all trajectories, by assumption; see Definition 8.1.2.)
  - Append a $\mathsf{recover}_v$ event to γ_v.
  - Append a trajectory τ with domain $dom(\tau) = RndLength_V - \epsilon$.
- Let $r = 0$.

Figure 11-2: Construction of execution γ_v.

Loop: Repeat until $\langle u_r, d_r \rangle$ is not in $updown(v)$:

- *Case 1. Finite Epoch:* If $d_r \neq \infty$, we consider virtual round d_r:
  - Construct an execution based on whether v broadcasts in round $d_r + 1$:
    - *Subcase A.* If some port $\langle j, v \rangle \in bcast\text{-}ports_B$ broadcasts $\langle \mathsf{vn}, v, m \rangle$, $m \neq \perp$, in the vn phase of virtual round $d_r + 1$ and also $\langle \mathsf{vn}, v, \cdot \rangle$ in the scheduled-ballot phase of virtual round $d_r + 1$ then:
      - Fix such a j; append $exec(u_r, d_r, j)_v$ to γ_v; let $s = \ell state(\gamma_v)$.
      - Let $\langle \cdot, \cdot, \cdot, e \rangle \leftarrow \mathsf{do\text{-}bcast}(s)_v$.
      - Append execution $e.\mathsf{fail}_v$ to γ_v.
    - *Subcase B.* Otherwise:
      - Append an arbitrary execution $e \in execs(u_r, d_r)_v$ to γ_v. (By Lemma 11.7.8, $execs(u_r, d_r)_v \neq \emptyset$.)
      - Append a fail_v event to the end of γ_v.
  - Add time passage until the next epoch begins, if there is one:
    - *Subcase A.* If $updown(v)$ contains $\langle u_{r+1}, d_{r+1} \rangle$:
      - Append trajectory τ where $dom(\tau) = (u_{r+1} - d_r - 1) \cdot RndLength_V + \epsilon$, for some $\epsilon : 0 < \epsilon < RndLength_V$. (Recall: the state of a process is constant over all trajectories; see Definition 8.1.2.)
      - Append a $\mathsf{recover}_v$ event to γ_v.
      - Append a trajectory τ with domain $dom(\tau) = RndLength_V - \epsilon$.
    - *Subcase B.* Otherwise, if $updown(v)$ does not contain any $\langle u_{r+1}, d_{r+1} \rangle$:
      - Append a trajectory τ where $dom(\tau) = \infty$ and all the variables of virtual node v are constant.
- *Case 2. Infinite Epoch:* Otherwise, if d_r is infinite:
  - There are an infinite number of virtual rounds $\geq u_r$. Since α is infinite, Lemma 8.1.31 implies that there are an infinite number of basic rounds in α. Let r' be the smallest virtual round $\geq u_r$ such that every basic round in virtual round r' is $\geq r_{gst} + 1$.
  - First, we construct the execution from u_r to r', and then inductively construct each round of the execution after that one at a time. Choose e to be an arbitrary execution in $execs(u_r, r')_v$. Append e to γ_v. (Notice that $|execs(u_r, r')_v| = 1$ by Corollary 11.7.12, since r' is a green round.)
  - Next, we construct each additional virtual round of execution γ_v inductively. Given an execution through virtual round r' where every basic round in round r' is $> r_{gst}$:
    - Let γ' be the (sole) execution in $execs(u_r, r' + 1)_v$. Notice that it extends $execs(u_r, r')_v$, since round r' is green for v.
    - Append to γ_v all events and trajectories in γ' not in $execs(u_r, r')_v$.
    - Increment round r'.
- $r \leftarrow r + 1$

11.10.1 Client Executions

We next define an execution γ_i, for each $i \in I_B$ as the restriction of α to the events and trajectories relating to the client automaton associated with node i. More formally:

Definition 11.10.1. *Let* $C = \mathsf{Remap}(A_V(i), i)$. *Let* $\boldsymbol{\gamma_i}$ *be* $\alpha|\langle C.actions, C.vars\rangle$.

(See Appendix A for the formal definition of an (A, V)-restriction of an execution.) It follows immediately that each γ_i is an execution of the remapped process $\mathsf{Remap}(A_V(i), i)$.

We now verify that the trace of each γ_i is "consistent" with an execution of the virtual infrastructure system. It is immediately clear that γ_i is an execution of a client automaton; it requires some further examination to show that the execution has the appropriate (virtual) round structure. If the client's execution, as just defined, is a component execution of a virtual infrastructure system, it must have the appropriate round structure: specifically, if a client does not fail in round r, then γ_i contains a round r bcast and a round r recv event. The first round in the execution is a special case: there is a recv event, but no bcast event.

Lemma 11.10.2. *If node i does not fail in virtual round 1, then γ_i contains a single round 1* $\mathsf{recv}_{i,*}$ *event.*

For every virtual round $r > 1$, if node i does not fail prior to beginning round r, then γ_i contains a single round r $\mathsf{bcast}_{i,*}$ *event; if node i does not fail in round r, then γ_i contains a single round r* $\mathsf{recv}_{i,*}$ *event at the end of round r.*

Proof. We show that for each round $r > 0$, if i does not fail in r, then γ_i contains a single round r $\mathsf{recv}_{i,*}$ event. The conclusion with respect to bcast events then follows immediately from the fact that the client is a process and γ_i is an execution of the client process: a process responds with bcast events immediately and only in response to recv events (see Definition 8.1.2).

Consider some virtual round r, and let r_a be the last basic round in r. We need to show that during virtual round r, exactly one $\mathsf{recv}_{i,*}$ event occurs, and that it occurs at the end of round r_a.

We argue that the restriction on trajectories (line 721) ensures that at least one $\mathsf{recv}_{i,*}$ occurs immediately after the round r_a recv event. Specifically, the *stops when* condition restricts the passage of time when $E(multiplexer).phase_i = \mathsf{out}$ and $E(multiplexer).failed_i = \mathsf{false}$. Notice that immediately after the round r_a $\mathsf{recv}_{i,0}$ event, $E(multiplexer).phase_i = \mathsf{out}$ and $E(multiplexer).failed_i = \mathsf{false}$ (as i does not fail prior to the end of r). The following sequence of dependencies show that a $\mathsf{recv}_{i,*}$ event must occur in order for time to pass:

- Time stops at the end of round r when $E(multiplexer).phase_i = \mathsf{out}$ and $E(multiplexer).failed_i = \mathsf{false}$.
- The only event which resets the *phase* to in is a round $r_a + 1$ $\mathsf{bcast}_{i,0}$ event. The precondition of the $\mathsf{bcast}_{i,0}$ event requires $E(multiplexer).clientBcast_i = \mathsf{true}$. After the $\mathsf{recv}_{i,0}$ event, the client broadcast flag $E(multiplexer).clientBcast_i = \mathsf{false}$.

- The only event which sets *clientBcast* to true is a $\mathsf{bcast}_{i,*}$ event.
- Since the client is a process (see Definition 8.1.2), a bcast event occurs only in response to a recv event. Thus the necessary $\mathsf{bcast}_{i,*}$ event occurs only if a $\mathsf{recv}_{i,*}$ event precedes it.

Since the process $A_B(i)$ (consisting of the composed emulator and client) is progressive, as per Lemma 10.3.6, we can conclude that time-passage is enabled after a finite sequence of events after the $\mathsf{recv}_{i,0}$ event, and hence at least one round r $\mathsf{recv}_{i,*}$ event occurs at the end of round r.

Finally, we argue that only one $\mathsf{recv}_{i,*}$ event occurs. Notice that a precondition of the $\mathsf{recv}_{i,*}$ action in Figure 10-13 is that $E(multiplexer).inVNs_i = V$, and an effect of the $\mathsf{recv}_{i,*}$ action is that the set of virtual nodes $E(multiplexer).inVNs_i \leftarrow \emptyset$. Moreover, a virtual node $v \in I_V$ is added to the set $E(multiplexer).inVNs_i$ only during a $\mathsf{vn\text{-}client\text{-}output}_v$ event.

The virtual node emulator $E(v)$, however, ensures that (1) $\mathsf{vn\text{-}client\text{-}output}_v$ events occur only at the end of a virtual round, and (2) only one $\mathsf{vn\text{-}client\text{-}output}_v$ event occurs per virtual round. This follows by noticing that the vn-client-output action has a precondition $E(v).do\text{-}client\text{-}recv_i = \mathsf{true}$, which is reset by the vn-client-output to false. Moreover, this flag is only set during the last basic round in r, i.e., during the basic round r_a recv event. Thus it is only set once per virtual round at the very end of the virtual round.

We therefore conclude that only one $\mathsf{recv}_{i,*}$ event can occur in each virtual round, concluding our proof. □

The next lemma we prove is that the clients' interaction with the virtual broadcast service satisfies the self delivery property. That is, we show that if some client port $\langle i, * \rangle$ broadcasts a message m in round r, then port $\langle i, * \rangle$ also receives message m in round r.

Lemma 11.10.3. *Let $i \in I_B$ be a client and $r > 0$ be a virtual round. If for some $m \in msgs_V$, a round r $\mathsf{bcast}(m, \cdot)_{i,*}$ occurs in γ_i and if a round r $\mathsf{recv}(M, \ldots)_{i,*}$ event occurs in γ_i, then $m \in M$.*

Proof. This lemma follows from the self-delivery property of the basic broadcast service. Immediately after the $\mathsf{bcast}(m, \cdot)_{i,*}$ event, we can determine that $E(multiplexer).outM_i = m$, and hence there is a $\mathsf{bcast}(\langle \mathsf{client}, m, \cdot \rangle, \cdot)_{i,0}$ event in the client phase of round r. Since there is a $\mathsf{recv}_{i,*}$ event at the end of virtual round r (by assumption), we can conclude that i does not fail in the client phase, and hence there is a client phase $\mathsf{recv}(M, \ldots)_{i,*}$ event. By the self-delivery property of the basic broadcast service, we know that $\langle \mathsf{client}, m, \cdot \rangle \in M$, and hence immediately after the $\mathsf{recv}_{i,*}$ event, $\langle \mathsf{client}, m, \cdot \rangle \in inM$. Thus, when the $\mathsf{recv}_{i,*}$ occurs at the end of virtual round r, we can conclude that $m \in M$, as desired. □

11.10.2 The Virtual Contention Manager Execution

We next construct an execution $\gamma_{virtual}$ of the virtual contention manager $CM_{virtual}$. The construction derives entirely from the γ_i, $i \in I_B$, and γ_v, $v \in I_V$, executions

that have already been constructed: every bcast event in these executions is added to $\gamma_{virtual}$; for each bcast event, an immediately following cm-advice event is added.

Recall that each contention manager in the virtual infrastructure system gives advice to every node in the system, real or virtual. However, the virtual contention manager here is designed to provide useful advice for the virtual nodes; thus, it always advises the clients to be passive.

For the virtual nodes, the contention manager derives its advice from the recv events in γ_v, $v \in I_V$. This construction thus ensures that the $\gamma_{virtual}$ execution is consistent with the virtual node executions. As in prior constructions, we implicitly insert point trajectories between actions in the sequence.

The construction itself is presented in Figure 11-3. We then prove that $\gamma_{virtual}$ is an execution of CanonicalCM (Lemma 11.10.4), that $\gamma_{virtual}$ is non-interfering (Lemma 11.10.6), and that $\gamma_{virtual}$ is eventually fair (Lemma 11.10.7).

Lemma 11.10.4. *$\gamma_{virtual}$ is an execution of* CanonicalCM*.*

Proof. Notice that $\gamma_{virtual}$ is an alternating sequence of trajectories and actions, and that all the in $\gamma_{virtual}$ are consistent with CanonicalCM. We need only show the following two facts: (1) the preconditions for transitions in CanonicalCM are satisfied whenever a cm-advice occurs in $\gamma_{virtual}$; (2) time-passage is enabled whenever τ_i is a trajectory with domain > 0.

First, consider some cm-advice$(\langle i, * \rangle, \cdot)$ event, for $i \in I_B \cup I_V$. We need to show that $\langle i, * \rangle \in \mathit{waiting}$. From the construction, we can determine that γ_i contains an immediately preceding $\mathsf{bcast}_{i,*}$ event, which adds $\langle i, * \rangle$ to the *waiting* set, as required.

Second, consider some trajectory τ with domain > 0. Notice that τ that was added in Step 3 of the construction. We need to show that in $fstate(\tau)$, $\mathit{waiting} = \emptyset$. That is, we need to show that for each preceding $\mathsf{bcast}_{i,*}$ event, there is an intervening cm-advice$(i, \cdots)$ event, for all $\langle i, * \rangle \in I_B \cup I_V$.

We can see immediately by construction that for all $i \in I_B$, each $\mathsf{bcast}_{i,*}$ event has an immediately following cm-advice$(\langle i, * \rangle, \mathsf{passive})$ event. For $v \in I_V$, the same conclusion follows: whether or not there is a recv event, some cm-advice event follows each $\mathsf{bcast}_{v,*}$ event. □

Next, we discuss the additional properties associated with the contention manager $CM_{virtual}$: eventual non-interference and eventually fairness. Before proceeding, we prove a basic lemma about when a node $v \in I_V$ is advised by $CM_{virtual}$ to be active. Specifically, we show that if virtual node v is advised to be active in round r, then it is scheduled in round r; if it is advised to be passive, then it is not scheduled. That is, the *schedule* exactly determines when v is advised to be active. The claim follows immediately from the construction of γ_v via the calculate-state function.

Lemma 11.10.5. *For $r > 0$, for $v \in I_V$, assume there exists a round r $\mathsf{recv}(\cdot, \cdot, cm, \cdot)_v$ event in γ_v. If $cm =$ active, then virtual node v is scheduled in round virtual round $r + 1$; if $cm =$ passive, then virtual node v is not scheduled in virtual round $r + 1$.*

Proof. The main argument in the proof is that the *cm* parameter of the recv transition in γ_v is determined by the calculate-state function which is used to construct executions

Figure 11-3: Constructing the Virtual Contention Manager Execution.

- Execution $\gamma_{virtual}$ begins with the point trajectory τ_0.
- Begin with $r = 1$.
- Repeat:
  1. For every $v \in I_V$, if there is a round r $\mathsf{bcast}(m, cm)_{v,*}$ event in γ_v, for any m and cm, then:
     - Insert $\mathsf{bcast}(m, cm)_{v,*}$ in $\gamma_{virtual}$.
     - If there is a round r $\mathsf{recv}(\cdot, \cdot, cm, \cdot)_{v,*}$ event in γ_v, for any cm, then insert $\mathsf{cm\text{-}advice}(\langle v, * \rangle, cm)_{virtual}$ in $\gamma_{virtual}$.
     - Otherwise, insert a $\mathsf{cm\text{-}advice}(\langle v, * \rangle, \mathsf{passive})_{virtual}$ in $\gamma_{virtual}$.
  2. For every $j \in I_B$, if there is a round r $\mathsf{bcast}(m, cm)_{j,*}$ event in γ_j, for any m and cm, then:
     - Insert $\mathsf{bcast}(m, cm)_{j,*}$ in $\gamma_{virtual}$.
     - Append a $\mathsf{cm\text{-}advice}(\langle j, * \rangle, \mathsf{passive})_{virtual}$ to $\gamma_{virtual}$.
  3. Append trajectory τ to $\gamma_{virtual}$, where $dom(\tau) = RndLength_V$.

of the virtual node; the calculate-state function uses the *schedule* to determine whether *cm* should be active or passive, implying the desired result. We now proceed in more detail.

By the construction of γ_v, if there exists a round r recv_v event in γ_v, then v is up in round r, which implies by the definition of an updown sequence that for some r_1, r_2 the following hold: (1) $r \in [r_1, r_2]$ and (2) $\langle r_1, r_2 \rangle \in \mathit{updown}(v)$.

Recall that γ_v was constructed by appending some execution in $\mathit{execs}(r_1, r')_v$, where $r' \leq r_2$ and $r \in [r_1, r']$, to an inductively constructed prefix. Thus, if there is a round r $\mathsf{recv}(\cdot, \cdot, cm, \cdot)_v$ event in γ_v, then there is some execution fragment $\gamma' \in \mathit{execs}(r_1, r')_v$ where a round r $\mathsf{recv}(\cdot, \cdot, cm, \cdot)_v \in \gamma'$.

Recall that γ' is constructed by calling the calculate-state function. Notice that in Figure 10-11, in each iteration of the loop, *inCM* is calculated in line 541cacl:schedule¿ by examining the schedule: *inCM* = active if and only if v is scheduled in round $r+1$. Moreover, the *cm* parameter of the recv action derives from *inCM* (see lines 556 and 566).

Thus, if there is a $\mathsf{recv}(\cdot, \cdot, \mathsf{active}, \cdot)_{v,*}$ in γ', then v is scheduled in round $r+1$; if there is a $\mathsf{recv}(\cdot, \cdot, \mathsf{passive}, \cdot)_{v,*}$ in γ', then v is not scheduled. Together these imply the desired result. □

Next, we show that $\gamma_{virtual}$ satisfies eventual non-interference and eventual fairness. The claim follows immediately from the fact that the *schedule*, which determines whether v is scheduled, is complete and non-interfering.

Lemma 11.10.6. *$\gamma_{virtual}$ satisfies eventual non-interference for the port set $S_2 = I_V \times \{*\}$.*

Proof. For the purpose of eventual non-interference, we consider the stabilization round to be 2. Consider some round $r \geq 2$. We need to show that if two ports $\langle v, * \rangle, \langle v', * \rangle \in S_2$ are advised to be active by $CM_{virtual}$ in round r, then $|\mathit{loc}(v) - \mathit{loc}(v')| > 2R'_V$.

If $\langle v, * \rangle$ is advised by $CM_{virtual}$ to be active in round r, then there is an advice event $\mathsf{cm\text{-}advice}(\langle v, * \rangle, \mathsf{active})_{virtual}$ in $\gamma_{virtual}$, which implies, according to the construction of $\gamma_{virtual}$, that there must have been a round $r-1$ $\mathsf{recv}(\cdot, \cdot, \mathsf{active}, \cdot)_{v,*}$ event in γ_v.

Similarly, if $\langle v', * \rangle$ is advised to be active in round r, that means that there must have been a round $r-1$ $\mathsf{recv}(\cdot, \cdot, \mathsf{active}, \cdot)_{v',*}$ event in $\gamma_{v'}$.

By Lemma 11.10.5, we know that both ports $\langle v, * \rangle$ and $\langle v', * \rangle$ are scheduled for round r. Since the schedule is non-interfering, we can therefore conclude that $|\mathit{loc}(v) - \mathit{loc}(v')| > 2R'_B + R_B = 2R'_V$, as desired. □

Finally, we show that $\gamma_{virtual}$ is an eventually fair contention manager:

Lemma 11.10.7. *$\gamma_{virtual}$ satisfies eventually fairness for $S_2 = I_V \times \{*\}$ with delay SMAX.*

Proof. For the purpose of eventual fairness, we consider the stabilization round to be 2. Consider some round $r \geq 2$. We need to show that if some port $\langle v, * \rangle \in S_2$ is not failed in rounds $[r, r + \mathsf{SMAX}]$, then there exists a round $r' \in [r, r + \mathsf{SMAX}]$

such that there is a round $r' - 1$ $\mathsf{recv}(\cdot, \cdot, \mathsf{active}, \cdot)_{v,*}$ event in γ_v. This implies, by the construction of $\gamma_{virtual}$, that $CM_{virtual}$ advises $\langle v, * \rangle$ to be active.

First, notice that the construction of γ_v guarantees that if port $\langle v, * \rangle$ is failed in a round, then it is down in that round: if $\langle u_i, d_i \rangle$ and $\langle u_{i+1}, d_{i+1} \rangle$ are elements of the sequence $updown(v)$, then v is failed in rounds $[d_i + 1, u_{i+1}]$. (If $\langle u_i, d_i \rangle$ is the last element in the sequence, then v is failed from round $d_i + 1$ onwards.)

Therefore, if $\langle v, * \rangle$ is not failed in rounds $[r, r+\mathsf{SMAX}]$, then it is up in rounds $[r, r+\mathsf{SMAX}]$. We therefore conclude that in each of these rounds, there is a $\mathsf{recv}(\cdot, \cdot, cm, \cdot)_{v,*}$ event in γ_v. Since the *schedule* is complete, there exists some $r' \in [r, r + \mathsf{SMAX}]$ such that $v \in scheduled[r' \mod \mathsf{SMAX}]$, i.e., that v is scheduled. If $cm = \mathsf{passive}$, then Lemma 11.10.5 would imply that v is not scheduled, resulting in a contradiction. Thus we can conclude that $cm = \mathsf{active}$, concluding the proof. □

11.10.3 The Client Contention Manager Execution

We now construct an execution γ_{client} of the client contention manager CM_{client}. As in the case of $\gamma_{virtual}$, the construction derives immediately from γ_i, $i \in I_B$, and γ_v, $v \in I_V$. As before, each broadcast event by a client or virtual node is added to γ_{client}. Since the client contention manager is intended to provide advice to clients, in this case, however, γ_{client} always advises virtual nodes to be passive, while deriving its cm-advice events from the recv events in γ_i, $i \in I_B$.

The construction is presented in Figure 11-4. We then discuss what it means for a client contention manager to cm-sample the "global contention manager", the contention manager in the basic system; we present two lemmas relating the client contention manager to the global contention manager. The first, Lemma 11.10.10, shows that in fact the client contention manager is a cm-sampling of the global contention manager. The second, Lemma 11.10.11, shows that if the client contends for the client contention manager, then the multiplexer contends for the global contention manager. The final result, then, is showing in Lemma 11.10.12 that if the global contention manager satisfies certain contention fairness properties, then γ_{client} satisfies some related contention fairness properties.

Since the construction is so similar to that of $\gamma_{virtual}$, the same argument shows that γ_{client} is an execution of CanonicalCM, the canonical contention manager automaton:

Lemma 11.10.8. *γ_{client} is an execution of* CanonicalCM.

Proof. Notice that γ_{client} is an alternating sequence of trajectories and actions, and that all the in γ_{client} are consistent with CanonicalCM. We need only show the following two facts: (1) the preconditions are satisfied whenever a cm-advice occurs in γ_{client}; (2) time-passage is enabled whenever τ_i is a trajectory with domain > 0.

First, consider some $\mathsf{cm\text{-}advice}(\langle i, * \rangle, \cdot)$ event, for $i \in I_B \cup I_V$. We need to show that $\langle i, * \rangle \in waiting$. From the construction, we can determine that γ_i contains an immediately preceding $\mathsf{bcast}_{i,*}$ event, which adds $\langle i, * \rangle$ to the *waiting* set, as required.

Second, consider some trajectory τ with domain > 0. That is, consider some τ that was added in Step 3 of the construction. We need to show that in $fstate(\tau)$,

Figure 11-4: Constructing the Client Contention Manager Executions.

- Execution γ_{client} begins with the point trajectory τ_0.
- Begin with $r = 1$.
- Repeat:
  1. For every $v \in I_V$, if there is a round r $\mathsf{bcast}(m, cm)_{v,*}$ event in γ_v, for any m and cm, then:
     - Insert $\mathsf{bcast}(m, cm)_{v,*}$ in γ_{client}.
     - Append a $\mathsf{cm\text{-}advice}(\langle v, * \rangle, \mathsf{passive})_{client}$ in γ_{client}.
  2. For every $j \in I$, if there is a round r $\mathsf{bcast}(m, cm)_{j,*}$ event in γ_j, for any m and cm, then:
     - Insert $\mathsf{bcast}(m, cm)_{j,*}$ in γ_{client}.
     - If there is a round r $\mathsf{recv}(\cdot, \cdot, cm, \cdot)_{j,*}$ event in γ_j, for any cm, then insert $\mathsf{cm\text{-}advice}(\langle j, * \rangle, cm)_{client}$ in γ_{client}.
     - Otherwise, insert a $\mathsf{cm\text{-}advice}(\langle j, * \rangle, \mathsf{passive})_{client}$ in γ_{client}.
  3. Append trajectory τ to γ_{client}, where $dom(\tau) = RndLength_V$.

waiting $= \emptyset$. That is, we need to show that for each preceding $\mathsf{bcast}_{i,*}$ event, there is an intervening $\mathsf{cm\text{-}advice}(\langle i, * \rangle, \cdots)$ event, for all $i \in I_B \cup I_V$.

We can see immediately by construction that for every $v \in I_V$, each $\mathsf{bcast}_{v,*}$ event has an immediately following $\mathsf{cm\text{-}advice}(\langle v, * \rangle, \mathsf{passive})$ event. For every $i \in I_B$, the same conclusion follows: whether or not there is a recv event, some $\mathsf{cm\text{-}advice}$ event follows each $\mathsf{bcast}_{i,*}$ event. □

To this point, we have made only limited assumptions about the global contention manager: we have assumed that it is conservative. With no other assumptions, we can show only that traces of the client contention manager are "sub-traces" of the global contention manager, in the following sense:

Definition 11.10.9. *We say that γ_{client} is a **cm-sampling** of some contention manager $CM \in \text{CM-names}_B$ with respect to set $S_1 = I_B \times \{*\}$ if the following holds for every virtual round $r > 1$:*

- *Let r_a be the first basic round in virtual round r.*
- *For every $i \in I_B$ that does not fail in virtual round r:*
  - *There is a round r $\mathsf{cm\text{-}advice}(\langle i, * \rangle, \mathsf{active})_{client}$ in $\gamma_{virtual}$*

if and only if

- *there is a round r_a $\mathsf{cm\text{-}advice}(\langle i, 0 \rangle, \mathsf{active})_{CM}$ in α.*

Essentially, γ_{client} is a cm-sampling of the global contention manager if its advice given at the beginning of a virtual round is equivalent to the advice given by the global contention manager at the beginning of that virtual round. The fact that γ_{client} is a cm-sampling of CM_{global} follows from the construction of γ_{client}, and the manner in which the multiplexer samples the global contention manager:

Lemma 11.10.10. *γ_{client} is a cm-sampling of CM_{global}.*

Proof. Consider some particular virtual round $r > 1$ and let r_a be the first basic round in r. Assume that i does not fail in virtual round r. We show that the construction of γ_{client} implies that there is a round r $\mathsf{cm\text{-}advice}(\langle i, * \rangle, \mathsf{active})_{client}$ in $\gamma_{virtual}$ if and only if there is a round r_a $\mathsf{cm\text{-}advice}(\langle i, 0 \rangle, \mathsf{active})_{global}$ in α.

First, we consider port $\langle i, 0 \rangle$. Since i does not fail in round r, it also does not fail in round r_a, and $r_a > 1$; hence there must be a round r_a $\mathsf{bcast}(\cdot, \cdot)_{i,0}$ event and a round r_a $\mathsf{recv}(\cdot, \cdot, gcm, \cdot)_{i,0}$ event, as per the operation of the broadcast service (see Lemmas 8.1.37 and 8.1.38). Since the regional contention managers are conservative, by assumption, we can conclude that there are no $\mathsf{cm\text{-}advice}(\langle i, 0 \rangle, \mathsf{active})_v$ events in α. Thus, we can conclude that $gcm = \mathsf{active}$ if and only if there is a $\mathsf{cm\text{-}advice}(\langle i, 0 \rangle, \mathsf{active})_{global}$ event in α.

Next, we consider port $\langle i, * \rangle$. Since i does not fail in round r and $r > 1$, we can conclude by Lemma 11.10.2 that γ_i contains a round r $\mathsf{bcast}(\cdot, \cdot)_{i,*}$ and a round r $\mathsf{recv}(\cdot, \cdot, ccm, \cdot)_{i,*}$. From this, we conclude according to the construction that the

contention manager execution that γ_{client} also contains the round r $\mathsf{bcast}_{i,*}$ event and a round r $\mathsf{cm\text{-}advice}(\langle i, * \rangle, ccm)_{client}$ event.

Our goal is to show that $ccm = \mathsf{active}$ if and only if $gcm = \mathsf{active}$. Once we have shown this fact, the conclusion follows from the preceding argument.

Moreover, this fact follows from the operation of the multiplexer. The $\mathsf{recv}_{i,*}$ event is an output of the multiplexer, and occurs at the end of round r due to the precondition on line 663, Figure 10-13. According to the pseudocode, $cm = E(multiplexer).inCM_i$.

Notice, however, that $E(multiplexer).inCM_i$ is modified in only one place: during a $\mathsf{recv}_{i,0}$ event. The $\mathsf{recv}_{i,0}$ input event occurs whenever the basic broadcast service delivers messages, i.e., at the end of each basic round. The only basic round of interest is the client round: in line 707, Figure 10-13, $E(multiplexer).inCM_i$ is modified only during the client phase. (The $E(multiplexer).rnd_i$ is a correct count of the basic rounds by Lemma 11.2.7.) We can therefore conclude that γ_{client} includes a round r $\mathsf{cm\text{-}advice}(\langle i, * \rangle, \mathsf{active})_{client}$ event if and only if there is a basic round r_a $\mathsf{recv}_{i,0}(\cdot, \cdot, \mathsf{active}, \cdot)$ event in α. As per the preceding argument, this occurs if and only if there is a round r_a $\mathsf{cm\text{-}advice}(\langle i, 0 \rangle, \mathsf{active})_{global}$ event in α, concluding our proof. □

In order to relate the two contention managers, we need one further lemma. So far, we have shown that the client contention manager is a cm-sampling of the global contention manager. We also need to show that the multiplexer contends for the global contention manager in a manner consistent with the way in which the clients contend for the client contention manager. Lemma 11.10.11 shows that if a non-failed client port $\langle i, * \rangle$ contends for the client contention manager in some virtual round r, then the multiplexer port $\langle i, 0 \rangle$ contends for the global contention manager in the basic rounds contained in $r - 1$. In fact, port $\langle i, 0 \rangle$ contends for the global contention manager throughout the entire virtual round $r - 1$, with the exception only of the first round.

In order to use Lemma 11.10.10, we need one final lemma about the multiplexer:

Lemma 11.10.11. *Assume that some port $\langle i, * \rangle$ contends for CM_{client} in virtual round r, and i does not fail prior to round r. Let r_a be the first basic round in virtual round $r - 1$. Then port $\langle i, 0 \rangle$ contends for CM_{global} in basic rounds $[r_a + 1, r_a + RndLength_V]$.*

Proof. This lemma follows immediately from the operation of the multiplexer: the precondition of the $\langle bcast \rangle_{i,0}$ event chooses $cm = \mathsf{global}$ if and only if $E(multiplexer).outCM_i = \mathsf{client}$; the only event that modifies $E(multiplexer).outCM_i$ is the $\mathsf{bcast}_{i,*}$ event which occurs only at the beginning of each virtual round (as per Lemma 11.10.2). □

We can now draw some conclusions about the properties of the client contention manager, under certain assumptions about the global contention manager: the more powerful the global contention manager, the more powerful the client contention manager; the weaker the global contention manager, the weaker the client contention manager. We give an example of how to use Lemma 11.10.10 to translate the power of one global contention manager to that of the client contention manager.

As an example of how Lemmas 11.10.10 and 11.10.11 can be used to prove properties about the client contention manager, we consider the case where CM_{global} guarantees eventual (a, b)-contention fairness for some integers $a, b > 0$. In this case, as a result of Lemma 11.10.10, we can conclude that the client contention manager guarantees eventual (a', b')-contention fairness:

Lemma 11.10.12. *Let $a, b \in \mathbb{N}$ be integers, $a, b > RndLength_V$, and assume that CM_{global} guarantees eventual (a, b)-contention fairness with radius $2R'_B + R_B$. Let $a' = \lceil a/RndLength_V \rceil$ and $b' = \lfloor b/RndLength_V \rfloor$. Then $\gamma_{virtual}$ satisfies eventual (a', b')-contention fairness for set S_1 with radius $2R'_V$.*

Proof. This lemma follows immediately from Lemmas 11.10.10 and 11.10.11. In the case of $\gamma_{virtual}$, choose the stabilization round to be the maximum of virtual round 3 and the first virtual round in which every basic rounds comes after $r_{cm} + 1$ for CM_{global} from α, posited by Definition 8.1.52. Consider some virtual round r that is no smaller than the stabilization round, and assume some port $\langle j, * \rangle$ contends for virtual rounds $[r, r + a' - 1]$.

Let r_c be the first basic round in virtual round $r - 1$, and recall that every basic round in round $r - 1$ comes after r_{cm}. By Lemma 11.10.11 we can conclude that port $\langle i, 0 \rangle$ contends in basic rounds $[r_c + 1, r_c + a' \cdot RndLength_V]$, that is, at least for basic rounds $[r_c + 1, r_c + a]$.

By assumption, we then conclude that there exists some round $r_d \in [r_c + 1, r_c + a]$ such that CM_{global} advises $\langle i, 0 \rangle$ to be active in rounds $[r_d, r_d + b - 1]$, and advises every port within distance $2R'_B + R'_B$ to be passive in rounds $[r_d, r_d + b - 1]$.

By Lemma 11.10.10, then, we can conclude that there is some round $r' \in [r, r + b']$ where $\langle i, * \rangle$ is advised to be active, and every port within distance $2R'_V$ is advised to be passive in rounds $[r, r + b']$. □

11.11 Integrity of the Virtual Broadcast Service

In this section, we prove a key lemma about the executions of the clients and the virtual nodes that we use to show that the virtual broadcast service guarantees integrity. (The execution for the virtual broadcast service is construct later in Section 11.13.) The main goal of this section is to show that if some node $i \in I_B \cup I_V$ receives some message m in virtual round r, then some other node $j \in I_B \cup I_V$ broadcast message m in round r. The result is stated formally in Lemma 11.11.12.

The main structure of the proof is broken down into two lemmas: Lemma 11.11.7 deals with the case where the receiving node i is a client; Lemma 11.11.11 deals with the case where the receiving node v is a virtual node.

In both lemmas, the first step is to trace the message back to its origin, i.e., to determine from where it originated. Any message that originates from a client $i \in I_B$ in round r is relatively easy to trace, and Lemmas 11.11.1 and 11.11.2 in Section 11.11.1 show that if a message $\langle \mathsf{client}, m, \cdot \rangle$ ever ends up either in the multiplexer (about to be delivered to a client), or in a ballot (about to be delivered to a virtual node), then there is some client j that performs a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ in γ_j, as is desired.

The more difficult case is tracking back messages of the form $\langle \mathsf{vn}, v', m \rangle$, i.e., messages that originate at some virtual node emulator. The main difficulty is to show that if message m is received, then a $\mathsf{bcast}(m, \cdot)_{v,*}$ event occurs in γ_v. The two cases diverge depending on whether the receiving node is a client or a virtual node.

If the receiving node is a client (Section 11.11.4), then some multiplexer must have delivered the message to the client via a $\mathsf{vn\text{-}client\text{-}output}_v$ transition, and by examining the emulator the produced this event (i.e., the subscript v on the event), we can identify which virtual node broadcast the message. Since a vn-client-output only outputs a message under certain conditions, we can then argue that v must be scheduled, and that the round itself must be green for v. Moreover, the message itself must have been included in some ballot b broadcast by an emulator for v. At this point we can invoke Lemma 11.11.5 (see Section 11.11.3), a key lemma which shows, loosely, that any message of the form $\langle \mathsf{vn}, v, m \rangle$ included in a ballot for node v in a good round is broadcast by virtual node v.

If the receiving node is a virtual node v (Section 11.11.5), somewhat more work is needed to trace back the origin of the message. First, we argue that the message must be in some ballot for v. Next, we identify the origin of the message: in the easy case, it is sent by v itself. Otherwise, we claim that it is only in the ballot for v if it was previously in some ballot for $v' \neq v$. Next, we show that v' must be scheduled, and v unscheduled (Lemma 11.11.8). We then show that, in this case, round r must be a good round for v' (Lemma 11.11.9) and there must be a round r ballot for v' containing the message (Lemma 11.11.10). At this point, we conclude the proof as in the previous case by invoking Lemma 11.11.5.

11.11.1 Tracing Messages Broadcast by Clients

Any message m received by a client or virtual node was originally introduced into the system by either a client or an emulator. We begin with two easy lemmas that trace messages of the form $\langle \mathsf{client}, m, \cdot \rangle$ back to the clients that originally broadcast these messages. Each lemma is a straightforward back-tracing of the unique sequence of transitions that place a message of the form $\langle \mathsf{client}, m, \cdot \rangle$ in the specified set.

Lemma 11.11.1. *Let α' be a prefix of α, $i \in I_B$ be a node, $m \in msgs_V$ a message (i.e., $\neq \perp$). Assume that in $\ell state(\alpha')$, $\langle \mathsf{client}, m, \cdot \rangle \in E(multiplexer).inM_i$. Then a $\mathsf{bcast}(m, \cdot)_{i,*}$ occurs in α'.*

Proof. We trace the message backwards along the only possible sequence of transitions that can arrive at $\langle \mathsf{client}, m, \cdot \rangle \in E(multiplexer).inM_i$: from the multiplexer, back to the $\mathsf{recv}_{i,0}$ event, back to the $\mathsf{bcast}_{i,0}$ event, back to the $\mathsf{bcast}_{i,*}$ event at which it originated.

- The only transition that adds a message of the form $\langle \mathsf{client}, m, \cdot \rangle$ to $E(multiplexer).inM_i$ is $\mathsf{recv}(M, \ldots)_{i,0}$ where $m \in M$ and $E(multiplexer).rnd_i$ indicates that the recv event is a client phase receive event for some virtual round r.
- The only ports that broadcast in the client phase of round r are ports $\langle j, 0 \rangle$, for $j \in I_B$. Thus, by the integrity of the basic broadcast service (Lemma 8.1.35),

there must be some $j \in I_B$ and some $\mathsf{bcast}(\langle \mathsf{client}, m, \cdot \rangle)_{j,0}$ event in α' preceding the $\mathsf{recv}_{i,0}$ event.

- Since $m \neq \perp$, we can conclude that $E(multiplexer).outM_j \neq \perp$ immediately prior to the $\mathsf{bcast}_{j,0}$ event. The only transition that sets $outM \neq \perp$ is $\mathsf{bcast}_{j,*}$.
- Thus, we conclude there must be a $\mathsf{bcast}(m, \cdot)_{j,*}$ event in α' that precedes the $\mathsf{bcast}_{j,0}$ event.

□

This second lemma assumes that a message $\langle \mathsf{client}, m, \cdot \rangle$ has been places in a ballot b for virtual round r. As before, the message is traced back to an originating broadcast.

Lemma 11.11.2. *Let $i \in I_B$ be a node, $m \in msgs_V$ a message ($\neq \perp$). Assume that for some ballot b, node i broadcasts a $\langle \mathsf{vn}, v, b \rangle$ ballot message such that $\langle \mathsf{client}, m, \cdot \rangle \in b.clientM$. Then a $\mathsf{bcast}(m, \cdot)_{i,*}$ occurs in α and precedes the broadcast of the ballot.*

Proof. We trace the message backwards along the only possible sequence of transitions that can arrive at $\langle \mathsf{client}, m, \cdot \rangle \in b.clientM$: to the $E(v).clientM_i$ set, to the client phase $\mathsf{recv}_{i,v}$ transition, to the client phase $\mathsf{bcast}_{j,0}$ event, to the originating $\mathsf{bcast}_{j,*}$ event.

- Immediately prior to broadcasting ballot b, we know that $\langle \mathsf{client}, m, \cdot \rangle \in E(v).clientM_i$.
- The only transition that adds a message of the form $\langle \mathsf{client}, m, \cdot \rangle$ to $E(v).clientM_i$ is a client phae $\mathsf{recv}(M, \ldots)_{i,v}$ where $m \in M$.
- The only ports that broadcast in the client phase of round r are ports $\langle j, 0 \rangle$, for $j \in I_B$. Thus, by the integrity of the basic broadcast service (Lemma 8.1.35), there must be some $j \in I_B$ and some $\mathsf{bcast}(\langle \mathsf{client}, m, \cdot \rangle)_{j,0}$ event in α' preceding the $\mathsf{recv}_{i,0}$ event.
- Since $m \neq \perp$, we can conclude that $E(multiplexer).outM_j \neq \perp$ immediately prior to the $\mathsf{bcast}_{j,0}$ event. The only transition that sets $outM \neq \perp$ is $\mathsf{bcast}_{j,*}$.
- Thus, we conclude there must be a $\mathsf{bcast}(m, \cdot)_{j,*}$ event in α' that precedes the $\mathsf{bcast}_{j,0}$ event.

□

11.11.2 Preliminary Lemmas Related to Ballots and Executions

We now consider messages of the form $\langle \mathsf{vn}, v, \cdot \rangle$, i.e., messages purported to be sent by virtual nodes. The goal of this subsection and Section 11.11.3 is to prove Lemma 11.11.5, which shows that if some message $\langle \mathsf{vn}, v, m \rangle$ is included in a ballot broadcast by an emulator for virtual node v, and if the ballot is broadcast in a good round r, then virtual node v does in fact broadcast message m in round r in γ_v.

This lemma is an important building block for Lemma 11.11.12, which shows that the client and virtual node executions satisfy the integrity property.

In order to prove Lemma 11.11.5, we need two preliminary lemmas, presented in this section. The first lemma provides the basis for the claim in Lemma 11.11.5 in the special case where: (1) $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node v, (2) v is scheduled for round r_2, and (3) round r_2 is not red. Note that this lemma does not address the case where v is down in round r_2; the following Lemma 11.11.4 considers this case.

The main claim in Lemma 11.11.3 is that if some message $\langle \mathsf{vn}, v', m \rangle$ is in the $b.vnM$, where b is the round r_2 ballot, and if i is the proposer that sends ballot b (in the scheduled-ballot phase), then we can conclude that the next external action in any extension of $exec(r_1, r_2 - 1, i)_v$ is the broadcast of message m. This lemma is then used in Lemma 11.11.5 by noting that since i is the round r proposer, γ_v includes– and hence extends—the execution fragment $exec(r_1, r_2, i)_v$, and hence message m is broadcast in round r_2 in γ_v, as desired.

The main idea behind the proof is to first argue that if i broadcasts a ballot containing message m, it must also have broadcast message m in the vn phase. Thus, i calculates the last state s in $exec(r_1, r_2 - 1, i)_v$ in the client phase for the purpose of choosing the message to broadcast in the vn, and then calls do-bcast$(s)_v$ to calculate the message, which we have already posited is m, the message included in the ballot.

Lemma 11.11.3. *Assume that $\langle r_1, r_2 \rangle$ delineates a finite epoch of virtual node $v \in I_V$, assume that v is scheduled for round r_2, and that $r_2 > r_1$. Moreover, assume that round r_2 is not red for v. Let i be the proposer for round r_2 and b be the ballot for round r_2. We conclude the following:*

- *If for some $m \in msgs_V$, message $\langle \mathsf{vn}, v, m \rangle \in b.vnM$, and if s is the last state in $exec(r_1, r_2 - 1, i)_v$, then $\langle \cdot, m, \cdot, \cdot \rangle =$ do-bcast$(s)_v$.*

Proof. The proof proceeds in three steps. First, we argue that i broadcasts some message $\langle \mathsf{vn}, v', m' \rangle$ in the vn phase, since by assumption it broadcasts a message in the scheduled-ballot phase. Second, we show that $m' = m$. Finally, we conclude the proof by arguing that message m is calculated by examining the last state in $exec(r_1, r_2 - 1, i)_v$ and calling the do-bast function, as described in the lemma statement. We now proceed in more detail.

Step 1. First, notice that node i, the proposer, broadcasts a message in the scheduled-ballot phase, since v is scheduled. We therefore argue that i also broadcasts a message in the vn phase: We know that $E(v).scheduled_i =$ true, $E(v).joined_i =$ true, and $E(v).roundCM_=$active when $E(v).outgoing\text{-}msg_i \leftarrow b$ at the end of the vn phase, in preparation for the scheduled-ballot phase. These same settings must hold at the end of the client phase when the message is chosen for the vn phase, as they are not changed during the vn phase. In addition, since $E(v).last\text{-}reset_i = r_1 - 1$ by Corollary 11.4.5, we know that $r_2 \neq E(v).last\text{-}reset_i + 1$. Thus all the conditions are met for *outgoing-msg* to be set $\neq \perp$ in line 240. We therefore conclude that node i broadcasts a message in the vn phase; assume that this message is $\langle \mathsf{vn}, v, m' \rangle$, for some message $m' \in msgs_V \cup \{\perp\}$.

Step 2. Second, we argue that $m' = m$. That is, at the end of the client phase, $E(v).outgoing\text{-}msg_i = \langle \mathsf{vn}, v, m \rangle$.

We trace the message $\langle \mathsf{vn}, v, m' \rangle$ forward from its broadcast in the vn phase by port $\langle i, v \rangle$ (posited in Step 1) until the formation of ballot b. Notice that $\langle \mathsf{vn}, v, m' \rangle$ is in the set $E(v).allM_i$ immediately prior to the recv event that concludes the vn phase: this follows immediately by the self-delivery property of the basic broadcast service which guarantees that node i receives every message that it sends (Lemma 8.1.36). Moreover, the message $\langle \mathsf{vn}, v, m' \rangle$ is in $E(v).nearby\text{-}msgs_i$ after line 257 during the recv event for the vn phase of round r_2.

Assume, for the sake of contradiction, that $m' \neq m$. In this case, there are at least two messages in $E(v).nearby\text{-}msgs_i$: $\langle \mathsf{vn}, v, m \rangle$ and $\langle \mathsf{vn}, v, m' \rangle$. As a result of the condition on line 264 (Figure 10-5), the set $E(v).vnM_i$ is left unmodified, empty. This implies that $\langle \mathsf{vn}, v, m \rangle \notin E(v).vnM_i$, and hence contradicts our assumption that $\langle \mathsf{vn}, v, m \rangle \in b.vnM$. Hence we can conclude that $m = m'$.

Step 3. The last step of the proof is to notice that i chooses the message *next-vn-msg* by first calculating $temp\text{-}state = exec(r_1, r_2{-}1, i)_v$, and then calculating $\langle next\text{-}vn\text{-}msg, \cdot \rangle = \mathsf{do\text{-}bcast}(temp\text{-}state)_v$. Node i then broadcasts $\langle \mathsf{vn}, v, next\text{-}vn\text{-}msg \rangle$, which concludes our proof. □

The second lemma in this section deals with the case where v is down in some virtual round r. Even when a virtual node is down in a virtual round, it may broadcast a message in that round if it does not fail until after the round begins. Recall from the construction of γ_v that when a virtual node is down in a virtual round, it either fails at some point during that round, or begins the round already failed. In this lemma, we show that if there is a broadcast by v in a virtual round, and it is down in that round, then it must be up in the previous virtual round. More specifically, we show that if some non-empty ballot is sent in round r, then even though v is down in round r, v must be up in round $r - 1$.

Lemma 11.11.4. *Let $r > 0$ be a virtual round, $v \in I_V$ a virtual node. Assume that v is down in round r, and that there is some node $i \in I_B$ that broadcasts a ballot b in the* scheduled-ballot *phase of round r such that $\langle \mathsf{vn}, v, m \rangle \in b.vnM$. Then, $r > 1$ and v is up in round $r - 1$.*

Proof. If v is down in round $r{-}1$, then we can conclude that either (1) $E(v).last\text{-}reset_i = r - 1$ at the beginning of the vn phase of round r, in which case $b.vnM = \emptyset$, or (2) there are no nodes that begin round r for v. Both cases imply a contradiction.

We now proceed in more detail. Assume for the sake of contradiction that one of two cases holds:

- $r = 1$, or
- $r > 1$ and v is down in round $r - 1$.

(As a technical matter, when we refer to the end of round $r - 1$ in the case where $r = 1$, we are actually referring to the beginning of round 1, i.e., the beginning of the execution.)

Since i broadcasts a ballot in the scheduled-ballot phase, we can conclude that i begins round r: node i broadcasts a ballot only if it has $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$ when the vn phase recv occurs, which is only the case when i begins round r, as there is no opportunity to join v prior to the join phase of round r.

If $r > 1$, there are two possible reasons why v is down in round $r-1$ (according to Definition 11.3.7): either no node completes round $r-1$, or virtual node v is reset in round $r-1$. Thus, along with the case where $r = 1$, there are three possible cases to cover:

1. $r = 1$, or
2. $r > 1$ and v is down in round $r-1$ due to the fact that some node resets v in round $r-1$,
3. $r > 1$ and v is down in round $r-1$ due to the fact that no node completes round $r-1$.

We consider these cases as follows: (1,2) $r = 1$ or some node resets v in round $r-1$, (3) $r > 1$ and no node completes round $r-1$.

(1,2). Either $r = 1$ or $r > 1$ and virtual node v is reset in round $r-1$: If $r > 1$, we apply Lemma 11.4.2 to conclude that at the beginning of round r, $E(v).last\text{-}reset_i = r-1$. If $r = 1$, then at the beginning of round 1, $E(v).last\text{-}reset_i = 0$ and $E(v).rnd_j = 0$, as each remains set to its initial value. After the client phase recv of round 1, $E(v).rnd_j = 1$, implying that $E(v).last\text{-}reset_i = r-1$. Thus, if $r = 1$ or if v is reset in round $r-1$, then $E(v).last\text{-}reset_i = r-1$.

As a result, there are no message of the form $\langle \mathsf{vn}, v, \cdot \rangle$ sent in the vn phase of round r (see line 238, Figure 10-5). Thus, by the integrity of the basic broadcast service, there are no messages of the form $\langle \mathsf{vn}, v, \cdot \rangle$ received in the vn phase of round r. Since the ballot data structure $b.vnM$ contains a subset of the messages received in the vn phase of round r by the round r proposer, we can conclude that $\langle \mathsf{vn}, v, m \rangle \notin b.vnM$, resulting in a contradiction.

(3). At the end of virtual round $r-1$, for every $j \in I_B$, node j does not complete round $r-1$, i.e., does not begin round r. This contradicts our claim that i begins round r.

□

11.11.3 Round r Ballot Implies a Round r Broadcast

In this section, we prove a key lemma which describes when a bcast event occurs in γ_v. Specifically, if some round r is good for virtual node v, and $b.vnM \neq \emptyset$, where b is the round r ballot, then γ_v includes a broadcast event for the message contained in the ballot. Both of the main lemmas that prove the integrity of the virtual broadcast service, Lemma 11.11.7 and Lemma 11.11.11, use this lemma as the final step to show that some message was actually broadcast.

Lemma 11.11.5. *Assume virtual round $r > 0$ is good for virtual node $v \in I_V$, and that v is scheduled for round r. Moreover, assume that some node $i \in I_B$ broadcasts a ballot b such that message $\langle \mathsf{vn}, v, m \rangle \in b.vnM$ in the* scheduled-ballot *phase of round r. Then γ_v contains a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event.*

Proof. There are two cases based on whether v is up or down in round r:

- Consider the case where v is up in round r:
  The proof proceeds as follows: First, we identify some r_1, r_2 such that some $\gamma' \in \mathit{execs}(r_1, r_2)_v$ is included in γ. Second, we argue that $r > r_1$, and, as a result, conclude by Lemma 11.7.21 that γ' extends $\mathit{exec}(r_1, r-1, i)_v$, where i is the round r proposer. Third, we apply Lemma 11.11.3 to show that every extension of $\mathit{exec}(r_1, r, i)_v$ contains a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event, and hence γ_v contains such an event. We now proceed in more detail.

  Step 1. Choose $r_1 \leq r$ such that $\langle r_1, r \rangle$ delineates a finite epoch of v. Choose r_2 such that $r \in \langle r_1, r_2 \rangle$ and $\langle r_1, r_2 \rangle \in \mathit{updown}(v)$; we know this is possible according to the construction of the updown sequence. If $r_2 = \infty$, set $r_2 = \max(r_{gst} + 1, r)$. Recall that γ_v is constructed by appending some $\gamma' \in \mathit{execs}(r_1, r_2)_v$ to an already constructed prefix.

  Step 2. We now show that $r_1 < r$. Assume for the sake of contradiction that $r_1 = r$. Notice that for every node $j \in I_B$, j does not send a message $\langle \mathsf{vn}, v, \cdot \rangle$ in the vn phase of round r: according to line 238 (Figure 10-5), a message is broadcast only if $E(v).\mathit{rnd}_j \neq E(v).\mathit{last\text{-}reset}_j + 1$ and $E(v).\mathit{joined}_j = \mathsf{true}$; it is also immediately clear that a message is broadcast by port $\langle j, v \rangle$ only if $E(v).\mathit{failed}_j = \mathsf{false}$. In this case, however, according to Corollary 11.4.5, $E(v).\mathit{last\text{-}reset}_j = r_1 - 1$ at the beginning of round r, implying that port $\langle j, v \rangle$ does not boadcast any messages in the vn phase of round r.

  Since no message of the form $\langle \mathsf{vn}, v, \cdot \rangle$ is broadcast in the vn phase of round r, we conclude by the integrity of the basic broadcast service that no message of this form is received in the vn phase of round r. As a result, when i forms ballot b on line 275, message $\langle \mathsf{vn}, v, m \rangle$ is not in $b.vnM$, contradicting our prior assumption. Thus we conclude that $r_1 < r$.

  Since $r_1 < r$, we apply Lemma 11.7.21 to show that for every $j \in I_B$ that completes round r_2, $\mathit{exec}(r_1, r_2, j)_v$ extends $\mathit{exec}(r_1, r-1, i)_v$; hence γ' extends $\mathit{exec}(r_1, r-1, i)_v$.

  Step 3. By Lemma 11.11.3, we conclude that if s is the last state in $\mathit{exec}(r_1, r-1, i)_v$, then $\langle m, \cdot \rangle = \mathsf{do\text{-}bcast}(s)_v$.

  Examine the calculation performed by calculate-state in extending $\mathit{exec}(r_1, r-1, i)_v$ to form round r of γ': we begin with the last state of $\mathit{exec}(r_1, r-1, i)_v$ and performs a $\mathsf{do\text{-}bcast}(s)_v$ followed by a $\mathsf{do\text{-}recv}(\ldots)_v$. Since $\langle m, \cdot \rangle = \mathsf{do\text{-}bcast}(s)_v$, we can conclude that a $\mathsf{bcast}(m, \cdot)_{v,*}$ occurs in γ', and hence in γ_v.

- Consider the case where v is down in round r:
  The proof generally proceeds as follows: from the construction of γ_v, we identify

a node j' that broadcasts a message $\langle \mathsf{vn}, v, m' \rangle$ in the vn phase of round r and a ballot b' containing $\langle \mathsf{vn}, v, m' \rangle$ in the scheduled-ballot phase of round r; if $j' = i$, then we can conclude that $m' = m$, concluding the proof; otherwise, if $j' \neq i$, we can conclude that the round is not good; resulting in a contradiction.

We know by Lemma 11.11.4 that $r > 1$ and v is up in virtual round $r - 1$. Choose r_1 such that $\langle r_1, r-1 \rangle$ delineates a (finite) epoch of v. We can conclude that $\langle r_1, r-1 \rangle \in \mathit{updown}(v)$, since v is down in round r.

Recall that γ_v is constructed by appending some execution $\gamma' \in \mathit{execs}(r_1, r-1)_v$ to a previously constructed prefix, followed by a do-bcast(s) event, where s is the last state in γ'. Moreover, recall that the execution γ' is chosen from $\mathit{execs}(r_1, r-1)_v$ by selecting $j' \in I_B$ such that $\gamma' = \mathit{exec}(r_1, r-1, j')_V$ and j' broadcasts some message $\langle \mathsf{vn}, v, m' \rangle$, $m \neq \perp$, in the vn phase and some message $\langle \mathsf{vn}, v, \cdot \rangle$ in the scheduled-ballot phase.

Since j' chooses message m' to broadcast in the vn phase by calculating $\mathit{exec}(r_1, r-1, j')_v$ and calling do-bcast, we can conclude that $\mathsf{bcast}(m', \cdot)_{v,*}$ is the next enabled external event in state s. Thus, the event $\mathsf{bcast}(m', \cdot)_{v,*}$ occurs in γ_v in round r.

It remains to show that $m' = m$. Assume not, for the sake of contradiction. Recall that virtual node v is scheduled in round r. Since $\langle \mathsf{vn}, v, m \rangle$ is in the round r ballot, some node j must have broadcast $\langle \mathsf{vn}, v, m \rangle$ in the vn phase. This would result in two messages broadcast in the vn phase: $\langle \mathsf{vn}, v, m \rangle$ and $\langle \mathsf{vn}, v, m' \rangle$; moreover nodes i, j, and j' are all within distance $R_B/4$ of $\mathit{loc}(v)$ since they have $E(v).\mathit{joined} = \mathsf{true}$; hence node i either receives two messages in the vn phase of round r, or node i detects a collision (by Lemma 8.1.40); either case would result in $E(v).\mathit{vnM}_i = \emptyset$, and hence a ballot b with $b.\mathit{vnM} = \emptyset$, contradicting our assumption. We thus conclude that $m' = m$, and therefore a $\mathsf{bcast}(m, \cdot)_{v,*}$ occurs in γ_v in round r.

□

11.11.4 Clients Receiving Messages

In this section, we prove one of the two main lemmas: Lemma 11.11.7 shows that when a client receives a message m, some other client or virtual node broadcast that message. In the case where message m originates at a client, we use Lemma 11.11.1 to trace the message back to the sender. In the case where message m originates at a virtual node, we show that the message appears in some ballot, and then invoke Lemma 11.11.5 to conclude the proof.

Recall that Lemma 11.11.5 holds only when a virtual round is good. We first prove a preliminary lemma which shows that a round r is good when certain conditions hold. When a node is participating in a virtual node, it is relatively straightforward to determine when a round is good; specifically, a round is good if $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$. However, if a node is not participating in a virtual node, then this does not immediately follow. Lemma 11.11.6, then, is of particular use in the case where some

node $i \in I_B$ delivers a message to a client, but is not participating in the virtual node. (When node i is participating in the virtual node, the lemma follows immediately.)

Lemma 11.11.6. *Let $v \in I_V$ be a virtual node, $r > 0$ a virtual round, and $i \in I_B$ a node that does not fail prior to the end of the last agreement phase of virtual round r. Assume the following three conditions hold:*

1. *Virtual node v is scheduled for virtual round r.*
2. *$E(v).round\text{-}status[r]_i = \mathsf{green}$ at the end of the last agreement phase of virtual round r.*
3. *$E(v).ballot[r].vnM_i \neq \emptyset$ at the end of the last agreement phase of virtual round r.*

Then round r is good for v.

Proof. This lemma follows from Lemma 11.5.6: By assumption $i \in I_B$ is non-failed through the end of the last agreement phase of virtual round r. Moreover, by assumption, $E(v).round\text{-}status[r]_i = \mathsf{green}$ at the end of the last agreement phase. It remains only to show that node i is within distance $3R_B/4$ of $loc(v)$ at the beginning of the scheduled-veto-2 phase of virtual round r, and the conclusion follows.

By assumption, $E(v).ballot[r].vnM_i \neq \emptyset$ at the end of the scheduled-veto-2 phase of virtual round r. Thus, we can conclude that $E(v).ballot[r].vnM_i \neq \emptyset$ at the end of the scheduled-veto-1 phase, as the ballot data structure is filled only in the vn phase and the ballot phase.

In the scheduled-veto-1 phase recv event, however, the ballot is reset if $|loc - loc(v) > R_V + R_B/4$. Thus, we can conclude that node i is within distance $3R_B/4$ at the end of the scheduled-veto-1 phase when the recv event occurs, and hence at the beginning of the scheduled-veto-2 phase (which occurs at the same instant in time as the scheduled-veto-1 recv event). □

We can now prove the main lemma showing the integrity of the basic broadcast service, with respect to clients receiving messages. Notice that most of the proof is related to the case where the client receives a message from a virtual node; tracing a message back to a client is a straightforward application of Lemma 11.11.1.

Lemma 11.11.7. *Let $r > 0$ be a virtual round, $i \in I_B$ a node, $m \in msgs_V$ a message, and $M \subseteq msgs_V$ a set of messages. If a round r $\mathsf{recv}(M, \cdot, \cdot, \cdot)_{i,*}$ occurs in γ_i, then for every $m \in M$ there exists some $j \in I_B \cup I_V$ such that a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ occurs in γ_j.*

Proof. We first divide the proof into two cases, depending on whether the message originated at a client or a virtual node. If a round r $\mathsf{recv}(M, \cdot, \cdot, \cdot)_{i,*}$ occurs in γ_i, then for every $m \in M$, either $\langle \mathsf{client}, m, \cdot \rangle \in E(multiplexer).inM_i$ or $\langle \mathsf{vn}, m, \cdot \rangle \in E(multiplexer).inM_i$, as enforced by the precondition in lines 664–667, Figure 10-13.

We next dispose of the straightforward case where the message originates with a client. Consider the case where $\langle \mathsf{client}, m, \cdot \rangle \in E(multiplexer).inM_i$. By Lemma 11.11.1, we conclude that a $\mathsf{bcast}(m, \cdot)_{i,*}$ occurs in α, and hence in γ_i, as required.

For the remainder of the proof, we consider the case where a message originates at a virtual node. That is, consider the case where $\langle \mathsf{vn}, m, \cdot \rangle \in E(\mathit{multiplexer}).\mathit{inM}_i$. We begin by tracing the message back to a specific vn-client-output transition which delivered the message to the multiplexer. We then examine the conditions under which the vn-client-output event occurs, leading to the conclusion that v is scheduled and that the round status is green for i at the end of the virtual round. We then show that the round is good for v (using Lemma 11.11.6 in one case), and conclude that for some node $k \in I_B$, port $\langle k, v \rangle$ broadcast a ballot b containing message m. We then apply Lemma 11.11.5 to conclude the proof. We now proceed in more detail.

There are two lines in which messages are added to the set inM: lines 651 and 708 (Figure 10-13). In the first case, all the messages added have the form $\langle \mathsf{client}, \cdot, \cdot \rangle$. Hence we can assume that the message $\langle \mathsf{vn}, m, \cdot \rangle$ is added as a result of a vn-client-output$(\langle \mathsf{vn}, m, \cdot \rangle, \cdot)_{i,v}$ transition.

Next, we examine the preconditions under which a vn-client-output transition from $E(v)_i$ produces message m:

- $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$ at the end of round r, as per line 177.
- Virtual node v is scheduled in round r, as per line 178.
- $\langle \mathsf{vn}, v, m \rangle \in E(v).\mathit{ballot}[r].\mathit{vnM}_i$, $m \neq \perp$, at the end of round r, as per line 179.

There are now two cases, depending on whether i joins v in round r. In either case, we know that $E(v).\mathit{failed}_i = \mathsf{false}$ through the end of round r, as a vn-client-output transition occurs.

- Assume i joins v in round r: We know that i does not reset v in round r, as $E(v).\mathit{round\text{-}status}[r]_i \neq \perp$ (see line 488) at the end of round r. Thus, virtual node v is up in round r: node i is non-failed through the end of round r, and node i prevents the virtual node being reset in the join-veto phase.

  Let basic round r_a be the last agreement phase of virtual round r, and let r_b be the join-ack phase in virtual round r; thus $r_b > r_a$. Since $i \in I_B$ completes the join protocol, we know that $E(v).\mathit{joined}_i = \mathsf{true}$ and $E(v).\mathit{failed}_i = \mathsf{false}$ at the end of basic round r_b. Therefore by Lemma 11.5.3, we conclude that there exists some $j \in I_B$ where j participates in virtual round r and at the end of basic round r_a, $E(v).\mathit{round\text{-}status}[r]_j = E(v).\mathit{round\text{-}status}[r]_i$ at the end of round r_b; also at the end of basic round r_a, $E(v).\mathit{ballot}[r]_j = E(v).\mathit{ballot}[r]_i$ at the end of round r_b.

  Since $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$ at the end of round r_2 and the virtual node is not reset in round r, we conclude that the round status is also green at the end of r_b, and hence $E(v).\mathit{round\text{-}status}[r]_j = \mathsf{green}$ at the end of the last agreement phase of r, and hence by definition round r is green.

  By the same logic, we can conclude that $\langle \mathsf{vn}, v, m \rangle \in E(v).\mathit{ballot}[r].\mathit{vnM}_j$ at the end of the last agreement phase of r. Moreover, by the integrity of the basic broadcast service (Lemma 8.1.35), we can conclude that some node k broadcast the ballot b received by j, i.e., $\langle \mathsf{vn}, v, m \rangle \in b.\mathit{vnM}$.

- Assume i does not join v in round r: In this case, i does not join or reset virtual node v, and hence i does not modify $E(v).\mathit{round\text{-}status}[r]_i$ or $E(v).\mathit{ballot}[r]_i$ after the last agreement phase of r prior to the end of round r. Thus, at the end of the last agreement phase of r, $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$ and $\langle \mathsf{vn}, v, m\rangle \in E(v).\mathit{ballot}[r].\mathit{vnM}_i$, $m \neq \bot$. Hence by Lemma 11.11.6, round r is good for v.

  Moreover, by the integrity of the basic broadcast service (Lemma 8.1.35), we can conclude that some node k broadcast the ballot b received by i, i.e., $\langle \mathsf{vn}, v, m\rangle \in b.\mathit{vnM}$.

In either case, we have conclude that round r is good, virtual node v is scheduled for round r, and that some node k broadcasts a ballot b in the scheduled-ballot phase where $\langle \mathsf{vn}, v, m\rangle \in b.\mathit{vnM}$. We then apply Lemma 11.11.5 to conclude that γ_v contains a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event. □

11.11.5 Virtual Nodes Receiving Messages

In this section, we prove the second of the two main integrity lemmas: Lemma 11.11.11 shows that when a virtual node v receives a message m in some round r of γ_v, then some other client or virtual node broadcast message m in round r. As before, the case where a virtual node receives a message from a client is relatively straightforward, and handled (for the most part) by Lemma 11.11.2. The majority of this section is examining the case where the message originates at some virtual node $v' \in I_V$.

We begin with three preliminary lemmas. First, we show that a virtual node v can only receive a message from another virtual node v' in some virtual round r if v' is scheduled and v unscheduled. The emulator does not allow virtual nodes to successfully broadcast messages when they are not scheduled: any such messages are lost. Thus we conclude from the operation of the emulator that v' is scheduled. Moreover, the emulator only delivers messages from nearby virtual nodes. Since v receives a message from v', we can conclude that the two virtual nodes are not too far apart, and thus we conclude from the non-interference property of the *schedule* that v is unscheduled.

Lemma 11.11.8. *Let $r > 0$ be a virtual round, $v \in I_V$ a virtual node, $m \in \mathit{msgs}_V$ a message, and $M \subseteq \mathit{msgs}_V$ a set of messages.*

Assume, that for some ballot b, there is a $\langle \mathsf{vn}, v, b\rangle$ message broadcast in either the scheduled-ballot *or* unscheduled-ballot *phase of round r, and $\langle \mathsf{vn}, v', m\rangle \in b.\mathit{vnM}$, for some $v' \in I_V$, $v \neq v'$. Then we conclude that virtual node v is not scheduled and virtual node v' is scheduled for round r.*

Proof. First, assume for the sake of contradiction virtual node v' is not scheduled. Assume node i is a node that broadcasts $\langle \mathsf{vn}, v, b\rangle$ in a ballot round. Since by assumption $\langle \mathsf{vn}, v', m\rangle \in b.\mathit{vnM}$, we can conclude that $\langle \mathsf{vn}, v', m\rangle \in E(v).\mathit{vnM}_i$ when ballot b is formed in the vn phase. Thus we can conclude that $\langle \mathsf{vn}, v', m\rangle \in E(v).\mathit{nearby\text{-}msgs}_i$ during the vn phase recv transition. However, in this case, since v' is not scheduled,

the condition on line 268 ensures that $E(v).vnM_i$ is left unmodified, that is, $= \emptyset$. This contradicts our previous assumption, from which we conclude that v' is scheduled.

Next, assume for the sake of contradiction that both virtual nodes v and v' are scheduled. In this case, we can conclude that v and v' are "far enough" apart, that is, v and v' are distance $> 2R'_V$ apart, since the schedule is non-conflicting. Let i be a node that broadcasts the ballot message $\langle \mathsf{vn}, v, b \rangle$. As before, every message that node i adds to the *vnM* field of the ballot is received in the vn phase. Moreover, since i broadcasts a ballot, we can conclude that i is within distance $R_B/4$ of i at the end of the vn phase; otherwise i no longer remains joined, and hence does not send a ballot. According to line 257, however, i includes in the set *nearby-msgs* only messages sent by virtual nodes v' within distance R_V of v, and every message that eventually ends up in the ballot $b.vnM$ was previously in *nearby-msgs*. Thus, since $\langle \mathsf{vn}, v', m \rangle \in b.vnM$ where ballot b is sent by port $\langle i, v \rangle$, we can conclude that v is within distance $R_V < 2R'_V$ of v', contradicting our conclusion that v and v' are far enough apart due to the non-conflicting property of the schedule.

Hence, we conclude that v' is scheduled, then v is not scheduled for virtual round r. □

Next, we show that a round is good under certain conditions. Specifically, the emulator delivers messages from virtual node v' only in rounds that are good for v'. (In rounds that are not good for v', there may be emulators that do not agree on the execution, and hence it is important that no externally visible behavior depend on the current state of v.) We show that if a ballot for virtual node v includes a messages from virtual node v', then the round must be good for v':

Lemma 11.11.9. *Let $v, v' \in I_V$ be two virtual nodes, $r > 0$ a virtual round, and assume that v is not scheduled for round r, and v' is scheduled for round r. If $\langle \mathsf{vn}, v, b \rangle$, for some ballot b, is broadcast in the* unscheduled-ballot *phase of round r, and $\langle \mathsf{vn}, v', m \rangle \in b.vnM$, then round r is good for v'.*

Proof. Assume for the sake of contradiction that round r is not good for v'. The proof proceeds by identifying the node $j \in I_B$ that has a round status of red or orange, and noticing that this node broadcasts a veto message in the scheduled-veto-2 phase. As a result, the node which broadcasts ballot b either receives the veto message, detects a collision, or is too far away from v' for either of the two preceding options. In all cases, the ballot-broadcasting node drops message $\langle \mathsf{vn}, v', m \rangle$, and does not include it in the ballot, resulting in a contradiction. We now proceed in more detail.

Since round r is not good for v', there exists some $j \in I_B$ that participates in v' and has $E(v').round\text{-}status[r]_j$ equal to red or orange at the end of the scheduled-veto-2 phase. Notice that $E(v').round\text{-}status[r]_j$ is copied from $E(v').scheduled\text{-}status_j$ in the scheduled-veto-2 phase, and $E(v').scheduled\text{-}status_j$ cannot be set to red or orange after the scheduled-veto-1 recv event. Thus, we can conclude that by the end of the scheduled-veto-1 recv event, $E(v').scheduled\text{-}status_j$ is red or orange.

Also, recall that since j participates in v', j has $E(v').joined_j$ set to true. Therefore node j sets *outgoing-msg* to $\langle \mathsf{vn}, v', \mathsf{veto} \rangle$, broadcasting a veto message in the

scheduled-veto-2 phase. Moreover, we know that node j is within distance $R_B/4$ of $loc(v)$.

Let $i \in I_B$ be the node that broadcasts ballot b. There are now two cases to consider: either i is "near enough" to $loc(v')$ or i is "far" from $loc(v')$. First, consider the case where i is within distance $3R_B/4$ of $loc(v')$ at the beginning of the scheduled-veto-2 phase. In this case, by Lemma 8.1.40, node i either receives the veto message or detects a collision. In either case, $E(v).scheduled\text{-}status_i$ is set to some value $\neq \perp$, and as a result $E(v).vnM_i$ is set to $\emptyset$ during the scheduled-veto-2 phase recv event, as is required.

Consider the case where i is not within distance $3R_B/4$ of $loc(v')$ at the beginning of the scheduled-veto-2 phase. In this case, during the scheduled-veto-1 phase recv event, $E(v).vnM_i$ is reset to $\emptyset$, and remains empty through the end of the scheduled-veto-2 phase, as required.

Thus, in either case, $b.vnM = \emptyset$, which contradicts our original assumption and conclude the proof. □

The third preliminary lemma traces messages backwards from a ballot for v to a ballot for v'. If a message from v' is included in a ballot for v, then this message must have previously been included in a ballot for v'. This lemma is key to the proof of Lemma 11.11.11 in that once the message has been traced back to a ballot for v', we can invoke Lemma 11.11.5 to draw the desired conclusion.

Lemma 11.11.10. *Let $v, v' \in I_V$ be two virtual nodes, $r > 0$ a virtual round, and assume that v is not scheduled for round r, and v' is scheduled for round r. If $\langle \mathsf{vn}, v, b \rangle$, for some ballot b, is broadcast in the* unscheduled-ballot *phase of round r, and $\langle \mathsf{vn}, v', m \rangle \in b.vnM$, then there is some ballot b' broadcast in the* scheduled-ballot *phase of round r where $\langle \mathsf{vn}, v', m \rangle \in b.vnM$.*

Proof. We trace back the series of necessary events for $\langle \mathsf{vn}, v', m \rangle$ to appear in $b.vnM$:

When some node j broadcasts the ballot message $\langle \mathsf{vn}, v, b \rangle$ in the unscheduled-ballot phase, we can conclude that $b.vnM = E(v).vnM_j$, and hence $\langle \mathsf{vn}, v', m \rangle \in E(v).vnM_j$.

According to line 299 (Figure 10-6), every message $\langle \mathsf{vn}, v', m \rangle$ in $E(v).vnM_j$ is contained in $b'.vnM$ for some ballot b' received as $\langle \mathsf{vn}, v', b' \rangle \in allM$ during the scheduled-ballot phase. The basic broadcast service integrity (Lemma 8.1.35) guarantees that some node broadcast ballot $\langle \mathsf{vn}, v', b' \rangle$, as required. □

Finally, we can put the pieces together and prove the main lemma of Section 11.11: if a virtual node v receives a message in γ_v, then some client or virtual node must have broadcast that message.

Lemma 11.11.11. *Let $r > 0$ be a virtual round, $v \in I_V$ a virtual node, $m \in msgs_V$ a message, and $M \subseteq msgs_V$ a set of messages. If a round r $\mathsf{recv}(M, \cdot, \cdot, \cdot)_{v,*}$ occurs in γ_v, then there exists some $j \in I_B \cup I_V$ such that a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ occurs in γ_j.*

Proof. The main structure of the proof proceeds as follows: first, we argue that for some round r ballot b for v, either $\langle \mathsf{client}, m, \cdot \rangle \in b.clientM$ or $\langle \mathsf{vn}, v', m \rangle \in b.vnM$. (A third alternative is that v sent the message m itself in round r.) In the case where

the message originates at a client, we apply Lemma 11.11.2. In the case where the message originates at a virtual node, we apply Lemma 11.11.10 to trace the message back to a ballot from v', and then Lemma 11.11.5 to conclude the proof.

First, notice that by the construction of γ_v, if v is down in round r, then there is no round r recv event in γ_v. Thus we can conclude that v is up in round r.

Since there is a round r $\mathsf{recv}(M, \ldots)_{v,*}$ in γ_v and v is up in round r, we know that there is some $\langle r_1, r_2 \rangle \in \mathit{updown}(v)$ such that $r \in [r_1, r_2]$. If $r_2 = \infty$, set $r_2 = \max(r_{gst} + 1, r)$. Thus γ_v is constructed using an execution $\gamma' \in \mathit{execs}(r_1, r_2)_v$, and as a result there is a round r $\mathsf{recv}(M, \ldots)_{v,*}$ in execution γ'. By Lemma 11.7.20, then, there is some prefix $\gamma'' \leq \gamma'$, $\gamma'' \in \mathit{execs}(r_1, r)_v$ where there is a round r $\mathsf{recv}(M, \ldots)_{v,*}$ in γ''. Choose $j \in I_B$ where j completes round r such that $\gamma'' = \mathit{exec}(r_1, r, j)_v$, as described in Lemma 11.7.20. By Lemma 11.5.3, there exists some node $k \in I_B$ that participates in round r and at the end of the last agreement phase of r has $E(v).\mathit{ballot}[r] k_{=} E(v).\mathit{ballot}[r]_j$.

Notice, then, that according to the construction of $\mathit{exec}(r_1, r, j)_v$ and the calculate-state function (lines 550, 555, and 565, Figure 10-11), since there is a $\mathsf{recv}(M, \ldots)_{v,*}$ event in γ'', we can conclude that:

$$\begin{aligned} M \quad \subseteq \quad & \{m' : \langle \mathsf{client}, m', \cdot \rangle \in E(v).\mathit{ballot}[r].\mathit{clientM}_j\} \cup \\ & \{m' : \langle \mathsf{vn}, v', m' \rangle \in E(v).\mathit{ballot}[r].\mathit{vnM}_j, v \neq v'\} \cup \\ & \{\mathit{temp\text{-}msg}\} \end{aligned}$$

In this case, *temp-msg* is the message broadcast by virtual node v in round r (see Figure 10-11). Thus, we can conclude that at the end of the last agreement phase of r, message m is in one of the following sets:

- $\{m' : \langle \mathsf{client}, m', \cdot \rangle \in E(v).\mathit{ballot}[r].\mathit{clientM}_k\}$, at the end of the last agreement phase of r,
- $\{m' : \langle \mathsf{vn}, v', m' \rangle \in E(v).\mathit{ballot}[r].\mathit{vnM}_k, v \neq v'\}$, at the end of the last agreement phase of r,
- $\{\mathit{temp\text{-}msg}\}$, the round r messages broadcast in $\mathit{exec}(r_1, r, j)_v$.

If there is a round r $\mathsf{bcast}(m, \ldots)_{v,*}$ event in γ_v, then the claim is satisfied. Assume, then, that no such round r event occurs in γ_v. We conclude that $\mathit{temp\text{-}msg} \neq m$.

Next, notice that initially $\mathit{ballot}[r]$ is a default, empty, value. It is set to be non-empty only in lines 317 and 386, when the ballot is received. We have shown, above, that ballot $b = E(v).\mathit{ballot}[r]_k$ is modified from its default initial value by the end of the last agreement phase. Choose $k' \in I_B$ to be the node that broadcast that ballot b for virtual node v. (We know such a node exists by the integrity of the basic broadcast service, Lemma 8.1.35.)

Thus, either $\langle \mathsf{client}, m, \cdot \rangle \in b.\mathit{clientM}$ or $\langle \mathsf{vn}, v', m \rangle \in b.\mathit{vnM}$, $v' \neq v$. By Lemma 11.11.2, in the former case we conclude that there is a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ for some $j \in I_B$.

Assume, then, that $\langle \mathsf{vn}, v', m \rangle \in b.\mathit{vnM}$, for some $v' \in I_V$, $v' \neq v$. (Notice that, at this point, nothing we have argued indicates that there is only one such message for one such v'; we arbitrarily choose one of them.)

According to Lemma 11.11.8, we can conclude that v' is scheduled and v is unscheduled. We will show that v', the scheduled virtual node, broadcast message m, and v, the unscheduled virtual node, received the message.

Since k' broadcasts a ballot for virtual node v, and v is unscheduled, we can conclude that k' does not fail prior to the beginning of the unscheduled-ballot phase. Moreover, $\langle \mathsf{vn}, v', m\rangle \in E(v).ballot[r].vnM_{k'}$.

We now apply Lemma 11.11.9 to show that round r is good for v': by assumption, there is some message $\langle \mathsf{vn}, v', m\rangle \in b.vnM$. By Lemma 11.11.10, we conclude that some node $i \in I_B$ broadcasts a ballot b' such that message $\langle \mathsf{vn}, v', m\rangle \in b.vnM$ in the scheduled-ballot phase of round r.

Finally, by Lemma 11.11.5, we conclude that γ_v contains a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event, as required. □

11.11.6 Putting the Pieces Together

We now combine Lemma 11.11.7 and Lemma 11.11.11 to state the main result: if any node $i \in I_B \cup I_V$ receives a message m in round r in an execution γ_i, then there is some $j \in I_B \cup I_V$ that broadcast message m in round r in execution γ_j.

Lemma 11.11.12. *Let $r > 0$ be a virtual round, $i \in I_B \cup I_V$ a node, $m \in msgs_V$ a message, and $M \subseteq msgs_V$ a set of messages. If a round r $\mathsf{recv}(M, \cdot, \cdot, \cdot)_{i,*}$ occurs in γ_i, then there exists some $j \in I_B \cup I_V$ such that a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ occurs in γ_j.*

Proof. The claim follows immediately from Lemma 11.11.7 and Lemma 11.11.11. □

11.12 The (Virtual) Collision Detector

In this section we discuss the collision detector that is integrated into the virtual broadcast service. (We construct the actual virtual broadcast service execution in Section 11.13; we will need the results in this section to complete the construction.) Recall (from Chapter 8), that a collision detector is modelled as a set of "collision detector rules" which defines the behavior of the collision detector in each round. The virtual collision detector encompasses the set of all rules that are complete and eventually accurate. We will show that the rule constructed is both complete (Lemma 11.12.10) and eventually accurate (Lemma 11.12.16), and hence a valid rule for the virtual collision detector.

11.12.1 Defining the Virtual Collision Detector Rule

In this section, we define a virtual collision detector rule. We define the collision detector rule by extracting from α information on collisions actually detected. This process ensures that the rule is consistent with an execution of the virtual broadcast service: whenever the virtual broadcast service delivers a collision in γ_j, $j \in I_B \cup I_V$, the collision detector rule specifies that a collision should be detected.

In Section 11.12.2, we show that the virtual collision detector rule is complete, and in Section 11.12.3 we show that the virtual collision detector rule is eventually accurate. Thus we conclude that the collision detector rule is in the set of rules specified for the virtual infrastructure system.

Before defining the collision detector rule itself, we first define the following functions, which we use to extract the set of broadcast events:

- The virtual round r broadcast events for the virtual broadcast service:

$$BC(r) = \{e : \ e \text{ is a round } r \ \mathsf{bcast} \text{ event in } \gamma_i,\ i \in I_B \cup I_V\} \ .$$

- For all $k \in I_V$, let $\ell\text{-}pre(k)$ be the location of node k at the beginning of round r, according to the broadcast service.

- The messages broadcast by round r broadcast events near node j: $sentM(j,r) = \{m : \mathsf{bcast}(m,\cdot)_{k,*} \in BC(r), |\ell\text{-}pre[j] - -\ell\text{-}pre[k]| \leq R_V, k \in I_B \cup I_V\}$.

- The messages broadcast by round r broadcast events that may *interfere* with node j: $sentMinterfere(j,r) = \{m : \mathsf{bcast}(m,\cdot)_{k,*} \in BC(r), |\ell\text{-}pre[j] - -\ell\text{-}pre[k]| \leq R'_V, k \in I_B \cup I_V\}$.

It should be noted that the set $sentM(j,r)$ corresponds to the set *sentM* on line 49 of Figure 8-5. Similarly, the set $sentM(j,r)'$ corresponds to the set *sentMinterfere* on line 51 of Figure 8-5. It is this correspondence that is used to show that the rule constructed is consistent with the execution of the virtual broadcast service. We now define $CD\text{-}rule_V$ in Figure 11-5.

11.12.2 Completeness

The goal of this section is to show that the $CD\text{-}rule_V$ defined in Figure 11-5 is complete, as per Definition 8.1.12: if more messages are broadcast than are received, then a collision is detected. Since the collision detector rule is defined from information extracted from executions γ_i, for $i \in I_B$, and γ_v, for $v \in I_V$, the proof involves examining broadcast and receive events in these executions and showing that they imply that the collision detector rule is complete. Specifically, we need to show that if some non-failed node $i \in I_B \cup I_V$ broadcasts a message m in some virtual round r, and some non-failed node $j \in I_B \cup I_V$ that is nearby does not receive that message, then j detects a collision in virtual round r. By the manner in which the collision detector rule is constructed, this is sufficient to show completeness.

In general, the proof breaks down into four cases, depending on whether i and j are clients or virtual nodes. In each case, we assume for the sake of contradiction that some message m is lost, yet no collision is detected. We then trace the progress of message m from the sender to the receiver, arguing at each step that either the message successfully arrives at the next step along the path, or a collision is generated, which results in the receiver detecting a collision. In this way, we produce a contradiction.

Figure 11-5: Constructing the Virtual Collision Detector Rule.

For every $j \in \mathit{bcast\text{-}ports}_V$, for every $r > 0$:

- If a round r $\mathsf{recv}(M, cd, \cdot, \cdot)_j$ event occurs in γ_j, for some $M \subseteq \mathit{msgs}$:
  - Let $p = |\mathit{sentM}(j, r)|$.
  - Let $q = |M \cap \mathit{sentM}(j, r)|$.
  - Let $p' = |\mathit{sentM}(j, r)'|$.
  - Let $q' = |M \cap \mathit{sentM}(j, r)'|$.
  - Define $\mathit{CD\text{-}rule}(\langle j, * \rangle, r, p, q, p', q')_V = cd$.
- For all other values $p, q, p', q' \in \mathbb{N}$ not defined above:

$$\mathit{CD\text{-}rule}(i, r, p, q, p', q')_V = \begin{cases} \mathsf{null}, & \text{if } p \leq q \\ \pm, & \text{if } p > q \end{cases}$$

We begin with a few preliminary lemmas. We then consider the case where the node receiving the messages is a client. Next, we consider the case where the node receiving the messages is a virtual node. We conclude with the main proof of completeness.

Preliminary Lemmas

We begin by proving two preliminary lemmas that are used in Sections 11.12.2 and 11.12.2 to show that the virtual collision detector rule is complete.

The first of these two lemmas, Lemma 11.12.1 shows that if some virtual node v' receives a message in round r and does not detect a collision, then we can conclude that round r is not red for v'; moreover, the round r ballot for v contains exactly the messages in set M, and includes no collisions. (As a point of notation, notice that throughout this section we tend to use the letter v to represent a virtual node that broadcasts a message and the letter v' to represent a virtual node that receives some messages.) When proving that the collision detector rule is complete, in cases where the receiving node is a virtual node, we can focus our attention on the round r ballot.

Lemma 11.12.1. *Let $v' \in I_B$ be a virtual node and $r > 0$ a virtual round. Assume that a round r $\mathsf{recv}(M, cd, \ldots)_{v',*}$ occurs where $cd = \mathsf{null}$ in $\gamma_{v'}$. Then:*

1. *Round r is not red for v'.*
2. *Let b be the (unique) round r ballot for v', and let i be the round r proposer for v'. Then, in addition, we conclude:*
   - $M = \{m : \langle \mathsf{client}, m, \cdot \rangle \in b.clientM$ **or** $\langle \mathsf{vn}, \cdot, m \rangle \in b.vnM\}$.
   - $b.clientCD = \mathsf{null}$.
   - $b.vnCD = \mathsf{null}$.

Proof. The proof depends primarily on examining the calculate-state function, and noticing that if round r is red, then the execution fragment constructed by calculate-state always includes a round r $\mathsf{recv}(\cdot, \pm, \ldots)_{v',*}$ event, which contradicts the lemma's hypothesis. Thus, the first step of the proof is to identify an execution fragment containing round r that is constructed by calculate-state and used during the construction of $\gamma_{v'}$. The conclusions then follows from a line-by-line examination of calculate-state (Figure 10-11).

First, notice that virtual node v' is up in round r: the construction of $\gamma_{v'}$ only includes a $\mathsf{recv}_{v',*}$ event in round r if v' is up. Since v' is up in round r, then we know that it is included in an epoch of the updown sequence: let $\langle r_1, r_2 \rangle \in updown(v')$ be the pair that delineates an epoch of v' such that $r \in [r_1, r_2]$. If $r_2 = \infty$, then set $r_2 = \max(r_{gst} + 1, r)$ so that $\langle r_1, r_2 \rangle$ is a finite epoch. Since the construction of $\gamma_{v'}$ uses the epochs from the updown sequence to construct execution fragments (with a special case when an epoch is infinite), we can choose γ to be the execution in $execs(r_1, r_2)_{v'}$ that is used in the construction $\gamma_{v'}$. By Lemma 11.7.20, there exists some node k' that completes round r and γ is an extension of $exec(r_1, r, k')_{v'}$. We

have now identified an execution γ that includes round r and is constructed by the calculate-state function based on the state of node k' at the end of round r.

Next, we examine how calculate-state constructs round r of execution γ. Specifically, notice that the construction branches based on whether $\textit{temp-status}[r] = \mathsf{green}$ or red (line 545, Figure 10-11). If the latter branch is chosen, i.e., if $\textit{temp-status}[r] = \mathsf{red}$, then do-recv is called with the second parameter equal to $\pm$, resulting in a round r $\mathsf{recv}(\cdot, \pm, \ldots)_{v',*}$ in γ. However, we have assumed that $cd = \mathsf{null}$, and hence can conclude that $\textit{temp-status}[r] = \mathsf{green}$. The array *temp-status* is calculated by node k' by calling calculate-status (Figure 10-10) based on the state of k' in the client phase of round r. By (the contrapositive of) Lemma 11.6.8, we conclude that round r is not red for v'.

Finally, notice that in the case where $\textit{temp-status}[r] = \mathsf{green}$, calculate-state constructs the set *inM* (line 550, Figure 10-11) as the set:

$$\{m : \langle \mathsf{client}, m, \cdot \rangle \in b.\textit{clientM} \textbf{ or } \langle \mathsf{vn}, \cdot, m \rangle \in \cup b.\textit{vnM}\} .$$

Since calculate-state calls do-recv with the parameter *inM*, we conclude that $M = \textit{inM}$, implying the desired result.

Similarly, calculate-state calculates the collision detection parameter *inCD* such that if either *b.clientCD* or *b.vnCD* is equal to $\pm$, then $\textit{inCD} = \pm$. However, since $cd = \mathsf{null}$ in the round r recv event in $\gamma_{v'}$, we can conclude that both *b.clientCD* and *b.vnCD* are equal to null. □

The second preliminary lemma, Lemma 11.12.2, considers the case where a virtual node v broadcasts some message m. In this case, we show that there exists some node i that broadcasts $\langle \mathsf{vn}, v, m \rangle$ in the vn phase of round r. That is, any message broadcast in γ_v in round r is also broadcast in the vn phase of round r. When proving that the virtual collision detector rule is complete, in the cases where a virtual node broadcasts a message, we can focus our attention on the message broadcast in the vn phase.

Lemma 11.12.2. *Let $v \in I_B$ be a virtual node, and $r > 0$ a virtual round. Assume that a round r $\mathsf{bcast}(m, \cdot)_v$ occurs in γ_v where $m \neq \perp$. Assume that one of the following three cases holds:*

- *Round r is good for v.*
- *Virtual node v is unscheduled for round r.*
- *Virtual node v is down in round r.*

Then there exists some $i \in I_B$ that begins round r for v and remains joined and not failed through the beginning of the scheduled-ballot *phase such that:*

1. *There is a round r $\mathsf{bcast}(\langle \mathsf{vn}, v, m \rangle, \cdot)_{i,v}$ event in the* vn *phase of round r.*
2. *If v is scheduled for round r, then there is a round r $\mathsf{bcast}(\langle \mathsf{vn}, v, b \rangle, \cdot)_{i,v}$ event in the* scheduled-ballot *phase of round r.*

Proof. We begin with an overview of the proof, which divides into two cases, based on whether v is up or down in round r. If v is down in round r, the claim follows immediately from the construction of γ_v: there is a broadcast in round r of γ_v only if there is a message broadcast by some port $\langle i, v\rangle$ in the vn and scheduled-ballot phases of round r. Since γ_v is constructed by the calculate-state function, which calculates the round r message in the same way that node i calculates the message to broadcast in the vn phase, we can conclude that port $\langle i, v\rangle$ broadcasts $\langle \mathsf{vn}, v, m\rangle$ in the vn phase, as claimed.

If v is up in round r, the proof is slightly more complicated. We first identify an execution fragment γ that includes round $r-1$ and is used in the construction of γ_v. We further restrict our attention to $exec(r_1, r-1, i)_v$, for some $i \in I_B$, and argue that γ is an extension of $exec(r_1, r-1, i)_v$, according to Lemma 11.7.20. Moreover, i completes round $r-1$ and remains joined and not failed until the scheduled-ballot phase of round r. From this we conclude that in the last state of $exec(r_1, r-1, i)_v$, a broadcast of message m is the next enabled external event. There are two subcases to consider: If v is scheduled in round r, then node i is the round r proposer, and hence has $E(v).roundCM_i = \mathsf{active}$. (This fact depends on the assumption that round r is good for v.) If v is not scheduled, then $E(v).scheduled_i = \mathsf{false}$. Thus in either case, port $\langle i, v\rangle$ satisfies line 239, and we are able to conclude that i broadcasts the message $\langle \mathsf{vn}, v, m\rangle$ in the vn phase, as claimed. It follows then that if v is scheduled, i also broadcasts a message in the scheduled-ballot phase.

We now proceed in more detail. There are two cases to consider, depending on whether v is up or down in r:

- Assume v is down in round r. By the construction of γ_v, the only way in which a bcast event occurs in round r is the following: there exists some epoch $\langle r_1, r-1\rangle \in updown(v)$, and there exists some node $i \in I_B$ such that port $\langle i, v\rangle$ broadcasts a message $\neq \perp$ in the vn and scheduled-ballot phases of round r. Thus, port $\langle i, v\rangle$ broadcasts a message in the vn phase, as desired, and i does not fail prior to the scheduled-ballot phase. It is easy to conclude that the message is of the form $\langle \mathsf{vn}, v, \cdot\rangle$.

  During the construction of γ_v, we see that m is chosen as follows: assume s is the last state of $exec(r_1, r-1, i)_v$; v broadcasts the message specified by $\mathsf{do\text{-}bcast}(s)_v$ in round r of γ_v. When port $\langle i, v\rangle$ chooses a message to broadcast in the vn phase of round r (the calculation occurs in the client phase recv of round r), it follows the same procedure, and hence we can conclude that there is a $\mathsf{bcast}(\langle \mathsf{vn}, v, m\rangle, \cdot)_{i,v}$ event in the vn phase of round r.

- Assume v is up in round r. Since v is up in round r, then we know that it is included in an epoch of the updown sequence: let $\langle r_1, r_2\rangle \in updown(v)$ be the pair that delineates an epoch of v such that $r \in [r_1, r_2]$. If $r_2 = \infty$, then set $r_2 = \max(r_{gst} + 1, r)$ so that $\langle r_1, r_2\rangle$ is a finite epoch. Since the construction of γ_v uses the epochs from the updown sequence to construct execution fragments (with a special case when an epoch is infinite), we can choose γ to be the execution in $execs(r_1, r_2)_v$ that is used in the construction γ_v. We also know

that since there is a round r bcast event in γ_v, virtual round node v is not down in round $r-1$, and hence $r > r_1$.

Thus, by Lemma 11.7.20, there exists some node $i \in I_B$ that completes round $r-1$ and does not fail or set $E(v).joined_i = \mathsf{false}$ prior to the $\mathsf{bcast}_{i,v}$ event in the scheduled-ballot phase, and γ is an extension of $exec(r_1, r-1, i)_v$.

Since there is a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event in γ_v and γ_v is an extension of γ, there is also a round r $\mathsf{bcast}_{v,*}$ event in γ, and thus we can conclude that in the last state s of $exec(r_1, r-1, i)_v$, the do-bcast(s) produces message m, as this is the procedure by which $exec(r_1, r-1, i)_v$ is extended to γ. Thus, we can conclude that in the client phase, when choosing which message to send in the vn phase, *next-vn-msg* is set equal to m in line 237.

If v is scheduled for round r, then, since v is up, we know by assumption, round r is good for v. (Recall we have assumed that one of three situations held: either round r is good for v, v is not scheduled for round r, or v is down in round r.) Thus, we can conclude that i is the round r proposer for v, since according to Corollary 11.7.15, γ extends the proposer's $[r_1, r-1]$ execution. In this case, we conclude that $E(v).roundCM_i = \mathsf{active}$ at the beginning of the vn phase, since i is the round r proposer.

Thus, at the end of the client phase, when the message is selected for the vn phase, we know the following: By the way in which i was chosen, we know that $E(v).joined_i = \mathsf{true}$ and $E(v).failed_i = \mathsf{false}$. Since $r > r_1$, we know that $E(v).last\text{-}reset_i + 1 \leq E(v).rnd_i$ (by Corollary 11.4.5). Either v is scheduled, in which case $E(v).roundCM_i = \mathsf{active}$, or v is not scheduled, in which case $E(v).scheduled_i = \mathsf{false}$. Thus, since $m \neq \perp$, node i sets $E(v).outgoing\text{-}msg_i = \langle \mathsf{vn}, v, m \rangle$ at the end of the client phase, and hence broadcast $\langle \mathsf{vn}, v, m \rangle$ in the vn phase of round r, as desired.

Moreover, since i remains joined and not failed through the $\mathsf{bcast}_{i,v}$ event in the scheduled-ballot phase of round r, it is easy to see that if v is scheduled for round r, then at the end of the vn phase: (1) $E(v).scheduled_i = \mathsf{true}$; (2) $E(v).joined_i = \mathsf{true}$; and (3) $E(v).roundCM_i = \mathsf{active}$. Thus, there is a $\mathsf{bcast}(\langle \mathsf{vn}, v, b \rangle, \cdot)_{i,v}$ event in the scheduled-ballot phase of round r, concluding the proof.

□

Delivering Messages to a Client

Next, we consider a particular client $j \in I_B$ that receives some set of messages M in virtual round r and does not detect a collision. Our goal is to show that every message broadcast by a nearby node—be it real or virtual—is delivered to j. We begin with Lemma 11.12.3, which shows that every messages broadcast by a nearby client is received by node j. This first lemma, Lemma 11.12.3, mainly focuses on the multiplexer, showing that the messages are correctly passed from the broadcasting client to the multiplexer, broadcast by the multiplexer using the basic broadcast

service, received by node j's multiplexer, and then passed from node j's multiplexer to node j.

Lemma 11.12.3. *Assume that $i, j \in I_B$ are two (possibly distinct) clients, and $r > 0$ is a virtual round. Assume that a round r $\mathsf{bcast}(m, \cdot)_{i,*}$, $m \neq \perp$, occurs in γ_i and a round r $\mathsf{recv}(M, \mathsf{null}, \ldots)_{j,*}$ event occurs in γ_j. If j is within distance R_V of i, then $m \in M$.*

Proof. In this case, we trace message m forward from the sending port $\langle i, * \rangle$ to the receiving port $\langle j, * \rangle$. The message is first passed from the client to the multiplexer at i; it is then broadcast by the multiplexer at i and received by the multiplexer at j; it is then passed from the multiplexer at j to the client at j. The only step in which the message can be lost is when it is broadcast over the basic broadcast service from the multiplexer at i to the multiplexer at j. If the message is lost, then the multiplexer at j detects a collision, since the collision detector for the basic broadcast service is complete. This collision notification is then passed to the client at j, contradicting the assumption that $cd = \mathsf{null}$.

We now proceed in more detail. Since clients only broadcast on port $\langle i, * \rangle$ at the beginning of a virtual round (as an immediate response to the round $r - 1$ $\mathsf{recv}_{i,*}$ event), we conclude that message m is broadcast in the client phase by port $\langle i, * \rangle$; that is, that there is a $\mathsf{bcast}(m, \cdot)_{i,*}$ event in the client phase of round r in α. At this point, $E(\mathit{multiplexer}).\mathit{outM}_i \leftarrow m$.

Next, the multiplexer broadcasts message m on port $\langle i, 0 \rangle$. (Since the multiplexer flag $E(\mathit{multiplexer}).\mathit{clientBcast}_i = \mathsf{true}$, the $\mathsf{bcast}_{i,0}$ event immediately follows the $\mathsf{bcast}_{i,*}$ event with no intervening time passage.) Thus we conclude that there is a $\mathsf{bcast}(\langle \mathsf{client}, m, \cdot \rangle, \cdot)$ event in the client phase of round r (see line 694, Figure 10-13).

Since the basic broadcast service is complete and i and j are within distance $R_V \leq R_B$ at the beginning of round r (i.e., the beginning of the client phase), either port $\langle j, 0 \rangle$ receives this broadcast of message m, or port $\langle j, 0 \rangle$ detects a collision (by Lemma 8.1.40). That is, either a $\mathsf{recv}(m, \ldots)_{j,0}$ event occurs in the client phase of round r, or a $\mathsf{recv}(\cdot, \pm, \ldots)_{j,0}$ event occurs in the client phase of round r. The first case implies that $m \in M$, as desired; the second case implies a contradiction, from which we conclude that this case does not occur under the assumptions of the lemma.

In the former case, where port $\langle j, 0 \rangle$ receives $\langle \mathsf{client}, m, \cdot \rangle$, the multiplexer at j adds message $\langle \mathsf{client}, m, \cdot \rangle$ to $E(\mathit{multiplexer}).\mathit{inM}_j$ (line 708); the only transition that removes messages from inM is the $\mathsf{recv}_{j,*}$ event, and thus the round r $\mathsf{recv}_{j,*}$ event delivers message m (see line 664). This implies that $m \in M$.

In the latter case, where port $\langle j, 0 \rangle$ detects a collision in the client phase, the collision detection flag $E(\mathit{multiplexer}).\mathit{inCD}_j \leftarrow \pm$ (line 709). No other round r event can set $\mathit{inCD} \neq \pm$, and hence the round r $\mathsf{recv}_{j,*}$ event delivers a collision to port $\langle j, * \rangle$, contradicting our assumption that $cd = \mathsf{null}$ in the round r $\mathsf{recv}_{j,*}$ event. □

Our next goal is to prove the equivalent of Lemma 11.12.3, but in the case where the broadcasting node is a virtual node (instead of a client). The main part of this argument will be showing that when virtual node v broadcasts a message in round r, the $\mathsf{vn\text{-}client\text{-}output}_{j,v}$ delivers this message to the client (via the multiplexer). We

need a few further preliminary lemmas in this case. We first show that when a virtual node broadcasts a message, then every node that is nearby and non-failed detects that some messages was broadcast in the vn phase. (The latter follows primarily from Lemma 11.12.2.) As a result, we can show in Lemma 11.12.6 that the vn-client-output event delivers the message to client j.

Lemma 11.12.4. *Let $v \in I_V$ be a virtual node and $r > 0$ a virtual round. Assume that a round r $\mathsf{bcast}(m, \cdot)_v$ occurs in γ_v, $m \neq \perp$, and let $j \in I_B$ be a node that is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the* scheduled-ballot *phase and at the beginning of the last agreement phase, and assume that $E(v).round\text{-}status[r]_j \in \{\mathsf{green}, \perp\}$ at the end of the last agreement phase of round r. Then we can draw the following conclusion:*

- *For all $k' \in I_B$ that are not failed at the end of round r and that are within distance R_V of $loc(v)_V$ at the beginning of round r, $E(v).vnphaseBcast_{k'} = \mathsf{true}$ at the end of virtual round r.*

Proof. We begin the lemma by arguing that round r is good for v, and by identifying some node k that broadcasts a message for v in the vn and ballot phases (as per Lemma 11.12.2). We argue that any node k' that is not failed and sufficiently close either receives the message $\langle \mathsf{vn}, v, m \rangle$ sent in the vn phase, or detects a collision in the vn phase; both cases result in k' detecting that a broadcast occurred in the vn phase, i.e., $E(v).vnphaseBcast_{k'} = \mathsf{true}$. We now proceed in more detail.

First, we argue that round r is good for v: since $E(v).round\text{-}status[r]_j \in \{\mathsf{green}, \perp\}$ at the end of the last agreement phase of round r, we conclude by Lemma 11.5.6, round r is good for v.

Therefore we conclude by Lemma 11.12.2 that there exists some $k \in I_B$ that begins round r for v and remains joined and not failed through the beginning of the scheduled-ballot phase such that there is a round r $\mathsf{bcast}(\langle \mathsf{vn}, v, m \rangle, \cdot)_{k,v}$ event in the vn phase of round r in α.

Fix some $k' \in I_B$ that does not fail prior to the end of virtual round r, and assume that k' is within distance R_V of $loc(v)_V$ at the beginning of round r. Since the velocity of a node is limited, we can conclude that k' is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the vn phase. Since k remains joined and not failed through the scheduled-ballot phase, by assumption, we know that k is within distance $R_B/4$ of $loc(v)_V$ at the beginning of the vn phase. Thus, we conclude that k and k' are within distance R_B of each other at the beginning of the vn phase. As a result, in the vn phase, k' either receives the $\langle \mathsf{vn}, v, m \rangle$ message, or detects a collision, as per the completeness of the basic broadcast service (Lemma 8.1.40).

First, assume k' receives the message $\langle \mathsf{vn}, v, m \rangle$. Since k is within distance $R_B/4$ of $loc(v)$, the vn phase recv transition reaches line 262. In this case, $\langle \mathsf{vn}, v, m \rangle \in E(v).nearby\text{-}msgs_{k'}$ after line 257 (Figure 10-5). Thus, on line 263 (Figure 10-5), the flag $E(v).vnphaseBcast_{k'} \leftarrow \mathsf{true}$.

Next, consider the case where k' detects a collision in the vn phase of round r. In this case, again, on line 263 (Figure 10-5), $E(v).vnphaseBcast_{k'} \leftarrow \mathsf{true}$.

Thus, we conclude that in either case, at the end of virtual round r either $E(v).vnphaseBcast_{k'} = \mathsf{true}$.

□

A second fact we need before proving Lemma 11.12.6 is that the round status of a node is not $\perp$, if that node is close to the virtual node. This lemma follows from a straightforward examination of the pseudocode.

Lemma 11.12.5. *Let $v \in I_V$ be a virtual node, $j \in I_B$ be a node, and $r > 0$ a virtual round. If j is within distance $R_B/2$ of $loc(v)_V$ at the end of the last agreement phase, then $E(v).round\text{-}status[r]_j \neq \perp$ at the end of the last agreement phase.*

Proof. The proof breaks down into two cases, depending on whether v is scheduled or unscheduled in round r. If v is scheduled, then in the scheduled-veto-2 phase, $E(v).round\text{-}status[r]_j$ is assigned a non-$\perp$ value. If v is unscheduled, then in the unscheduled-veto-2 phase, $E(v).round\text{-}status[r]_j$ is assigned a non-$\perp$ value.

In more detail, consider the case where v is scheduled, and examine lines 359–362 in the scheduled-veto-2 phase (Figure 10-7). In the case where $E(v).scheduled\text{-}status_j = \perp$, since $E(v).scheduled_j = \mathsf{active}$ and j is within distance R_V of v, we can conclude that $E(v).round\text{-}status[r]_j = \mathsf{green}$. In the case where $E(v).scheduled\text{-}status_j \neq \perp$, $E(v).round\text{-}status[r]_j \leftarrow E(v).scheduled\text{-}status_j$. In both of these cases, the status $E(v).round\text{-}status[r]_j \neq \perp$ at the end of the scheduled-veto-2 phase.

Consider the case where virtual node v is not scheduled, and examine line 426 in the unscheduled-veto-2 phase (Figure 10-8). Notice that if $E(v).round\text{-}status[r]_j = \perp$, then $E(v).round\text{-}status[r]_j \leftarrow \mathsf{green}$, since j is within distance $R_V/2$ of v. Thus, at the end of the last agreement phase, $E(v).round\text{-}status[r]_j \neq \perp$. □

Finally, we can show that if a virtual node broadcasts a message in round r, and if a nearby client does not detect a collision, then the client receives that message. The main part of the proof is an examination of the possible $\mathsf{vn\text{-}client\text{-}output}_{j,v}$ transitions; we use Lemma 11.12.4 to show that the only possible outcome consistent with our assumptions and the vn-client-output preconditions is that message m is delivered to j.

Lemma 11.12.6. *Assume that $v \in I_V$ is a virtual node, $j \in I_B$ is a client, and $r > 0$ is a virtual round. Assume that a round r $\mathsf{bcast}(m, \cdot)_{v,*}$, $m \neq \perp$, occurs in γ_v and a round r $\mathsf{recv}(M, \mathsf{null}, \ldots)_{j,*}$ event occurs in γ_j. If j is within distance R_V of $loc(v)_V$, then $m \in M$.*

Proof. We begin by noting that there is a round r $\mathsf{vn\text{-}client\text{-}output}_{j,v}$ event in α: by assumption, there is a round r $\mathsf{recv}_{j,*}$ event; a precondition of the $\mathsf{recv}_{j,*}$ event (in the multiplexer) is that $inVNs = V$, which implies that for all $v' \in I_V$, a round r $\mathsf{vn\text{-}client\text{-}output}_{j,v'}$ event occurs.

We divide the possible vn-client-output events into three categories, and in each case argue that $m \in M$. The first case addresses the situation where the vn-client-output event delivers a message from v to j (via the multiplexer). In this case it is relatively easy to see that m is the only possible message that could be delivered, and that the message delivered is included in the set M, as required.

The second case addresses the situation where the vn-client-output notifies j (via the multiplexer) of a collision resulting from a broadcast by virtual node v. We argue that this case cannot occur, since, by assumption, port $\langle j, * \rangle$ does not detect a collision in round r.

The third case addresses the situation where the vn-client-output event neither delivers a message to j nor notifies j of a collision. We argue that this case too is impossible: since there is a $\mathsf{bcast}_{v,*}$ event in γ_v, we can conclude (from Lemma 11.12.4) that either message m is in the round r ballot or $vnphaseBcast = \mathsf{true}$. In either case, however, the preconditions of vn-client-output preclude the third case from occurring.

We now proceed to examine the three cases in more detail:

- vn-client-output$(\langle \mathsf{vn}, m', loc(v) \rangle, \cdot)_{j,v}$: In this case, some message m' is delivered by the emulator for virtual node v to j. We claim that in this case, as per the operation of the multiplexer, $m' \in M$: message $\langle \mathsf{vn}, v, m' \rangle$ is added to $E(multiplexer).inM_j$ during the vn-client-output transition, and thus m' is delivered to port $\langle j, * \rangle$ during the (immediately) following $\mathsf{recv}_{j,*}$ transition. (See line 665, Figure 10-13.) If $m' = m$, this implies that $m \in M$, as desired.

  We argue that in fact m' must be equal to m in this case: Since m' is delivered to the client port $\langle j, * \rangle$ for node j, we can conclude by the integrity of the virtual broadcast service (Lemma 11.11.12) that some node $k \in I_B \cup I_V$ must have sent message m'. Moreover it is clear that in this case, if one follows the path constructed by Lemma 11.11.12 that identifies the sender of a message, virtual node v is the unique node that could have sent this message. (This is clear since all messages of the form $\langle \mathsf{vn}, v, \cdot \rangle$ originate at virtual node v.) We have, however, assumed that virtual node v broadcast message m in round r, resulting in the conclusion that $m' = m$.

- vn-client-output$(\perp, \pm)_{j,v}$: In this case, a collision is delivered by the emulator for virtual node v to j. We can conclude in this case that, as per the operation of the multiplexer, that a $\pm$ is delivered to port $\langle j, * \rangle$ during the round r $\mathsf{recv}_{j,*}$ event, since immediately after the vn-client-output transition, $E(multiplexer).inCD_j = \pm$ (line 653, Figure 10-13). This contradicts our assumption that there is a round r $\mathsf{recv}(M, \mathsf{null}, \cdot, \cdot)_{j,*}$ event in γ_j. Thus we conclude that this case does not occur.

- vn-client-output$(\perp, \mathsf{null})_{j,v}$: In this case, no message or collision is delivered from the emulator for virtual node v to j. We need to show that since there is a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event in γ_v, this case does not occur.

  We proceed to examine the preconditions for the vn-client-output transition (lines 201–207, Figure 10-4), and notice that there are only four cases that can result in a vn-client-output$(\perp, \mathsf{null})_{j,v}$, as we have assumed. We examine each of these cases in turn, and show that each implies a contradiction.

  The first case is one in which the round is designated as green, the virtual node is scheduled, $vnphaseBcast = \mathsf{false}$, and yet no message broadcast by virtual node v appears in the ballot; this is contradicted by Lemma 11.12.4, which

shows that if virtual node v broadcasts a message in round r, then the messages appears in the ballot or $vnphaseBcast = \mathsf{true}$.

The second and third cases are also ones in which the round is designated as green, but the virtual node is not scheduled. Moreover, again in these cases, $E(v).vnphaseBcast_j = \mathsf{false}$. Again, Lemma 11.12.4 implies a contradiction, indicating that since virtual node v broadcasts a message in round r, $E(v).vnphaseBcast_j = \mathsf{true}$.

The fourth case is one in which the round is not designated as green, but instead $E(v).round\text{-}status[r]_j = \perp$. In this case, the contradiction is implied by Lemma 11.12.5, as whenever j is near to v, we know that $E(v).round\text{-}status[r]_j \neq \perp$.

We now proceed in more detail to enumerate each of these cases:

- lines 187–188: $E(v).round\text{-}status[r]_j = \mathsf{green}$, v is scheduled for round r, and:

  $$\nexists \langle \mathsf{vn}, v, \cdot \rangle \in E(v).ballot[r].vnM_j \ .$$

  We can also conclude that $E(v).vnphaseBcast_j = \mathsf{false}$, as that case has already been eliminated by the earlier **else if** clause on line 183 (Figure 10-4. There are two subcases depending on whether j joins v in round r. First, assume that j does not join v in round r. (Note that j may or may not have already joined; we have not assumed, for example, that j is participating.) In this case, we can determine that $E(v).round\text{-}status[r]_j = \mathsf{green}$ at the end of the last agreement phase, as it is not modified between the end of the last agreement phase and the end of the virtual round. By Lemma 11.12.4, Part 1, we conclude that $E(v).vnphaseBcast_j = \mathsf{true}$, contradicting the assumptions of this case.

  If j does join v in round r, then there is some other node $j' \in I_B$ that broadcasts the join information to j in the join-ack phase. In particular, we can conclude that $E(v).round\text{-}status[r]_{j'} = \mathsf{green}$, as j adopts this state from the join information sent by j'. By Lemma 11.12.4 we can conclude that $E(v).vnphaseBcast_{j'} = \mathsf{true}$ at the end of the last agreement phase. Since j' sends the *vnphaseBcast* flag as part of the join information, we can conclude that $E(v).vnphaseBcast_j = \mathsf{true}$, again contradicting the assumption of this case.

- lines 194–195, lines 200–201: $E(v).round\text{-}status[r]_j \in \{\mathsf{green}, \perp\}$, v is not scheduled for round r, and $E(v).vnphaseBcast_j = \mathsf{false}$. There are two subcases depending on whether j joins v in round r. First, assume that j does not join v in round r. As in the previous case, we can thus determine that $E(v).round\text{-}status[r]_j \in \{\mathsf{green}, \perp\}$ at the end of the last agreement phase, and hence we apply Lemma 11.12.4 to conclude that $E(v).vnphaseBcast_j = \mathsf{true}$, contradicting the possibility that this case occurs.

  If j does join v in round r, then there is some other node $j' \in I_B$ that

broadcasts the join information to j in the join-ack phase. In particular, we can conclude that $E(v).\mathit{round\text{-}status}[r]_{j'} = \mathsf{green}$, as j adopts this state from the join information sent by j'. By Lemma 11.12.4 we conclude that $E(v).\mathit{vnphaseBcast}_{j'} = \mathsf{true}$ at the end of the last agreement phase. Since j' sends the ballot data structure as part of the join information, we can conclude that $E(v).\mathit{vnphaseBcast}_{j} = \mathsf{true}$, contradicting the possibility that this case occurs.

- lines 203–204: $E(v).\mathit{round\text{-}status}[r]_j = \perp$. By Lemma 11.12.5, we can conclude that this case does not occur since $E(v).\mathit{round\text{-}status}[r]_j \neq \perp$ when j is near to v.

Since these four sub-cases cannot occur, we conclude that this particular output event $\mathsf{vn\text{-}client\text{-}output}(\perp, \mathsf{null})_{j,v}$ cannot occur.

Thus we conclude that the first case in which message m is delivered by the vn-client-output is the only possible case, under the assumptions of this lemma, and hence $m \in M$, as desired. □

Delivering Messages to a Virtual Node

We next consider a particular virtual node $v' \in I_V$ that receives some set of messages M in virtual round r and does not detect a collision. Our goal is to show that every message broadcast by a nearby node—be it real or virtual—is delivered to v'. We first argue in Lemma 11.12.7 that if the sending node is virtual, then the sending node is scheduled for round r. We next prove in Lemma 11.12.8 a straightforward claim that if the round is not green, then the *scheduled-status* of a nearby emulator is also not green.

Lemma 11.12.7. *Let $v \in I_V$ and $v' \in I_V$ be two virtual nodes, $v \neq v'$, where $|loc(v)_V - loc(v')_v| < R_V$, and $r > 0$ be a virtual round. Assume that a round r $\mathsf{bcast}(m, \cdot)_v$ occurs in γ_v and a round r $\mathsf{recv}(\cdot, \mathsf{null}, \ldots)_{v',*}$ occurs in $\gamma_{v'}$. Then virtual node v is scheduled for round r.*

Proof. The proof consists of three basic steps, first we conclude via Lemma 11.12.1 that round r is not red for v', and we identify the round r ballot b for v'. The second part of the proof argues via Lemma 11.12.2 that some node $i \in I_B$ broadcasts a message $\langle \mathsf{vn}, v, m \rangle$ in the vn phase of round r, since there is a round r $\mathsf{bcast}(m, \cdot)_v$ event in γ_v. Finally, we put the pieces together to show that v must be scheduled in round r. We now proceed in more detail.

First, by Lemma 11.12.1, we conclude that round r is not red for v'. Let b be the round r ballot for v', and let j be the round r proposer. We conclude (still by Lemma 11.12.1) that $b.\mathit{vnCD} = \mathsf{null}$.

Next, assume for the sake of contradiction that v is not scheduled for round r. We conclude as per Lemma 11.12.2 that there exists some $i \in I_B$ that begins round r for v and remains joined and not failed through the beginning of the scheduled-ballot phase such that there is a round r $\mathsf{bcast}(\langle \mathsf{vn}, v, m \rangle, \cdot)_{i,v}$ event in the vn phase of round r in α.

At the beginning of the vn phase, nodes i and j are within distance $R_B = R_B/4 + R_B/4 + R_B/2$ of each other: node i is within distance $R_B/4$ of $loc(v)_V$, since $E(v).joined_i = \mathsf{true}$; node j is within distance $R_B/4$ of $loc(v')_V$ since j broadcasts a ballot for v' in round r, thus implying that $E(v').joined_j = \mathsf{true}$; v and v' are within distance $R_V = R_B/2$ of each other.

Thus, by the completeness of the basic broadcast service (Lemma 8.1.40), we conclude that node j either receives the message $\langle \mathsf{vn}, v, m \rangle$ from i, or detects a collision in the vn phase of round r.

In the first case, on line 270, node j sets $E(v').vnCD_= \pm$, since it discovers (on line 268) that virtual node v' is not scheduled (and $v' \neq v$, by assumption). In the second case, on line 266, node j detects a collision, and again on line 270 sets $E(v').vnCD_= \pm$.

In either case, when the ballot for v' is formed (on line 389), $b.vnCD \leftarrow \pm$, contradicting our previous claim that $b.vnCD = \mathsf{null}$. From this we conclude that virtual node v is scheduled in round r. □

The following claim, Lemma 11.12.8, is quite similar to the contrapositive of Lemma 11.5.6. The main difference is that Lemma 11.5.6 examines the emulator for virtual node v, and conclude that all the emulators for virtual node v have a certain round status—green or yellow—when the virtual round is green for v. In this case, however, we are interested in emulators for some virtual node $v' \neq v$. If an emulator for v' is close to v, and if round r is not good for v, we argue that the *scheduled-status* for v' is not green at the end of the scheduled-veto-2 phase.

Lemma 11.12.8. *Let $v, v' \in I_V$ be two virtual nodes, $r > 0$ a virtual round where v is scheduled for roundr, and $i \in I_B$ a node that is non-failed through the end of the* scheduled-veto-2 *phase. Moreover, assume that node i is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the* scheduled-veto-2 *phase of virtual round r. If round r is not good for v, then $E(v').scheduled\text{-}status_i \neq \mathsf{green}$ at the end of the* scheduled-veto-2 *phase of virtual round r.*

Proof. Choose some $j \in I_B$ that participates in virtual round r and has status $E(v).round\text{-}status[r]_j \in \{\mathsf{red}, \mathsf{orange}\}$ at the end of the scheduled-veto-2 phase of virtual round r. Since j has designates round r as red or orange, we argue that j broadcasts a veto message in the scheduled-veto-2 phase: Notice that since $E(v).round\text{-}status[r]_j$ is red or orange at the end of the scheduled-veto-2 phase, $E(v).scheduled\text{-}status_j$ must be red or orange at the beginning of the scheduled-veto-2 phase, as there is no later opportunity to downgrade the round to red or orange. However, if j has *scheduled-status* either red or orange by the end of the scheduled-veto-1 recv event, then node j broadcasts a veto message in the scheduled-veto-2 phase, since node j participates in the round and has *joined* set to true.

Since node j and node i are within distance R_B of each other, we conclude by the completeness of the basic broadcast service (Lemma 8.1.40) that node i either receives the veto message or detects a collision in the scheduled-veto-2 phase. In either case, node i sets $E(v').scheduled\text{-}status_i \neq \mathsf{green}$, concluding the proof. □

Finally, we can show that if a node—real or virtual—broadcasts a message in round r, and if a nearby virtual node does not detect a collision, then the virtual node receives that message. The proof breaks down into two cases, depending on whether the broadcasting node is real or virtual, and shows that in both cases the message arrives successfully.

Lemma 11.12.9. *Assume that $v' \in I_V$ is a virtual node, $j \in I_B \cup I_V$ is a client or a virtual node, and $r > 0$ is a virtual round. Assume that a round r $\mathsf{bcast}(m, \cdot)_{j,*}$, $m \neq \perp$, occurs in γ_j and a round r $\mathsf{recv}(M, \mathsf{null}, \ldots)_{v',*}$ event occurs in $\gamma_{v'}$. If j is within distance R_V of $loc(v')_V$, then $m \in M$.*

Proof. We begin by invoking Lemma 11.12.1 to conclude that round r is not red for v'. Moreover, we can determine that v is up in round r, as the construction of $\gamma_{v'}$ only inserts a $\mathsf{recv}_{v',*}$ event in round r if v' is up. It then follows from Corollary 11.5.16 that there is a unique round r proposer for v, which we label k, and a uniqure round r ballot b for v. Also by Lemma 11.12.1, we conclude that:

- $M = \{m : \langle \mathsf{client}, m, \cdot \rangle \in b.clientM$ **or** $\langle \mathsf{vn}, \cdot, m \rangle \in \cup b.vnM\}$
- $b.clientCD = \mathsf{null}$.
- $b.vnCD = \mathsf{null}$.

The proof now divides into two parts, depending on whether $j \in I_B$ or $j \in I_V$, i.e., whether j is a client or j is a virtual node.

Part 1: Assume $j \in I_B$. It follows from the construction of γ_j that there is also a $\mathsf{bcast}(m, \cdot)_{j,*}$ event in α, and we can conclude that this broadcast event occurs in the client phase of round r. By the operation of the multiplexer, port $\langle j, 0 \rangle$ broadcasts message $\langle \mathsf{client}, m, \cdot \rangle$ in the client phase of round r.

Notice that i is within distance R_B of k: i is within distance $R_B/2$ of $loc(v')_V$, by assumption, and k is within distance $R_B/4$ of $loc(v')_V$ (since a port only broadcasts a ballot if it is joined, and it only remains joined if it is near the virtual node).

We can thus conclude that at the end of the client phase, by the completeness of the basic broadcast service (Lemma 8.1.40), either $\langle \mathsf{client}, m, \cdot \rangle \in E(v').clientM_k$ or $E(v').clientCD_k = \pm$. Thus, either $m \in b.clientM$, or $b.clientCD = \pm$. We have already concluded that the latter case does not occur, and hence $m \in b.clientM$. As a result, $m \in M$, as per the conclusion of Lemma 11.12.1.

Part 2: Assume $j \in I_V$. Assume that for some $v \in I_V$ where v is within distance $R_B/2$ of $loc(v')_V$, there is a round r $\mathsf{bcast}(m', \cdot)_{v,*}$ event in γ_v where $m' \neq \perp$. (For the sake of consistent notation, we use v here instead of j, as in the lemma statement.)

We first conclude from Lemma 11.12.7 that v is scheduled for round r. (This implies that v' is not scheduled for round r, since v and v' are within distance R_V of each other and the schedule guarantees non-interference.)

Next, we argue that round r is good for v. Assume for the sake of contradiction that round r is not good for v. Then by (the contrapositive of) Lemma 11.12.8, we conclude that $E(v').scheduled\text{-}status_k \neq \mathsf{green}$ at the end of the scheduled-veto-2 phase, and hence $E(v').vnCD_k \leftarrow \pm$ on line 368 (Figure 10-7). Thus, when node k

forms the ballot for v' on line 389 (Figure 10-8), $b.vnCD = \pm$, which contradicts our claim above that $b.vnCD = \mathsf{null}$. Thus we conclude that round r is good for v.

We conclude by Lemma 11.12.2 that there exists some $i \in I_B$ that begins round r for v and remains joined and not failed through the beginning of the scheduled-ballot phase such that there is a round r $\mathsf{bcast}(\langle \mathsf{vn}, v, m\rangle, \cdot)_{i,v}$ event in round r of α.

We can conclude that i is within distance $R_B/4$ of $loc(v)_V$, since port $\langle i, v\rangle$ only broadcasts a message $\neq \perp$ in the vn phase if it is joined, and a node is only joined if it is within distance $R_B/4$ of v.

Since v is within distance $R_B/2$ of v', and k is within distance $R_B/4$ of v', we conclude that i is within distance R_B of k. Thus, by the completeness of the basic broadcast service (Lemma 8.1.40), we can conclude that either port $\langle k, v'\rangle$ receives the message $\langle \mathsf{vn}, v, m\rangle$ in the vn phase of round r, or port $\langle k, v'\rangle$ detects a collision. In the latter case, $E(v').vnCD_k \leftarrow \pm$, resulting in $b.vnCD = \pm$, which we have already shown is not the case.

We thus conclude that after line 257 (Figure 10-5), $\langle \mathsf{vn}, v, m\rangle \in E(v').nearby\text{-}msgs_k$. Since k is within distance $3R_B/4$ of v, we can determine that on lines 261–272 (Figure 10-5), either $E(v').vnM_k \leftarrow E(v').nearby\text{-}msgs_k$, or $E(v').vnCD_k \leftarrow \pm$. As we have already determined that the latter case does not occur, we can conclude that $\langle \mathsf{vn}, v, m\rangle \in E(v).vnM_k$ after line 272 (Figure 10-5), and similarly, as the end of the scheduled-veto-2 phase. Hence $\langle \mathsf{vn}, v, m\rangle \in b.vnM$, which implies that $m \in M$ as desired. □

Main Completeness Lemma

Finally, we conclude Section 11.12.2 by showing that $CD\text{-}rule_V$ is complete. Lemma 11.12.10 relates the definition of $CD\text{-}rule_V$ to the broadcast and receive events in the executions γ_i, $i \in I_B$, and γ_v, $v \in I_V$, and argues that if the rule is not complete, then there must exist some message broadcast in one of these executions that is not received by a nearby neighbor; moreover, the nearby neighbor does not detect a collision. The argument then proceeds to invokes Lemmas 11.12.3, 11.12.6 and 11.12.9 to show that in each case the message is, in fact, delivered, from which we conclude that the collision detector rule is complete.

Lemma 11.12.10. *The collision detector rule $CD\text{-}rule_V$ is complete.*

Proof. We have to show that for every broadcast port $j \in bcast\text{-}ports_V$, for every round $r > 0$, for every $p, q \in \mathbb{N}_0$, if $p > q$, then $CD\text{-}rule(j, r, p, q, \cdot, \cdot)_V = \pm$.

Recall the rule for constructing $CD\text{-}rule_V$: Consider the case where a round r $\mathsf{recv}(M, cd, \cdot, \cdot)_{j,*}$ event occurs in γ_j, for some $M \subseteq msgs$ with the following properties:

1. $p = |sentM(j, r)|$,
2. $q = |M \cap sentM(j, r)|$,
3. $p' = |sentMinterfere(j, r)|$, and
4. $q' = |M \cap sentMinterfere(j, r)|$.

Then the construction specifies that $\textit{CD-rule}(r, p, q, p', q')_V = cd$. If $p > q$, we must show that $cd = \pm$.

In all other cases, i.e., for all $j \in I_B$, $p, p', q, q' \in \mathbb{N}_0$ where the round r recv event does not satisfy Properties 1–4, the desired conclusion follows immediately from the definition of $\textit{CD-rule}_V$. For the rest of this proof, fix $j \in I_B$, $r \in \mathbb{N}$, $p, p', q, q' \in \mathbb{N}_0$, where $p > q$, such that the round r $\mathsf{recv}(M, cd, \ldots)_{j,*}$ in γ_j satisfies Properties 1–4. Assume for the sake of contradiction that $cd = \mathsf{null}$.

We begin by choosing a message $m \neq \perp$ that will be of interest: since $p > q$, we can fix some $m \in \textit{sentM}(j, r)$ such that $m \notin M \cap \textit{sentM}(j, r)$, i.e., $m \notin M$. Every message in $\textit{sentM}(j, r)$ originates (by definition) with either a $\mathsf{bcast}_{i,*}$ event, for $i \in I_B$, or a $\mathsf{bcast}_{v,*}$ event, for $v \in I_V$, where the sending node is within distance R_V of node j at the beginning of round r. Thus we can identify some node $i \in I_B$ or $v \in I_V$ that broadcasts message m.

We proceed to consider the four cases separately, depending on whether j is a client or a virtual node, and depending on whether the sender of m is a client or a virtual node: (1) $i \in I_B$ and $j \in I_B$ are both clients; (2) $v \in I_V$ is a virtual node; $j \in I_B$ is a client; (3) $i \in I_B$ is a client, $j \in I_V$ is a virtual node; (4) $v \in I_B$ is a virtual node, $j \in I_V$ is a virtual node.

1. There is a $\mathsf{bcast}(m, \cdot)_{i,*}$ event in round r in γ_i for some $i \in I_B$, and a $\mathsf{recv}(M, \mathsf{null}, \ldots)_{j,*}$ in round r for some $j \in I_B$: We invoke Lemma 11.12.3 to conclude that since j is within distance R_V of i, $m \in M$. This, however, contradicts our assumption that $m \notin M$.

2. There is a $\mathsf{bcast}(m, \cdot)_{v,*}$ event in γ_v for some $v \in I_V$, and a $\mathsf{recv}(M, \mathsf{null}, \ldots)_{j,*}$ in round r for $j \in I_B$: We invoke Lemma 11.12.6 to conclude that since j is within distance R_V of $\textit{loc}(v)_V$, $m \in M$. This, however, contradicts our assumption that $m \notin M$.

3. There is a $\mathsf{bcast}(m, \cdot)_{i,*}$ event in round r in γ_i for some $i \in I_B$, and a $\mathsf{recv}(M, \mathsf{null}, \ldots)_{v',*}$ in round r for some $v' \in I_V$: We invoke Lemma 11.12.9 to conclude that since j is within distance R_V of $\textit{loc}(v')_V$, $m \in M$. This, however, contradicts our choice of m above.

4. There is a $\mathsf{bcast}(m, \cdot)_{v,*}$ event in round r in γ_v for some $v \in I_V$, and a $\mathsf{recv}(M, \mathsf{null}, \ldots)_{v',*}$ in round r for some $v' \in I_V$: We invoke Lemma 11.12.9 to conclude that since $\textit{loc}(v)_V$ is within distance R_V of $\textit{loc}(v')_V$, $m \in M$. This, however, contradicts our choice of m above.

□

11.12.3 Eventual Accuracy

The goal of this section is to show that the $\textit{CD-rule}_V$ defined in Section 11.12 is eventually accurate (as per Definition 8.1.14). As in Section 11.12.2, the proof involves examining broadcast and receive events in executions γ_i, for $i \in I_B$, and γ_V, for $v \in I_V$, and showing that they imply that the collision detector rule is eventually accurate.

Specifically, we need to show that from some virtual round onwards, if some non-failed node $i \in I_B \cup I_V$ detects a collision in a virtual round r, then some nearby node $j \in I_B \cup I_V$ broadcasts a message m in round r that is not received by i. By the manner in which the collision detector rule is constructed, this is sufficient to show eventual accuracy. In fact, we show that the collision detector rule stabilizes in round $r_{gst} + 1$.

Preliminary Lemmas

We begin with some preliminary lemmas. The key difficulty in proving eventual accuracy is showing that when $\pm$ is recorded by the $ballot[r]$ data structure in some round $r \in I_B$ (for a round r large enough), then in fact some message was broadcast by a nearby node and not received. The ballot data structure records collisions in two subfields: the *clientCD* field, for collisions that originate from messages sent in the client phase, and the *vnCD* field, for collisions that originate from messages that originate in the vn phase. In each case, we trace the collision back until we find the message that was lost. Lemma 11.12.11 considers the case where a message is broadcast in the client phase; Lemma 11.12.12 considers the case where a message is broadcast in the vn phase.

Our first claim, Lemma 11.12.11 shows that if a message $\langle \mathsf{client}, m, \cdot \rangle$ is broadcast in the client phase of round r by some $i \in I_B$, then message m is broadcast in round r of γ_i. The lemma follows almost immediately from the construction of γ_i and the operation of the multiplexer:

Lemma 11.12.11. *Let $i \in I_B$ be a node, and assume that α contains a $\mathsf{bcast}(\langle \mathsf{client}, m, \cdot \rangle, \cdot)_{i,0}$ event, $m \neq \perp$, in the client phase of round r. Then, there is a round r $\mathsf{bcast}(m, \cdot)_v$ event in γ_i.*

Proof. The multiplexer port $\langle i, 0 \rangle$ broadcasts a message $\langle \mathsf{client}, m, \cdot \rangle$ in the client phase of round r only if $E(multiplexer).outM_i = m$. However, $E(multiplexer).outM_i = m$, $m \neq \perp$, only if there is a preceding $\mathsf{bcast}(m, \cdot)_{i,*}$ event. It remains only to argue that this preceding $\mathsf{bcast}_{i,*}$ event occurs in the client phase of round r. By the use of the *clientBcast* flag and the *stops when* condition limiting the trajectories of the multiplexer, we know that the $\mathsf{bcast}_{i,0}$ event immediately follows the $bcast_{i,*}$ event with no intervening time passage: the $bcast_{i,0}$ transition is the only one which resets the *clientBcast* flag. Thus we conclude that the preceding $\mathsf{bcast}_{i,*}$ event occurs in the client phase of round r. Since γ_i contains every client phase $\mathsf{bcast}_{i,*}$ event, we thus conclude that there is a round r $\mathsf{bcast}(m, \cdot)_{i,*}$ event in γ_i. □

Our next claim, Lemma 11.12.12, proves a similar claim as Lemma 11.12.11, except with respect to messages broadcast in the vn phase, rather than the client phase. Specifically, we show that if some message $\langle \mathsf{vn}, v, m \rangle$ is broadcast in the vn phase in a round $r \geq r_{gst} + 1$, then for some virtual node $v \in I_V$, message m is broadcast in round r in execution γ_v. That is, every message broadcast in the vn phase is also broadcast in the execution of the virtual system. (Notice that now we are considering round $r_{gst} + 1$, rather than round r_{gst}; the emulator takes one round to stabilize.)

This lemma is almost the converse of Lemma 11.12.2, which states that if a messages m is broadcast in round r of γ_v, then message $\langle \mathsf{vn}, v, m\rangle$ is broadcast in the vn phase of round r. The main difference is that Lemma 11.12.2 holds from the beginning of the execution, while Lemma 11.12.12 holds only for rounds $r \geq r_{gst} + 1$. (There are also some differences in the precise conditions under which the respective lemmas hold; for example, Lemma 11.12.2 assumes that if v is scheduled, then round r is good for v.) In many ways, Lemma 11.12.12 is the key lemma necessary to prove the eventual accuracy of the collision detector rule.

Lemma 11.12.12. *Let $r \geq r_{gst} + 1$ be a virtual round, $i \in I_B$ be a node, and $v \in I_V$ be a virtual node. Assume that α contains a $\mathsf{bcast}(\langle \mathsf{vn}, v, m\rangle, \cdot)_{i,v}$ event, $m \neq \perp$, in the vn phase of round r. Then, there is a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event in γ_v.*

Proof. The proof proceeds in three steps. First, we identify an epoch $\langle r_1, r_2\rangle$, where $r_1 < r \leq r_2 + 1$, such that the (sole) execution $\gamma' \in \mathit{execs}(r_1, r_2)_v$ is used in the construction of γ_v. (By Lemma 11.8.12 we know that round r_2 is green, and hence we can conclude by Corollary 11.7.12 that there is only one execution in the set.) Second, we identify an execution $\gamma < \gamma'$ which includes round $r - 1$, but not round r. Since there is also only one execution in the set $\mathit{execs}(r_1, r - 1)_v$ (for the same reasons as there is only one execution in $\mathit{execs}(r_1, r_2)_v$), and since there is a message $\langle \mathsf{vn}, v, m\rangle$ broadcast in the vn phase of round r, we can conclude that in the last state of γ, virtual node v is enabled to broadcast message m. Finally, we conclude the proof by arguing that this broadcast event occurs in γ', and hence also in γ_v.

First, by Lemma 11.8.12, we conclude that round r is green, and by Corollary 11.5.7, that round r is good. Choose r_1, r_2 such that $\langle r_1, r_2\rangle \in \mathit{updown}(v)$ delineates an epoch of v and $r \in [r_1, r_2 + 1]$. This is possible since v is up in round $r-1$—otherwise no broadcast occurs in round r in the vn phase—and since $r \geq r_{gst}+1$, we can conclude that round $r-1$ is good, which implies that $r-1$ is included in some epoch of the updown sequence, since the updown sequence contains maximal epochs. If $r_2 = \infty$, then set $r_2 = r$ so that $\langle r_1, r_2\rangle$ is a finite epoch of v.

The next fact to notice is that $r > r_1$: if $r = r_1$, then $E(v).\mathit{last\text{-}reset}_i = r_1 - 1$ at the beginning of the client phase of round r, according to Corollary 11.4.5; thus, $E(v).\mathit{rnd}_i = E(v).\mathit{last\text{-}reset}_i + 1$, and $E(v).\mathit{outgoing\text{-}msg}_i$ is unmodified in line 240, resuling in the broadcast of $\perp$ in the vn phase, contradicting our hypothesis that some message $\langle \mathsf{vn}, v, m\rangle$ is broadcast in the vn phase. Thus we conclude that $r > r_1$.

Next, note that by Lemma 11.8.12, both rounds $r-1$ and r_2 are both green, since v is up in both of these rounds. By Corollary 11.7.12, $|\mathit{execs}(r_1, r - 1)_v| = 1$ and $|\mathit{execs}(r_1, r_2)_v| = 1$. Fix $\gamma \in \mathit{execs}(r_1, r - 1)_v$, and $\gamma' \in \mathit{execs}(r_1, r_2)_v$. We invoke Lemma 11.7.20 to conclude that γ' is an extension of γ: the lemma states that γ' in an extension of some execution in $\mathit{execs}(r_1, r - 1)_v$; γ is the only one.

Next, notice that according to the construction of γ_v, γ' is included in γ_v. In the case where $\langle r_1, r_2\rangle \in \mathit{updown}(v)$ (i.e., the case where the epoch containing r is finite), this is immediate by construction. In the case where $\langle r_1, r_2\rangle \notin \mathit{updown}(v)$, this follows from the manner in which γ_v is constructed by successive extension.

Finally, we argue that the γ' extension of γ contains a round r $\mathsf{bcast}(m, \cdot)_v$ event. Since node i broadcasts a message $\langle \mathsf{vn}, v, m\rangle$ in the vn phase of round r, we conclude

that $E(v).outgoing\text{-}msg_i = \langle \mathsf{vn}, v, m \rangle$ at the end of the client phase of round r. The message m is constructed in lines 235–237 by calculating the last state s of $\gamma = exec(r_1, r-1, i)_v$, and then calling the function do-bcast(s).

Recall that γ' extends γ be first calling do-bcast on s, the last state in γ (see Figure 10-11). Thus we conclude that γ' contains a round r $\mathsf{bcast}(m, \cdot)_v$ event, from which we conclude that γ_v contains a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event, as desired. □

In the previous two lemmas, we argued that a broadcast in the client or vn phases indicates a similar broadcast in γ_i, $i \in I_B$, or γ_v, $v \in I_V$ (respectively). The next step in tracing the cause of the collision is to show that a collision in the *clientCD* or *vnCD* fields of the ballot indicate that some message was in fact broadcast in the client or vn phases, and that this message was lost. When combined with the two previous lemmas, this indicates that there was a message broadcast in γ_i, $i \in I_B$ or in γ_v, $v \in I_V$, that was lost, hence justifying the collision.

More specifically, in Lemma 11.12.14, we assume that there is a ballot b broadcast for some virtual node $v \in I_V$ in one of the two ballot phases of round r, and that this ballot indicates a collision in the client phase, i.e., $b.clientCD = \pm$. We conclude that in this case, there is some message m that was broadcast by $j \in I_B$ in round r of execution γ_j. Moreover, this message m is not included in the ballot, i.e., $\langle \mathsf{client}, m, \cdot \rangle \notin b.clientM$. Thus, messages m was broadcast in round r and not received. Finally, we note that j is within the interference range of node v, and hence v's collision detector is behaving accurately when it detects a collision in round r.

Lemma 11.12.13. *Let $v \in I_V$ be a virtual node, and $r \geq r_{gst} + 1$ a virtual round in which some node $i \in I_B$ broadcasts a ballot b in a ballot phase, either the* scheduled-ballot *phase of the* unscheduled-ballot *phase.*

Assume that ballot $E(v).b.clientCD_i = \pm$ immediately after either (1) line 275 in the vn *phase of round r, or (2) line 389 in the* unscheduled-ballot *phase of round r, i.e., immediately after the ballot is created.*

Then there exists some $m \in msgs_V$ and some $j \in I_B$ where j is within distance R'_V of v at the beginning of round r, such that a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ event occurs in γ_j and $\langle \mathsf{client}, m, \cdot \rangle \notin E(v).b.clientM_i$ immediately after the ballot b is set.

Proof. The proof consists of three steps. First, we identify a $\mathsf{recv}(\cdot, cd, \ldots)_{i,v}$ event where $cd = \pm$ in the client phase of round r; this is the event which results in the collision indication in the ballot. Second, we conclude that as a result if the eventual accuracy of the basic broadcast service, there is some $\mathsf{bcast}_{j,0}$ event that broadcasts a message $\langle \mathsf{client}, m, \cdot \rangle$ in the client phase that is lost. Finally, we apply Lemma 11.12.11 to conclude that message m is broadcast in round r of γ_j, completing the proof.

First, notice that ballot b is formed from the $E(v).clientM_i$ variable (as well as *clientCD*, *vnM*, and *vnCD*). Moreover, prior to the ballot being formed, the *clientCD* variable is modified only in the client phase of virtual round r, specifically, on line 228 (Figure 10-5), $E(v).clientCD_i \leftarrow cd$. Thus we conclude in this case that $cd = \pm$ in the client phase $\mathsf{recv}(\cdot, cd, \ldots)_{i,v}$ for round r.

Second, by the eventual accuracy of the basic broadcast service (Lemma 8.1.42), we can conclude that if $cd = \pm$, then there is some message sent in the client phase by

a nearby port that was not delivered. That is, there exists some port $p \in \mathit{bcast\text{-}ports}_B$ such that p is within distance R'_B of i at the beginning of the client phase (and hence, the beginning of virtual round r) and a round r $\mathsf{bcast}(m', \cdot)_p$ event occurs in α. Moreover, since message m' never arrives at port $\langle i, v\rangle$, it is straightforward to see that message m' is never added to $E(v).\mathit{clientM}_i$, and hence $m' \notin b.\mathit{clientM}$.

Notice, however, that for all $j \in I_B$, for all $v \in I_V$, port $\langle j, v\rangle$ does not broadcast in the client phase of round r. Thus we conclude that port p is a multiplexer port, i.e., for some $j \in I_B$, port $p = \langle j, 0\rangle$; fix this node $j \in I_B$. Thus j is within distance R'_B of i. Since i is within distance $R_B/4$ of $\mathit{loc}(v)_V$ (as it must be joined to broadcast a ballot, we conclude that j is within distance $R'_V \geq R'_B + R_B/4$ of $\mathit{loc}(v)_V$.

We can also conclude that $m' = \langle \mathsf{client}, m, \cdot\rangle$, for some $m \in \mathit{msgs}_V$, as per line 694 (Figure 10-13), since the multiplexer only broadcasts messages of this form. (We have already concluded that $m' \neq \perp$, as the eventual accuracy property indicates that a non-$\perp$ message is broadcast.)

Finally, by Lemma 11.12.11, we conclude that there is a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ event in γ_j. Since $\langle \mathsf{client}, m, \cdot\rangle \notin b.\mathit{clientM}$, this concludes our proof. □

In Lemma 11.12.14 we prove the equivalent result for collision detected in the vn phase. More specifically, we assume that there is a ballot b broadcast for some virtual node $v \in I_V$ in one of the two ballot phases of round r, and that this ballot indicates a collision in the vn phase, i.e., $b.\mathit{vnCD} = \pm$. We conclude that in this case, there is some message m that was broadcast by $v' \in I_V$ in round r of execution $\gamma_{v'}$. Moreover, this message m is not included in the ballot, i.e., $\langle \mathsf{vn}, v', m\rangle \notin b.\mathit{vnCD}$. Thus, messages m was broadcast in round r and not received. Finally, we note that v' is within the interference range of node v, and hence v's collision detector is behaving accurately when it detects a collision in round r.

Lemma 11.12.14. *Let $v \in I_V$ be a virtual node, and $r \geq r_{gst} + 1$ a virtual round in which some node $i \in I_B$ broadcasts a ballot b in a ballot phase, either the* scheduled-ballot *phase of the* unscheduled-ballot *phase.*

Assume that ballot $E(v).b.\mathit{vnCD}_i = \pm$ immediately after either (1) line 275 in the vn *phase of round r, or (2) line 389 in the* unscheduled-ballot *phase of round r, i.e., immediately after the ballot is created.*

Then for some message $m \in \mathit{msgs}_V$, for some $v' \in I_V$, $v' \neq v$, where v' is within distance R'_V of v (at the beginning of round r[1]), we conclude that there is a round r $\mathsf{bcast}(m, \cdot)_{v',*}$ event in $\gamma_{v'}$ and that $\langle \mathsf{vn}, v', m\rangle \notin E(v).b.\mathit{vnM}_i$ immediately after the ballot b is set.

Proof. The ballot is created in either line 275 or line 389 from $E(v).\mathit{vnM}_i$ and $E(v).\mathit{vnCD}_i$ (along with $E(v).\mathit{clientM}_i$, $E(v).\mathit{clientCD}_i$, and $E(v).\mathit{prev\text{-}rnd}_i$). We trace back the point at which $E(v).\mathit{vnCD}_i$ was set to $\pm$, and identify a message that was lost.

At some point prior to the ballot being formed we know that $E(v).\mathit{vnCD}_i$ is set to $\pm$, by assumption. The collision flag $E(v).\mathit{vnCD}_i$ are first initialized during round r in

[1] *Recall that virtual nodes are not mobile.*

the vn phase recv transition. The collision flag $E(v).vnCD_i$ is then potentially modified, i.e., set to $\pm$, once in the vn phase—line 270—and once in the scheduled-veto-2 phase—line 368. We examine the possible points at which this could have occurred, and show that it implies that some message m was broadcast in the vn phase that is not included in vnM. We then invoke Lemma 11.12.12 to show that m was also broadcast in $\gamma_{v'}$, for some $v' \in I_V$ that is within the interference range of v, concluding the proof.

1. The first case where $E(v).vnCD_i \leftarrow \pm$ is on line 270 (Figure 10-5) under three conditions: (1) port $\langle i, v\rangle$ received more than one message from a nearby virtual node, (2) port $\langle i, v\rangle$ received a message from an unscheduled virtual node, or (3) port $\langle i, v\rangle$ detected a collision. In each of these cases, the set of messages $E(v).vnM_i$ remains empty (as initialized on line 258), and hence when the ballot b is formed, $b.vnM = \emptyset$, and no messages are delivered to v from a node $v' \neq v$ in round r. It remains to show that some message m was broadcast in $\gamma_{v'}$ for some $v' \neq v$ near to v. We consider separately the three sub-cases that trigger the execution of line 270:

   - line 264: Immediately prior to line 270, $|E(v).\textit{nearby-msgs}_i| > 1$. Notice, however, that there is at most one message of the form $\langle \mathsf{vn}, v, \cdot\rangle$ in $E(v).\textit{nearby-msgs}_i$ since round $r \geq r_{gst}$, and hence at most one joined, non-failed node has $E(v).roundCM\_\mathsf{active}$ and broadcasts a message in the vn phase. Thus, there must be some message $\langle \mathsf{vn}, v', m\rangle$ in $E(v).\textit{nearby-msgs}_i$ where $v' \neq v$. We can determine that the message was sent by some port $\langle k, v'\rangle$ in the vn phase, for some $k \in I_B$ (by the integrity of the basic broadcast service). We then invoke Lemma 11.12.12 to conclude that there is a $\mathsf{bcast}(m, \cdot)_{v',*}$ event in $\gamma_{v'}$. As argued above, $\langle \mathsf{vn}, v', m\rangle \notin b.vnM$, as $b.vnM = \emptyset$.
   - line 266: In this case, $cd = \pm$. By the eventual accuracy of the basic broadcast service (Lemma 8.1.42), since $cd = \pm$ in the $\mathsf{recv}(\cdot, c, \ldots)_{i,v}$ transition in the vn phase of round r, there is some message m' that was sent by a port $\langle j, v'\rangle$, $j \in I_B$, $v' \in I_V$, in the vn phase, where j is within distance R'_B of i, and message m' is not received by i in the vn phase of round r. Since j is within distance R'_B of i, and i and j are within distance $R_B/4$ of v and v' (respectively), we can conclude that v is within distance $R'_B + R_B/2 = R'_V$ of v'.

     All the messages sent in the vn phase are of the form $\langle \mathsf{vn}, v', \cdot\rangle$; therefore, we conclude that for some $m \in \textit{msgs}_V$ there is a $\mathsf{bcast}(\langle \mathsf{vn}, v', m\rangle, \cdot)_{j,v'}$ event in the vn phase of round r.

     By Lemma 11.12.12, we conclude that there is a $\mathsf{bcast}(m, \cdot)_{v',*}$ event in $\gamma_{v'}$. Since $\langle \mathsf{vn}, v', m\rangle$ is not received by i in the vn phase, message $\langle \mathsf{vn}, v', m\rangle \notin E(v).\textit{nearby-msgs}_i$, and hence is not included in the ballot $b.vnM$.

     It remains only to show that $v \neq v'$. If $v = v'$, then we argue that $i = j$, which implies by the self-delivery property of the basic broadcast service that j receives message $m' = \langle \mathsf{vn}, v', m\rangle$ in the vn phase

of round r, contradicting our choice of message m'. Thus it suffices to show that if $v = v'$, then $i = j$. Recall that node i was selected because it broadcasts ballot b in a ballot phase. Thus we conclude that $E(v').roundCM_i = E(v).roundCM_i = \mathsf{active}$ at the end of the client phase (since *roundCM* is modified only in the client phase of round r). Moreover, since port $\langle j, v\rangle$ broadcasts a message in the vn phase, we can conclude that $E(v).roundCM_j = \mathsf{active}$ at the end of the client phase. However, Lemma 11.8.3 shows that there is at most one node $i \in I_B$ that has $E(v).roundCM_i = \mathsf{active}$ at the end of the client phase of round r, and thus $i = j$, resulting in the desired contradiction. We conclude that $v \neq v'$.

- line 268: In this case, there exists a message $\langle \mathsf{vn}, v', m\rangle$ from some unscheduled virtual node v' in *nearby-msgs*, where $v' \neq v$. By the definition of the set *nearby-msgs*, we know that v' is within distance R_V of v. We can determine that the message was sent by some port $\langle k, v'\rangle$ in the vn phase, for some $k \in I_B$ (by the integrity of the basic broadcast service). We then invoke Lemma 11.12.12 to conclude that there is a $\mathsf{bcast}(m, \cdot)_{v',*}$ event in $\gamma_{v'}$. As argued above, $\langle \mathsf{vn}, v', m\rangle \notin b.vnM$, as $b.vnM = \emptyset$.

2. The second case where $E(v).vnCD_i \leftarrow \pm$ is on line 368 (Figure 10-7) in the scheduled-veto-2 phase. In this case, $E(v).scheduled\text{-}status_i \neq \perp$ and $E(v).scheduled_i = \mathsf{false}$ (due to the **if/else** condition lines 358 and 366, Figure 10-7). Additionally, we can determine that i does not fail prior to the end of the scheduled-veto-2 phase (or it would not reach the line in question). However, according to Lemma 11.8.8, $E(v).scheduled\text{-}status_i = \perp$, implying that this case does not occur.

Thus we conclude that there exists some $m \in msgs_V$, $m \neq \perp$, and some $v' \in I_V$, $v' \neq v$, where v' is within distance R'_V of v such that a round r $\mathsf{bcast}(m, \cdot)_{v',*}$ event occurs in $\gamma_{v'}$ and $\langle \mathsf{vn}, v', m\rangle \notin E(v).b.vnM_i$ immediately after the ballot b is set. □

The previous four lemmas address the case where a collision is detected in a ballot for some virtual node. Thus, these lemmas are used to show that if a virtual node $v \in I_V$ detects a collision in round r of γ_v, then some message is lost. The final preliminary lemma is related to the case where a client $j \in I_B$ detects a collision in round r of γ_i. We show that if a vn-client-output delivers the collision notification to i (via the multiplexer), then some message broadcast by a virtual node $v \in I_V$ is lost.

Lemma 11.12.15. *Let $v \in I_V$ be a virtual node, $j \in I_B$ a client, and $r \geq r_{gst} + 1$ a virtual round. Assume a $\mathsf{vn\text{-}client\text{-}output}(m, \pm)_{j,v}$ occurs at the end of virtual round r in α. Then $m = \perp$ and for some $v' \in I_V$, where j is within distance R'_V of $loc(v')_V$, there exists a round r $\mathsf{bcast}(m', \cdot)_{v',*}$ event in $\gamma_{v'}$, where $m \neq \perp$.*

Proof. The proof proceeds as follows: Since $r \geq r_{gst} + 1$, we can conclude that $E(v).round\text{-}status[r]_j = \mathsf{green}$ and $E(v).round\text{-}status[r\text{-}1]_j \in \{\mathsf{green}, \perp\}$. By examining the preconditions of vn-client-output we conclude that $m = \perp$ and that $E(v).vnphaseBcast_j = \mathsf{true}$. This latter conclusion provides the key evidence needed

to show that some message was broadcast in the vn phase, which by Lemma 11.12.12 proves the desired result.

First, we can conclude immediately that $m = \perp$ by examination of the preconditions for $\mathsf{vn\text{-}client\text{-}output}_{j,v}$ (lines 177–207, Figure 10-4): the preconditions consist of a series of cases, each of which determines *vnm-out* and *vnCD-out*, the message and collision parameters of the vn-client-output transition; in every case in which $vnCD\text{-}out = \pm$, notice that $vnm\text{-}out = \mathsf{null}$. For example, on line 192, $vnCD\text{-}out = \pm$; in the preceding line 191, $vnm\text{-}out = \mathsf{null}$.

Next, notice that $E(v).round\text{-}status[r]_j \in \{\mathsf{green}, \perp\}$ immediately prior to the vn-client-output: if v is scheduled, then this follows from Lemma 11.8.8; if v is unscheduled, then this follows from Lemma 11.8.11. By the same argument, we conclude that $E(v).round\text{-}status[r\text{-}1]_j \in \{\mathsf{green}, \perp\}$. Moreover, it is clear that $E(v).round\text{-}status[r]_j \neq \perp$, since in that case the preconditions require that $vnCD\text{-}out = \mathsf{null}$, contradicting our assumption that $\langle j, v\rangle$ delivers a collision to j.

Next, we conclude that $E(v).vnphaseBcast_j = \mathsf{true}$: since $round\text{-}status[r\text{-}1]j \in \{\mathsf{green}, \perp\}$, we can conclude that the execution never reaches line 193 (Figure 10-4), which would result in $vnCD\text{-}out = \mathsf{null}$. Thus we are either in the case where v is scheduled—lines 183–185—or the case where v is unscheduled—lines 190–192.

Since $E(v).vnphaseBcast_j = \mathsf{true}$, we can conclude that some message $\langle \mathsf{vn}, v, m\rangle$ was sent in the vn phase of round r, which implies the desired result by Lemma 11.12.12, since the flag *vnphaseBcast* is set in two cases (line 262, Figure 10-5):

1. Some message $\langle \mathsf{vn}, v, m'\rangle$ is in the set *nearby-msgs* in the vn phase: We can conclude that j must be within distance $3R_B/4 \leq R'_V$ of $loc(v)_V$ since $E(v).vnphaseBcast_i$ is only set after the distance check on line 261 (Figure 10-5). By the integrity of the basic broadcast service we can conclude that for some node $k \in I_B$, port $\langle k, v\rangle$ broadcast message $\langle \mathsf{vn}, v, m'\rangle$ in the vn phase of round r. We thus conclude by Lemma 11.12.12 that a round r $\mathsf{bcast}(m', \cdot)_{v,*}$ event occurs in $gamma_v$.

2. A collision is detected in the vn phase of round r: We conclude by the eventual accuracy of the basic broadcast service) that for some $k \in I_B$, where k is within distance R'_B of j, a message $\langle \mathsf{vn}, v', m'\rangle$ was sent in the vn phase of round r by port $\langle k, v'\rangle$. Since k is within distance R'_B of j, and k is also within distance $R_B/4$ of $loc(v')_V$ (as k sends a message in the vn phase only if it is joined), we conclude that j is within distance $R'_B + R_B/4 \leq R'_V$ of $loc(v')_V$. We conclude by Lemma 11.12.12 that a round r $\mathsf{bcast}(m', \cdot)_{v',*}$ event occurs in $gamma_{v'}$.

□

Main Eventual Accuracy Result

We can now show the main result of this section: the virtual collision detector rule $CD\text{-}rule_V$ is eventually accurate. Lemma 11.12.16 first relates the definition of $CD\text{-}rule_V$ to the broadcast and receive events in the executions γ_i, $i \in I_B$, and γ_v, $v \in I_V$, and argues that if the rule is not eventually accurate and a collision is

detected in round r, then there must exist some message broadcast in one of these executions that is not received by a nearby neighbor.

The argument then divides into two cases, depending on whether the receiving node is a client or a virtual node. If the receiving node is a client, the argument relies on Lemma 11.12.15 to show that some messages is lost; if the receiving node is a virtual node, the argument relies on Lemmas 11.12.13 and 11.12.14 to show that some message is lost.

Lemma 11.12.16. *The collision detector rule CD-rule$_V$ is eventually accurate, stabilizing in round $r_{gst} + 1$.*

Proof. We have to show that for every broadcast port $j \in \mathit{bcast\text{-}ports}_V$, for every round $r > r_{gst} + 1$, for every $p', q' \in \mathbb{N}_0$, if $p' \leq q'$, then $\mathit{CD\text{-}rule}(j, r, \cdot, \cdot, p', q')_V = \mathsf{null}$.

Recall the rule for constructing $\mathit{CD\text{-}rule}_V$: Consider the case where a round r $\mathsf{recv}(M, cd, \cdot, \cdot)_{j,*}$ event occurs in γ_j, for some $M \subseteq \mathit{msgs}$ with the following properties:

1. $p = |\mathit{sentM}(j, r)|$,
2. $q = |M \cap \mathit{sentM}(j, r)|$,
3. $p' = |\mathit{sentMinterfere}(j, r)|$, and
4. $q' = |M \cap \mathit{sentMinterfere}(j, r)|$.

In this case, the construction specifies that $\mathit{CD\text{-}rule}(r, p, q, p', q')_V = cd$, i.e., the rule depends on whether the recv event delivers a collision to j. In all other cases, i.e., for all $j \in I_B$, $p, p', q, q' \in \mathbb{N}_0$ where the round r recv event does not satisfy Properties 1–4, $\mathit{CD\text{-}rule}(r, p, q, p', q')_V$ is set to a default value that satisfies the accuracy property: if $p' \leq q'$, then $\mathit{CD\text{-}rule}(r, p, q, p', q')_V = \mathsf{null}$.

Thus, we must show that if a round r recv event occurs satisfying Properties 1–4 and if $p' \leq q'$, then $cd = \mathsf{null}$. In all other cases, i.e., where no such round r recv event occurs, the desired conclusion follows immediately from the definition of $\mathit{CD\text{-}rule}_V$. For the rest of this proof, fix $j \in I_B$, $r \in \mathbb{N}$, $p, p', q, q' \in \mathbb{N}_0$, where $p' \leq q'$, such that the round r $\mathsf{recv}(M, cd, \ldots)_{j,*}$ in γ_j satisfies Properties 1–4. Assume for the sake of contradiction that $cd = \pm$. Our main goal is to show that there exists some message $m \in \mathit{sentM}(j, r)'$ that is not in M, i.e., $p' > q'$, resulting in a contradiction. (Notice that by definition $p' \geq q'$, since $M \cap \mathit{sentM}(j, r)'$ is no larger than $\mathit{sentM}(j, r)'$; hence for the purposes of constructing $\mathit{CD\text{-}rule}_V$, the assumption that $p' \leq q'$ is equivalent to assuming that $p' = q'$.)

The proof divides into two cases, depending on whether j is a client or a virtual node, that is, whether $j \in I_B$ or $j \in I_V$. In each case, we trace back the cause of the collision, and identify some message in $\mathit{sentM}(j, r)'$ that is not in M.

- If j is a client, then there are two possible reasons why port $\langle j, * \rangle$ detects a collision in round r: either it detects a collision during the client phase, which implies that a message sent by a client was lost, or it is notified of a collision by the emulator $\mathsf{vn\text{-}client\text{-}output}_{v,j}$ event for some virtual node $v \in I_V$; the latter event indicates that some message sent by virtual node v was lost, as per Lemma 11.12.15.

- If j is a virtual node, then we can conclude that the round r ballot b for the emulator for virtual node j includes a collision in either $b.clientCD$ or $b.vnCD$. (We can refer to *the* round r ballot since $r > r_{gst}$, and hence round r is green, implying that there is a unique proposer and hence a unique ballot.) In the former case, we know from Lemma 11.12.13 that some client broadcast a message that was not received by virtual node j; in the latter case, we know from Lemma 11.12.14 that some virtual node broadcast a message that was not received by virtual node j.

We now proceed in more detail.

Case 1. $j \in I_B$ is a client. We begin by examining the multiplexer (Figure 10-13): notice that the multiplexer delivers a collision notification to a client port $\langle j, * \rangle$ if and only if $E(multiplexer).inCD_j = \pm$. The variable $inCD$ can be set equal to $\pm$ in two different transitions: $\mathsf{recv}_{j,0}$ (line 709), and $\mathsf{vn\text{-}client\text{-}output}_{j,v}$ (line 653). Thus, this case divides into two subcases: in the first, we show that a message sent by a client is lost; in the second, we show that a message sent by a virtual node is lost. Since these are the only ways in which $E(multiplexer).inCD_j = \pm$, showing a contradiction in each of these cases is sufficient to complete Case 1.

Case 1, Subcase A. Assume that a $\mathsf{recv}(\cdot, \pm, \cdot, \cdot)_{j,0}$ event occurs in the client phase of round r, thus setting $E(multiplexer).inCD_j = \pm$. By the eventual accuracy of the basic broadcast service, we can conclude that there was some message $m \notin M$ sent in the client phase of round r by a port $p \in bcast\text{-}ports_B$ where p is within distance R'_B. Moreover, for all $v \in I_V$, for all $i \in I_B$, port $\langle i, v \rangle$ does not broadcast in the client phase. (Notice that $E(v).outgoing\text{-}msg_i$ is unmodified in the join-ack phase.) Thus we can conclude that for some $i \in I_B$, $p = \langle i, 0 \rangle$. Thus, we conclude that there is a $\mathsf{bcast}(m, \cdot)_{i,0}$ in α in the client phase of round r. From the precondition of the bcast event (line 694, Figure 10-13), we can see that $m = \langle \mathsf{client}, outM, \cdot \rangle$, where $E(multiplexer).outM_i \neq \perp$. The final step is to trace back the origin of the message in $outM$, which is set only in the $\mathsf{bcast}_{i,*}$ transition at the beginning of virtual round r. Thus, we conclude that there is a $\mathsf{bcast}(m, \cdot)_{i,*}$ event in α, and thus in γ_i. Since i is within distance $R'_B \leq R'_V$ of j, we can conclude that $m \in sentM(j, r)'$, and we have just argued that $m \notin M$, resulting in a contradiction: $p' > q'$.

Case 1, Subcase B. Assume that a $\mathsf{vn\text{-}client\text{-}output}(m', \pm)_{j,v}$ occurs at the end of virtual round r, thus setting $E(multiplexer).inCD_j = \pm$. Since the vn-client-output event occurs, we can conclude by Lemma 11.12.15 that $m' = \perp$ and that there exists a round r $\mathsf{bcast}(m, \cdot)_{v',*}$ event in $\gamma_{v'}$, where $m \neq \perp$ and j is within distance R'_V of $loc(v)_V$.

Moreover, it is easy to see that message m is not delivered to port $\langle j, * \rangle$ in round r: the multiplexer delivers messages to the client only when no collisions are reported, as per line 666. Thus we can conclude that $m \notin M$, where M is the set of messages delivered via the $\mathsf{recv}(M, cd, \ldots)_{j,*}$ event described above, resulting in a contradiction: $p' > q'$.

Case 2. $j \in I_v$ is a virtual node. We next examine the case where j is a virtual node. Let $\langle r_1, r_2 \rangle \in updown(v)$ be the epoch containing virtual round r that delineates an execution of v. If $r_2 = \infty$, then choose $r_2 = r$, so that $\langle r_1, r_2 \rangle$ delineates a finite

epoch of v. Thus, γ_v contains some execution from $execs(r_1, r_2)_v$.

Since $r \geq r_{gst}$, we can conclude by Lemma 11.8.12 that rounds r and r_2 are both green rounds (since v is up in rounds r and r_2), and thus by Corollary 11.7.12 that $|execs(r_1, r)_v| = 1$ and $|execs(r_1, r_2)_v| = 1$. Fix $\gamma \in execs(r_1, r)_v$ and $\gamma' \in execs(r_1, r_2)_v$. By Lemma 11.7.20, we can conclude that γ' extends γ, since γ' extends some execution in $execs(r_1, r)_v$, and γ is the only such execution. Also, fix $k \in I_B$ to be a node that completes round r such that $\gamma = exec(r_1, r, k)_v$. (By definition of $execs$, such a k exists.)

Thus, by assumption, there is a round r $\mathsf{recv}(M, \pm, \ldots)_{j,*}$ event in γ. Examine now the calculate-state function (Figure 10-11) which is used to construct γ at the end of round r. From Lemma 11.8.13 we conclude that $E(j).round\text{-}status[r]_k = \mathsf{green}$. Thus it is clear that the round r recv event is constructed by the do-recv function to deliver a collision only if $inCD = \pm$ (line 556, Figure 10-11). Moreover, $inCD \leftarrow \pm$ in two specific circumstances: (1) $E(j).ballot[r].clientCD_k = \pm$, or (2) $E(j).ballot[r].vnCD_k = \pm$ (see lines 546–549, Figure 10-11). Thus the proof breaks down into two subcases. In the first subcase, we show that a round r message sent by a client is lost; in the second subcase, we show that a round r message sent by another virtual node is lost. Since these are the only ways in which a round r ballot can contain a collision, and hence the only ways in which virtual node j can detect a collision in round r, it is sufficient to show a contradiction in each of these cases to conclude the proof.

Case 2, Subcase A. Since $E(j).ballot[r].clientCD_k = \pm$, we can conclude that some node $k' \in I_B$ broadcast ballot $b = E(j).ballot[r]_k$ in the ballot phase of round r. We thus conclude from Lemma 11.12.13 that there exists some $m \in msgs_V$ and some $j' \in I_B$ where j' is within distance R'_V of virtual node j at the beginning of round r such that a round r $\mathsf{bcast}(m, \cdot)_{j',*}$ event occurs in $\gamma_{j'}$ and $\langle \mathsf{client}, m, \cdot \rangle \notin E(j).b.vnM_{k'}$ immediately after the ballot b is set. Thus, we conclude that $m \in sentM(j, r)'$. Moreover, since $\langle \mathsf{client}, m, \cdot \rangle \notin E(j).b.vnM_k$, we can conclude that $\langle \mathsf{client}, m, \cdot \rangle \notin E(j).ballot[r].vnM_k$, and hence from the construction of γ via the calculate-state function, we can conclude that $m \notin M$, the set of messages delivered to virtual node j in round r of γ_j (see line 550, Figure 10-11). This results in a contradiction: $p' > q'$, which concludes our proof.

Case 2, Subcase B. Since $E(j).ballot[r].vnCD_k = \pm$, we can conclude that some node $k' \in I_B$ broadcast ballot $b = E(j).ballot[r]_k$ in the ballot phase of round r. We thus conclude from Lemma 11.12.14 that there exists some $m \in msgs_V$ and some $v' \in I_V$ where v' is within distance R'_V of j at the beginning of round r (and throughout the execution, since virtual nodes do not move) such that a round r $\mathsf{bcast}(m, \cdot)_{v',*}$ event occurs in $\gamma_{v'}$ and $\langle \mathsf{vn}, v', m \rangle \notin E(j).b.vnM_{k'}$ immediately after the ballot b is set. Thus, we conclude that $m \in sentM(j, r)'$. Moreover, since $\langle \mathsf{vn}, v', m \rangle \notin E(j).b.vnM_k$, we can conclude that $\langle \mathsf{vn}, v', m \rangle \notin E(j).ballot[r].vnM_k$, and hence from the construction of γ via the calculate-state function, we can conclude that $m \notin M$, the set of messages delivered to virtual node j in round r of γ_j (see line 550, Figure 10-11). This results in a contradiction: $p' > q'$, which concludes our proof. □

11.13 Constructing an Execution of the Virtual Broadcast Service

In Sections 11.9 and 11.10 we constructed executions for each of the clients and virtual nodes, and for each of the contention managers. In Section 11.12, we constructed a collision detector rule, $\mathit{CD\text{-}rule}_V$ that is complete and eventually accurate. The only component of the system we have not yet considered is the virtual broadcast service itself. In this section, we construct an execution γ_B for the virtual broadcast service. We begin by constructing a trace of the virtual broadcast service, β_B, by extracting the appropriate events from the previously constructed executions. We then construct the full execution, γ_B, by filling in the additional internal events missing from the trace. Finally, we show in Lemma 11.13.1 that the constructed sequence is, in fact, an execution of γ_B.

11.13.1 Constructing a Trace of the Virtual Broadcast Service

We construct a trace β_B of the virtual broadcast service by merging the traces of the clients, virtual nodes, and contention managers. (Notice that every external action of the broadcast service appears in the signature of either a client, virtual node, or contention manager.) We proceed to construct the trace in two steps. First, we combine the traces of the clients and virtual nodes. Since there are no actions shared between clients or virtual nodes, this merging process is straightforward: for each time t, we take every event that occurs at time t in any of the client or virtual node traces and add it to the new merged trace at the same time t. The second step is to combine the resulting trace with the traces from the two virtual contention managers. The contention manager traces contain two types of events: bcast events and cm-advice advice. The bcast events are already included in the merged trace, as they were derived from the client and virtual node traces during the construction of the contention manager traces. It suffices, then, to add the cm-advice events to the merged trace. Recall from the construction of the contention manager traces that the cm-advice events always come after the bcast events. Thus, at each time t we add all the cm-advice events from the contention manager traces after all the other events at time t.

Step 1: Merging the client and virtual nodes traces. We now proceed in more detail to describe the construction of the merged trace. We define β_i to be the trace of execution γ_i, for $i \in I_B \cup I_V$: $\beta_i = \mathit{trace}(\gamma_i)$, $i \in I_B \cup I_V$. More formally, let $E_i = \{\mathsf{bcast}_{i,*}, \mathsf{recv}_{i,*}, \mathsf{fail}_i, \mathsf{recover}_i\}$, that is, the set of external events at process i. Then $\beta_i = \gamma_i|(E_i, \emptyset)$.

We begin with an empty trace, β_B, and add events from β_i, $i \in I_B \cup I_V$. Our goal is to construct a trace with the following property: for all $i \in I_B \cup I_V$, $\beta_B|(E_i, \emptyset) = \beta_i$:

- (Base case:) We begin by adding all events that occur at time $t = 0$. That is, for every event e that occurs in β_i for some $i \in I_B \cup I_V$, add event e to β_B.

- (Inductive step:) Next, assume that we have constructed β_B up to and including time t. Choose t' to be the minimum time $> t$ such that there exists an event e in β_i, for $i \in I_B \cup I_V$ at time t'. If no such t' exists, add a trajectory τ over the empty set of variables where $dom(\tau) = \infty$ and stop. Otherwise, add a trajectory τ (over the empty set of variables) to β_B such that $\ell time(\beta_B) = t'$. That is, $dom(\tau) = t' - t$. Then, for every event e that occurs at time t in β_i for some $i \in I_B \cup I_V$, add event e to β_B.

- Repeat the previous step until $\ell time(\beta_B) = \infty$.

Notice that when this process is complete, every event e in β_i, $i \in I_B \cup I_V$, has been added to β_B at the exact time at which it occured in β_i. Moreover, notice that for all $i, j \in I_B \cup I_V$, the sets E_i and E_j are distinct, that is, $E_i \cap E_j = \emptyset$. This implies that the traces β_i and β_j contain no shared events. Thus, we can conclude that $\beta_B|(E_i, \emptyset) = \beta_i$.

Step 2: Merging β_B and the contention manager traces. The second step of the process is to merge the two contention manager traces, derived from $\gamma_{virtual}$ and γ_{client}, into β_B. Let $\beta_{virtual} = trace(\gamma_{virtual})$ and $\beta_{client} = trace(\gamma_{client})$.

Recall that executions $\gamma_{virtual}$ and γ_{client} are constructed by adding a bcast event to the contention manager execution for every bcast event in γ_i, $i \in I_B \cup I_V$, and adding an immediately following cm-advice event. (See Section 11.10.) As a result of this construction process, every bcast event in $\beta_{virtual}$ and β_{client} has already been added to β_B. It remains only to add the cm-advice events to β_B.

Notice that cm-advice events immediately follow bcast events, and bcast events occur only at the beginning of virtual rounds. Thus each cm-advice event occurs at the beginning of a virtual round, and after a bcast event. Specifically, the event $\mathsf{cm\text{-}advice}(\langle i, * \rangle, \cdot)$ immediately follows a $\mathsf{bcast}_{i,*}$ event. We therefore modify β_B as follows:

- For every round $r > 0$:

  - For every $\mathsf{cm\text{-}advice}(\langle i, * \rangle, \cdot)_d$ event e at the beginning of round r in $\beta_{virtual}$ or β_{client}, where $i \in I_B \cup I_V$ and $d \in \{client, virtual\}$:

    - Add event e to γ_B immediately following the $\mathsf{bcast}_{i,*}$ event at the beginning of round r.

Let $E_{virtual} = \{\mathsf{cm\text{-}advice}_{virtual}, \mathsf{bcast}_{i,*} : i \in I_B \cup I_V\}$, i.e., the external actions for the virtual contention manager. Similarly, let Let $E_{client} = \{\mathsf{cm\text{-}advice}_{client}, \mathsf{bcast}_{i,*} : i \in I_B \cup I_V\}$, i.e., the external actions for the client contention manager. It is straightforward to see that $\beta_B|(E_{virtual}, \emptyset) = \beta_{virtual}$ and $\beta_B|(E_{client}, \emptyset) = \beta_{client}$: when restricted to the bcast events, it is clear by construction that the contention manager traces are equivalent to β_B; the addition of the cm-advice events in the same order and at the same time ensures the desired property.

11.13.2 Constructing an Execution of the Virtual Broadcast Service

In this section we construction an execution γ_B of the virtual broadcast service, based on the trace β_B. We add events related to non-external actions—specifically, next-round events—and construct trajectories over the variables of the virtual broadcast service. We then show in Lemma 11.13.1 that this construction results in an execution. The key claim in the proof is that each event is enabled when it occurs in the hybrid sequence. The proof this claim relies on the integrity of the virtual broadcast service (Lemma 11.11.12) and the construction of the collision detector rule (Section 11.12).

Let $\mathit{ext\text{-}act}_B$ be the set of external actions for the virtual broadcast service, that is,

$$\begin{aligned}
\mathit{ext\text{-}act}_B &= \{\mathsf{bcast}_j : j \in I_B \cup I_V\} \cup \\
&= \{\mathsf{recv}_j : j \in I)B \cup I_V\} \cup \\
&= \{\mathsf{fail}_j : j \in I_B \cup I_V\} \cup \\
&= \{\mathsf{recover}_j : j \in I_B \cup I_V\} \cup \\
&= \{\mathsf{cm\text{-}advice}_d : d \in \{\mathsf{client}, \mathsf{virtual}\}
\end{aligned}$$

Let act_B be the actions of the broadcast service, i.e., $\mathit{act}_B = \mathit{ext\text{-}act}_B \cup \{\mathsf{next\text{-}round}\}$. Additionally, let vars_B be the state variables of the broadcast service (see Figure 8-4).

Recall that an execution γ_B of the broadcast service is a $\langle \mathit{act}_B, \mathit{vars}_B \rangle$-hybrid sequence satisfying two conditions: (1) Each trajectory τ_i in the sequence γ is in the set trajs_B, the trajectories of the broadcast service. (2) For each event a_i in the sequence γ, $\ell\mathit{state}(\tau_{i-1}) \xrightarrow{a_i} f\mathit{state}(\tau_i)$. The first condition indicates that each trajectory is allowed by the broadcast service automaton; the second condition ensures that each action is enabled and applied appropriately according to the transition function.

We construct the sequence γ_B in three steps: (1) Begin with $\gamma_B = \beta_B$, a $\langle \mathit{ext\text{-}act}_B, \emptyset \rangle$-hybrid sequence; (2) Insert next-round events at the end of each virtual round, resulting in a $\langle \mathit{act}_B, \emptyset \rangle$-hybrid sequence; (3) For each trajectory $\tau_i \in \gamma_B$ over the empty set of variables, replace it with a trajectory over vars_B. Finally, we prove in Lemma 11.13.1, that the resulting hybrid sequence γ_B is in fact an execution of the broadcast service.
Step 1: We begin with $\gamma_B = \beta_B$. Thus, at this point, γ_B consists of a hybrid sequence alternating actions in the set $\mathit{act}_B \subset \mathit{ext\text{-}act}_B$ and trajectories over the empty set of variables. That is, γ_B is a $\langle \mathit{act}_B, \emptyset \rangle$-hybrid sequence.
Step 2: We now insert next-round events at the end of each virtual round in γ_B. For each round r we insert a next-round event immediately after the last recv event for round r. More formally, for all $r > 0$:

- Let $t = (r-1) \cdot \mathit{RndLength}_V$.
- Let a be the last recv event that occurs at time t in γ_B. If no such a exists, then let a be the last event prior to time t. Let t' be the time at which event a

occurs.

- Immediately after event a insert a trajectory τ over the empty set of variables where $dom(\tau) = t - t'$. Notice that if there is at least one round r recv event in γ_B, then τ is a point trajectory.

- Immediately after trajectory τ, insert a next-round event.

Notice that after each insertion of a trajectory and a next-round event, γ_B remains a $\langle act_B, \emptyset \rangle$-hybrid sequence.
Step 3: We now replace each trajectory τ_i in γ_B over the empty set of variables with a trajectory τ_i' over the variables $vars_B$. We proceed inductively:

- Initially, for all discrete variables $x \in vars_B$ except *CD-rule*, set $fstate(\tau_0').x$ to the initial state of the broadcast service as specified in Figure 8-4. Choose $fstate(\tau_0').CD\text{-}rule = CD\text{-}rule_B$.

- Initially, for the continuous time variable, $fstate(\tau_0').time = 0$.

- Initially, for the continuous node locations: Let τ_0'' be the first trajectory in execution α of the emulator system. For every $j \in I_B$, assign $fstate(\tau_0').location[j] = fstate(\tau_0'').location[j]$. That is, the initial state of each client is defined to be the initial state of the associated node in α.

- Initially, for every $v \in I_V$, $fstate(\tau_0).location[v] = loc(v)$, the location of the virtual node v.

- Assume inductively that we have determined the initial state of trajectory τ_i', and the initial and final states of all trajectories preceding τ_i' in γ_B.

- For all the discrete variables, the state does not change during the trajectory, and $\ell state(\tau_i') = fstate(\tau_i')$.

- For the continuous time variable, $\ell state(\tau_i').time$ evolves at a rate of 1 throughout the trajectory. That is, $\ell state(\tau_i').time = fstate(\tau_i').time + dom(\tau_i')$.

- For the continous virtual node location variables, the state does not change during the trajectory, and $\ell state(\tau_i').location[v] = fstate(\tau_i').location[v]$, for all $v \in I_V$, since the virtual nodes are static.

- For the continous client location variables, we copy the evolution of the trajectory from execution α, i.e., from the trajectory's evolution in the emulator system. That is, for all $j \in I_B$, for all $t \leq dom(\tau_i)$, define $\tau_i'(t).location[j]$ to be the state variable $location[j]$ at time $fstate(\tau_i).time + t$ in α. (This latter expression represents the real time of the system when time t of the trajectory has passed.)

  More formally, when we say "the state variable $location[j]$ at some time t' in α," we mean as follows: Let τ be a trajectory in execution α such that time

$t' \in [fstate(\tau).time, \ell state(\tau).time]$. (Since the *time* variable advances at a rate of 1, this is simply shorthand for the interval of time covered by the trajectory in α; since $\ell time(\alpha)$ is infinite, we know that there exists such a τ.) Let $t'' = t' - fstate(\tau').time$; that is, t'' is the point in the trajectory where the execution reaches time t'. The state at time t' is thus $\tau'(t'').location[\langle j, 0\rangle]$.

- Lastly, we define $fstate(\tau_{i+1})$, if $dom(\tau_i) \neq \infty$. Let a_{i+1} be the action that occurs immediately after trajectory τ_i and before trajectory τ_{i+1}. (Recall that γ_B is a hybrid sequence in which actions and trajectories alternate, and that γ_B has infinite limit time.) If action a_{i+1} is enabled in $\ell state(\tau_i)$, then define $fstate(\tau_{i+1})$ such that $\ell state(\tau_i) \xrightarrow{a_{i+1}} fstate(\tau_{i+1})$, according to the transition function for the virtual broadcast service. If action a_{i+1} is not enabled, choose $fstate(\tau_{i+1})$ arbitrarily. We will show that this case does not occur.

- Increment i and go to Step 4.

It is straightforward to see that, by construction, $\gamma_B|(vars_B, \emptyset) = \beta_B$; that is, if γ_B is an execution of the broadcast service, then β_B is a trace of the virtual broadcast service. The main claim of this section is that γ_B, as thus defined, is an execution of the virtual broadcast service:

Lemma 11.13.1. *γ_B is an execution of the virtual broadcast service.*

Proof. We need to show two things: (1) Each trajectory is allowed by the trajectories *trajs* of the virtual broadcast service. In our case, this requires showing that whenever time passes, there is no stopping condition that restricts the time passage. More formally, we must show that for each $\tau_i \in \gamma_B$, τ_i is in the trajectories of the broadcast service. (2) For each event in γ_B, the event is enabled and correctly transforms the state prior to the event to the state after the event. More formally, we must show that for each $\tau_i \in \gamma_B$, $\ell state(\tau_i) \xrightarrow{a_{i+1}} fstate(\tau_{i+1})$. As a result of the way in which γ_B is constructed, it is only necessary to show that a_{i+1} is enabled in $\ell state(\tau_i)$.

First, let us consider the trajectories. There are two stopping conditions that restrict the trajectories. First, when $time = round \cdot RndLength$, time cannot pass, i.e., only the point trajectories are available. By inserting next-round events at round boundaries, it is immediately clear that this condition is false when each time-passage trajectory occurs in the sequence. Second, time-passage is not enabled when two ports associated with the same node have different failure status; in the virtual infrastructure system, each node only has one port associated with it, so this condition is never met. That is, for every $i \in I_B \cup I_V$, the only port in $bcast\text{-}ports_V$ associated with node i is port $\langle i, * \rangle$.

Next, we must show that for each $\tau_i \in \gamma_B$, $\ell state(\tau_i) \xrightarrow{a_{i+1}} fstate(\tau_{i+1})$. Due to our construction, we need only show that for each τ_i, a_{i+1} is enabled in $\ell state(\tau_i)$.

We show this by induction on events in γ. Initially, τ_0 is a point trajectory, and $\ell state(\tau_0)$ is the initial state of the broadcast service. We now consider each of the locally-controlled (internal or output) events that occur in γ_B, and argue that they are enabled:

- next-round: The precondition of next-round is as follows:

$$doneP \cup failed = bcast\text{-}ports_V$$
$$time = round \cdot RndLength_V$$

Thus, for the first precondition, we need to show that for each virtual node and client, i.e., for every $i \in I_B \cup I_V$, either port $\langle i, * \rangle \in failed$ or $\langle i, * \rangle \in doneP$: either i has failed and not recovered, or the broadcast service has delivered messages to node i.

First we consider clients. Assume that client $i \in I_B$ is not failed in round $r+1$. Since clients are non-recoverable, this means that there have been no prior fail events. We argue that the following sequence of events must occur in α at the end of virtual round r:

$$\mathsf{recv}_{i,*}, \mathsf{bcast}_{i,*}, \mathsf{bcast}_{i,0} \ .$$

We show that this sequence of events occurs by following the dependencies backwards. By the properties of the basic broadcast service, we know that there is a recv event in the last phase of virtual round r on port $\langle i, 0 \rangle$ in the basic system. In this case, the multiplexer sets *phase* to out. Therefore, according to the trajectories, no time passage is possible until a bcast_0 event, as this is the only event which resets the *phase* flag. Since we are considering the last phase of round r, this bcast_0 event cannot occur until *clientBcast* = true. The *clientBcast* event is set during the $\mathsf{bcast}_{i,*}$ transition, and since a client is a process, this does not occur until a $\mathsf{recv}_{i,*}$ occurs in the previous basic round. Since $\ell time(\alpha)$ is infinite, we know that the desires sequence of events occurs. Thus there is a $\mathsf{recv}_{i,*}$ transition at the end of virtual round r in α; thus by construction there is a $\mathsf{recv}_{i,*}$ event in γ_i, and hence in γ_B. During this $\mathsf{recv}_{i,*}$ transition (line 58, Figure 8-5), the port $\langle i, * \rangle$ is added to *doneP*, as desired.

For virtual nodes, if v is not in a failed state at the end of round r, then it is up in round r. (If v is down in round r, then a fail event is inserted at some time $\leq$ the beginning of round $r+1$, and no recover event is inserted until the next round, by construction.) Since it is up in round r, γ_v contains a round r receive event: γ_v contains a recv event for every round in which v is up. Thus the first precondition for the next-round event is satisfied for both clients and virtual nodes.

The second precondition for the next-round event requires that it occur only at the end of each virtual round, i.e., at some time $r \cdot RndLength_V$. Notice that this follows immediately by Step 2 of the construction of γ_B: we insert the next-round events into the hybrid sequence only at the appropriate times.

- fail, recover: The precondition for the fail event is:

$$\nexists t \in \mathbb{N} : \langle i, t \rangle \in failed$$

The precondition for the recover event is:

$$\exists t \text{ such that } \langle i, t\rangle \in \mathit{failed}$$
$$\mathit{time} \geq t + \mathit{RndLength}_V$$

Consider the case where $v \in I_V$ is a virtual node. For virtual nodes, by construction, the fail and recover events are strictly alternating in execution γ_v: a recover event is only inserted into γ_v following a prior fail event (and an intervening trajectory). This ensures that the precondition for fail events is met, and that the first of the two preconditions for recover events is met.

We now consider the second precondition of recover events, which ensures that a node does not recover until at least time $\mathit{RndLength}_V$ after it fails. Notice thatfail events for virtual nodes are inserted only at the beginning of virtual rounds, and recover events are inserted some $\epsilon > 0$ after the beginning of virtual rounds. Thus, since a virtual node cannot fail and recover in the same virtual round, the time between a fail event and a recover event is at least $\mathit{RndLength}_V$, as required.

The preconditions in this case follow trivially for clients as there is only one fail event per execution and clients are non-recoverable.

- recv: The recv transition has the most involved set of preconditions, and hence is the most interesting case in this proof. Fix some $p = \langle i, * \rangle$ for $i \in I_B \cup I_V$. We address each of the preconditions for the recv_p transition:

  - $\mathit{time} = \mathit{round} \cdot \mathit{RndLength}_V$: This precondition indicates that recv events occur only at the end of a virtual round. If i is a client, this property follows immediately from Lemma 11.10.2, which indicates that the recv events occur at the correct times. If v is a virtual node, this property follows from the construction of γ_v, specifically from the fact that $\mathit{exec}(r_1, r_2, \cdot)_v$ is constructed by calling, alternately, do-bcast and do-recv, the former of which inserts a fragment with zero time passage, and the latter of which inserts a fragment with $\mathit{RndLength}_V$ time passage. Thus all bcast and recv events are inserted at round boundaries.
  - $\mathit{some\text{-}msgs} \subseteq M[\mathit{round}].\mathit{msgs}\text{-}\{\perp\}$: This precondition indicates that every messages delivered in round r is broadcast in round r. By Lemma 11.11.12 we know that if there is a round r $\mathsf{recv}(M, \ldots)_p$ event in γ_B where $m \in M$, then for some $j \in I_B \cup I_V$ there is a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ event in γ_j, and hence also in γ_B. The effects of a round r $\mathsf{bcast}(m, \cdot)_{j,*}$ event is to add message m to $M[r]$, as required. It is straightforward to see that the non-message $\perp$ is never delivered by a recv event either in the case of a client (lines 664 and 665, Figure 10-13) or a virtual node (by construction).
  - **if** $\langle m, p, \cdot\rangle \in M[\mathit{round}]$ **then** $m \in \mathit{some\text{-}msgs}$: This precondition indicates that if port p broadcasts a message in round r, then port p receives that

message in round r. That is, this precondition guarantees the self-delivery property of the virtual broadcast service.

In the case where p is a port on a client, this property follows from Lemma 11.10.3 which states that if port $\langle i, * \rangle$ broadcasts a message m in round r of γ_i, and if a round r $\langle recv \rangle(M, \ldots)_{i,*}$ event occurs in round r, then $m \in M$.

In the case where p is a port on a virtual node, this property follows from Lemma 11.7.13, since the construction of γ_v ensures that every recv event is derived from some execution fragment $exec(r_1, r_2, \cdot)_v$ where $r \in [r_1, r_2]$.

- $p \notin doneP$: Since there is only one $\mathsf{recv}_{i,*}$ event per virtual round (by Lemma 11.10.2 in the case of clients; by construction in the case of virtual nodes), and since $doneP \leftarrow \emptyset$ at the beginning of each virtual round, it follows immediately that $\langle i, * \rangle \notin doneP$.
- $p \notin failed$: If i is a client, it is immediately clear that a round r recv event occurs only if i is not failed. If i is a virtual node, it follows from the construction of γ_i that a round r recv occurs in γ_i only if i is up in round r, and hence only if i is not failed in round r.
- The following set of conditions are all related to the collision detector rule:

  let $sentM = \{m \neq \perp : \exists\, j \in bcast\text{-}ports, \langle m, j, \cdot \rangle \in M[round], |beginRound[j] - beginRound[i]| \leq R\}$
  let $rcvedM = sentM \cap some\text{-}msgs$
  let $sentMinterfere = \{m \neq \perp : \exists\, j \in bcast\text{-}ports, \langle m, j, \cdot \rangle \in M[round], |beginRound[j] - beginRound[i]| \leq R'\}$
  let $rcvedMinterfere = sentMinterfere \cap some\text{-}msgs$
  $cd = CD\text{-}rule(i, round, |sentM|, |rcvedM|, |sentMinterfere|, |rcvedMinterfere|)$

  Notice that by the construction of $CD\text{-}rule_V$, $sentM(i, r)$ is exactly the set $sentM$ defined above during the round r recv by port $p = \langle i, * \rangle$. The same holds for $sentMinterfere(j, r)$, in Section 11.12, and $sentmInterfere$ in the precondition. Thus, by precisely the definition of $CD\text{-}rule_V$, the precondition for cd holds.
- $cm = advice[i]$: By the construction of $\gamma_{virtual}$, if $cm =$ active in this case, then there is a cm-advice(active, p) at the beginning of round r, ensuring that the precondition is met during the round r recv.
- $loc = location[i]$: For the clients, this follows from the operation of the multiplexer: the loc delivered to the client during the $\mathsf{recv}_{i,*}$ event is equal to the loc received by port $\langle i, 0 \rangle$ in the last phase of round r. Since the trajectory of node i is the same in both the real and virtual infrastructure systems, we can conclude that the location is correct. For virtual nodes, this precondition is immediately satisfied, since virtual node locations are static, and hence the location is built into the recv event calculated in do-recv (Figure 10-12).

Thus we conclude that all the preconditions are satisfied for each transition, and hence γ_B is an execution of the virtual broadcast service. □

11.14 Pasting the Executions Together

In this section, we paste all the component executions together, thus creating a single execution γ of the virtual infrastructure system. We begin by stating a basic lemma related to execution-pasting, which is an extension of Theorem 7.3 from [45, 46]:

Lemma 11.14.1. *Let A_1 and A_2 be compatible timed automata, and $A = A_1 \times A_2$. Let α_1 and α_2 be executions of A_1 and A_2, respectively.*

Let β be an $(E, \emptyset)$-sequence, where E is the set of external actions of A. Suppose that $\beta|(E_i, \emptyset) = trace(\alpha_i)$, $i \in \{1, 2\}$.

Then there exists an execution α of A such that $trace(\alpha) = \beta$, and $\alpha_i = \alpha|(A_i, X_i)$, $i \in \{1, 2\}$.

It follows that we can paste together a countable collection of components:

Corollary 11.14.2. *Let $A_1, A_2, \ldots, A_k$ be a finite collection of compatible timed automata, and let $A = A_1 \times A_2 \times \ldots \times A_k$. Let α_i be an execution of A_i.*

Let β be an $(E, \emptyset)$-sequence, where E is the set of external actions of A. Suppose that $\beta|(E_i, \emptyset) = trace(\alpha_i)$, $i \in \{1, 2, \ldots, k\}$.

Then there exists an execution α of A such that $trace(\alpha) = \beta$, and $\alpha_i = \alpha|(A_i, X_i)$, $i \in \{1, 2, \ldots, k\}$.

Using Corollary 11.14.2, we can paste together all the various components executions that we have constructed in the preceding sections:

- γ_i, $i \in I_B$, the clients,
- γ_v, $v \in I_V$, the virtual nodes,
- $\gamma_{virtual}$, the virtual contention manager,
- γ_{client}, the client contention manager, and
- γ_B, the virtual broadcast service.

The end result is an execution of the virtual infrastructure system. Moreover, since the execution of the virtual infrastructure system is constructed from the client executions, we can conclude that the resulting execution, when restricted to a given client's behavior, is equivalent to the original execution *alpha*, when restricted to the client's behavior. This implies that the execution α is a valid emulation of a virtual system. The following theorem, then, proves the main result that the algorithm presented is a correct implementation of a virtual infrastructure system:

Theorem 11.14.3. *For all $i \in I_B$, let C_i be client i, that is, automaton $Remap(A(i)_V, i)$. Then there exists a timed execution γ of the virtual infrastructure system such that for all $i \in I_B$, $\gamma|\langle C.actions_i, C.vars_i\rangle = \alpha|\langle C.actions_i, C.vars_i\rangle$.*

Proof. We first notice that, by construction, β_B contains all the events from the client traces, the virtual node traces, the contention manager traces, and (tautologically) the virtual broadcast trace. That is:

- For all $i \in I_B \cup I_V$, $\beta_B|(E_i, \emptyset) = \beta_i$, where E_i is the set of external actions of automaton $Remap(A(i)_V, i)$.
- $\beta_B|(E_{virtual}, \emptyset) = \beta_{virtual}$, where $E_{virtual}$ is the set of external actions of the virtual contention manager.
- $\beta_B|(E_{client}, \emptyset) = \beta_{client}$, where E_{client} is the set of external actions of the client contention manager.

Since the virtual infrastructure system is the composition of the clients, virtual nodes, contention managers, and the broadcast service, we conclude by Corollary 11.14.2 that there exists an execution γ such that $trace(\gamma) = \beta_B$ and:

- For all $i \in I_B \cup I_V$, $\gamma|(A_i, X_i) = \gamma_i$, where A_i is the set of actions of automaton $Remap(A(i)_V, i)$ and X_i is the set of variables of automaton $Remap(A(i)_V, i)$.
- $\gamma|(A_{virtual}, X_{virtual}) = \gamma_{virtual}$, where $A_{virtual}$ is the set of actions of the virtual contention manager, and $X_{virtual}$ is the set of variables of the virtual contention manager.
- $\gamma|(A_{client}, X_{client}) = \gamma_{client}$, where A_{client} is the set of actions of the client contention manager, and X_{client} is the set of variables of the client contention manager.
- $\gamma|(A_{bcast}, X_{bcast}) = \gamma_B$, where A_{bcast} is the set of actions of the broadcast service, and X_{bcast} is the set of variables of the broadcast service.

In particular, notice that $\gamma|\langle C.actions_i, C.vars_i\rangle = \gamma_i$, which by the construction of γ_i is equal to $\alpha|\langle C.actions_i, C.vars\rangle_i$, concluding the proof. □

11.15 Eventual Collision Freedom

In this section, we show that the virtual broadcast service satisfies the ECF[GA] property, that is, eventual collision free with good advice. In particular, we show that the broadcast service stabilizes in round $r_{gst} + 1$. The proof breaks down into four cases, depending on whether the sending and receiving nodes are clients or virtual nodes:

- When the sending and receiving ports are both clients, the conclusion follows simply from the operation of the multiplexer and the eventual collision freedom property of the basic broadcast service.
- When the sending port is a virtual node and the receiving port is a client, the conclusion follows from noticing (1) that the round is green, and hence good, since we are considering rounds $\geq r_{gst}+1$, (2) that the virtual node is scheduled, since we assume it is advised to be active, and (3) that Lemma 11.12.4 implies that in this situation, the message broadcast is in the appropriate ballot at the receiver, which immediately leads to the desired conclusion.

- When the sending port is a client and the receiving port is a virtual node, the conclusion follows by tracing the message from the client until it arrives in the ballot for the virtual node, again implying the desired result.

- When both the sending and receiving ports are virtual nodes, we argue that Lemma 11.12.2 shows that message $\langle \mathsf{vn}, v, m \rangle$ is broadcast in the vn phase of round r; the ECF property of the basic broadcast service ensures that the message is received by the round r proposer for v', and thus is included in the round r ballot for v', leading to the desired result.

Lemma 11.15.1. *Execution γ satisfies the eventual collision freedom with good advice property.*

Proof. Consider some virtual round $r \geq r_{gst} + 1$, and some message $m \in msgs_V$. (Note that $m \neq \perp$.) We assume that some node broadcasts message m in virtual round r, and that the other conditions for ECF[GA] hold, i.e., the node in question is advised to be active, and no other nearby node broadcasts in round r; we then show that each nearby node—satisfying the requisite conditions—receives message m. The proof breaks down into four cases, depending on whether the sending and receiving nodes are clients or virtual nodes.

- Let $i, j \in I_B$ be two clients and that $j \in \mathsf{near}(location[i], r, I_B \times \{*\}$. That is, j and i are within distance $R_B/4$ of each other at the beginning of round r. We now assume the hypotheses of eventual collision freedom with good advice: Assume that a round r $\mathsf{bcast}(m, \cdot)_{i,*}$ event occurs in γ_B, and that for all $k \in I_B$, $k \neq i$, if $k \in \mathsf{interfere}(location[i], r, I_B \times \{*\})$, then a round r $\mathsf{bcast}(\perp, \cdot)_{k,*}$ event occurs in γ_B. Moreover, assume that port $\langle i, * \rangle$ is advised to be active in round r. We need to show that a round r $\mathsf{recv}(M, \ldots)_{j,*}$ event occurs in γ_B where $m \in M$.

  In this case, the proof follows from the properties of the multiplexer and the eventual collision freedom property of the basic broadcast service. We first argue that port $\langle i, 0 \rangle$ is the only "nearby" port to broadcast a message in the client phase of round r, and that it broadcasts message $\langle \mathsf{client}, m, \cdot \rangle$. Thus we can invoke the eventual collision freedom property of the basic broadcast service to conclude that port $\langle j, 0 \rangle$ receives this message, and hence delivers m to $\langle j, * \rangle$ as desired. We now proceed in more detail.

  It is easy to see that in the client phase of round r there is a $\mathsf{bcast}(\langle \mathsf{client}, m, \cdot \rangle, \cdot)_{i,0}$ event, since $E(multiplexer).outM_i \leftarrow m$ during the hypothesized $\mathsf{bcast}_{i,*}$ event. (Also, see the precondition on line 694, Figure 10-13, where the message broadcast is derived from $E(multiplexer).outM_i$.)

  Similarly, for every $k \in I_B$, $k \neq i$, we can conclude that port $\langle k, 0 \rangle$ broadcasts $\perp$ in the client phase, since $E(multiplexer).outM_k \leftarrow \perp$ during the client phase $\mathsf{bcast}(\perp, \cdot)_{k,*}$ event that we have assumed above.

  Finally, notice that for every $k \in I_B$, for every $v \in I_V$, port $\langle k, v \rangle$ broadcasts $\perp$ in the client phase, since port $\langle k, v \rangle$ never broadcasts during the client phase.

Thus by the eventual collision freedom property of the basic broadcast service, port $\langle j, 0\rangle$ receives message $\langle \mathsf{client}, m, \cdot\rangle$ in the client phase of round r, and adds $\langle \mathsf{client}, m, \cdot\rangle$ to the set $E(multiplexer).inM_i$. Thus, message m is delivered to port $\langle j, *\rangle$ at the end of virtual round r (see line 664, Figure 10-13), as required.

- Let $i \in I_B$ be a client and $v \in I_V$ a virtual node, where $v \in \mathsf{near}(location[i], r, I_V \times \{*\}$. Assume that a round r $\mathsf{bcast}(m, \cdot)_{i,*}$ event occurs in γ_B, and that for all $k \in I_B$, $k \neq i$, if $k \in \mathsf{interfere}(location[i], r, I_B \times \{*\})$, then a round r $\mathsf{bcast}(\perp, \cdot)_{k,*}$ event occurs in γ_B. Moreover, assume that port $\langle i, *\rangle$ is advised to be active in round r. We need to show that if a round r $\mathsf{recv}(M, \ldots)_{v,*}$ event occurs in γ_B, then $m \in M$.

  We begin be noticing that since there is a round r $\mathsf{recv}_{v,*}$ event in γ_B, virtual node v is up in round r: by construction, a $\mathsf{recv}_{v,*}$ event is only added to execution γ_v—and hence γ_B—if v is up in round r. Thus, since $r \geq r_{gst} + 1$, we can conclude by Lemma 11.8.12 that round r is good for v. Let k be the round r proposer for v, and let b be the round r ballot for v, which we know exists by Corollary 11.5.16.

  The key claim, in this case, is that message $\langle \mathsf{client}, m, \cdot\rangle \in b.clientM$. We argue that message $\langle \mathsf{client}, m, \cdot\rangle$ is sent by port $\langle i, 0\rangle$ in the client phase of round r, and that no other port sends a message in the client phase of round r. We can then conclude that port $\langle k, v\rangle$ receives this message in the client phase, by the eventual collision freedom property of the basic broadcast service, and that it is thus added to the ballot.

  As in the previous case, it is easy to see that in the client phase of round r there is a $\mathsf{bcast}(\langle \mathsf{client}, m, \cdot\rangle, \cdot)_{i,0}$ event, since $E(multiplexer).outM_i \leftarrow m$ during the hypothesized $\mathsf{bcast}_{i,*}$ event. (Also, see the precondition on line 694, Figure 10-13, where the message broadcast is derived from $E(multiplexer).outM_i$.)

  Similarly, for every $k \in I_B$, $k \neq i$, we can conclude that port $\langle k, 0\rangle$ broadcasts $\perp$ in the client phase, since $E(multiplexer).outM_k \leftarrow \perp$ during the client phase $\mathsf{bcast}(\perp, \cdot)_{k,*}$ event that we have assumed above.

  Finally, notice that for every $k \in I_B$, for every $v \in I_V$, port $\langle k, v\rangle$ broadcasts $\perp$ in the client phase, since port $\langle k, v\rangle$ never broadcasts during the client phase.

  By the eventual collision freedom property of the basic broadcast service, we can conclude that port $\langle k, v\rangle$ receives $\langle \mathsf{client}, m, \cdot\rangle$ in the client phase and adds it to $E(v).clientM_k$. Since no messages are removed from this set until the beginning of virtual round $r+1$, we can conclude that $\langle \mathsf{client}, m, \cdot\rangle \in b.clientM$.

  The final step of the proof in this case is to argue that message m is received in the round r $\mathsf{recv}_{v,*}$ event in γ_v, and hence in γ_B. Let γ' be the execution including round r used in the construction of γ_v, and choose k', r_1, r_2 such that $\gamma' = exec(r_1, r_2, k')_v$. (By the way in which γ_v is constructed, we know that $r_2 \neq \infty$.) Since b is the round r ballot, and since round r is good for v, we know that at the end of round r, $E(v).ballot[r]_{k'} = b$. Thus, it follows that the

calculate-state function which constructs γ' includes message m in the round r recv event, as desired.

- Let $v \in I_V$ be a virtual node and $j \in I_B$ be a client, where $j \in \mathsf{near}(\mathit{location}[v], r, I_B \times \{*\})$ and j does not fail prior to the end of round r. Assume that a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event occurs in γ_B, $m \neq \perp$, and that for all $k \in I_V$, $k \neq v$, if $k \in \mathsf{interfere}(\mathit{location}[v], r, I_V \times \{*\})$, then a round r $\mathsf{bcast}(\perp, \cdot)_{k,*}$ event occurs in γ_B. Moreover, assume that port $\langle v, * \rangle$ is advised to be active in round r. We need to show that if a round r $\mathsf{recv}(M, \ldots)_{j,*}$ event occurs in γ_B, then $m \in M$.

  Since port $\langle v, * \rangle$ is advised to be active in round r, by the construction of $\gamma_{\mathit{virtual}}$ we can conclude that v is scheduled for round r. Thus the scheduled-veto-2 phase is the last agreement phase of round r for v.

  Next, we notice that $E(v).\mathit{round\text{-}status}[r]_j = \mathsf{green}$ at the end of the last agreement phase of virtual round r: since j is within distance $R_B/4$ of $\mathit{loc}(v)_V$ at the beginning of round r, we can conclude that j is within distance $3R_B/4$ at the beginning of the last agreement phase, since the velocity of a node is bounded; we apply Lemma 11.8.8 to conclude that $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$ at the end of the last agreement phase, the scheduled-veto-2 phase.

  By Lemma 11.12.2 we conclude that there exists some $i \in I_B$ that begins round r for v and remains joined and not failed through the beginning of the scheduled-ballot phase such that there is a round r $\mathsf{bcast}(\langle \mathsf{vn}, v, m \rangle, \cdot)_{i,v}$ event in the vn phase of round r, and a round r $\mathsf{bcast}(\langle \mathsf{vn}, v, b \rangle, \cdot)_{i,v}$ event in the scheduled-ballot phase of round r. Since i is the unique proposer for round r, we can conclude that $E(v).\mathit{ballot}[r]_j = b$.

  We now show that $\langle \mathsf{vn}, v, m \rangle \in b.\mathit{vnM}$, when subsequently implies that $\langle \mathsf{vn}, v, m \rangle \in E(v).\mathit{ballot}[r].\mathit{vnM}_j$, which is sufficient to imply that message m is delivered to client j. First, we can conclude that by the self-delivery property of the basic broadcast service, port $\langle i, v \rangle$ receives the message $\langle \mathsf{vn}, v, m \rangle$ in the vn phase, and adds it to the set *nearby-msgs*. Since i broadcasts a ballot in the scheduled-ballot phase, we can conclude that i is within distance $R_B/4$ of $\mathit{loc}(v)_V$, and hence satisfies the condition on line 261 (Figure 10-5).

  We now argue that all other ports near to $\langle i, v' \rangle$—with the exception of port $\langle i, v \rangle$—broadcast $\perp$ in the vn phase. We first consider ports of the form $\langle k, v \rangle$, for all $k \in I_B$, then ports of the form $\langle k, v'' \rangle$, for $k \in I_B$, $v'' \in I_V$, and finally ports of the form $\langle k, 0 \rangle$, for $k \in I_B$.

  - Ports of the form $\langle k, v \rangle$: By Lemma 11.8.3, there is only one port that has $E(v).\mathit{roundCM} = \mathsf{active}$ at the end of the client phase; moreover we know that $E(v).\mathit{roundCM}_i = \mathsf{active}$, since i broadcasts a message in the vn phase. Thus, we can conclude that for all $k \in I_B$, $k \neq i$, port $\langle k, v \rangle$ does not broadcast a message in the vn phase of round r, that is, there is a $\mathsf{bcast}(\perp, \cdot)_{k,v}$ transition in the vn phase of round r.

– Ports of the form $\langle k, v'' \rangle$: For all $k \in I_B$, for all $v'' \in I_V$, $v'' \neq v$, we argue that if $\langle v'', * \rangle \in \mathsf{interfere}(\mathit{location}[v], r, I_V \times \{*\})$, then either there is a round r $\mathsf{bcast}(\bot, \cdot)_{v'',*}$ in $\gamma_{v''}$, or there is no round r $\mathsf{bcast}_{v'',*}$ event.

  We have assumed that no nearby port broadcasts a non-$\bot$ message in round r in γ_B. For the sake of contradiction, consider some $k \in I_B$ such that there is a $\mathsf{bcast}(\langle \mathsf{vn}, v'', m' \rangle, \cdot)_{k,v''}$ transition in the vn phase of round r. Then by Lemma 11.12.12, there is a round r $\mathsf{bcast}(m', \cdot)_{v'',*}$ event in $\gamma_{v''}$. However, by assumption $m' = \bot$. Since, as per line 238, we know that $m' \neq \bot$, this implies a contradiction. We conclude that if there is a $\mathsf{bcast}_{k,v''}$ event in the vn phase of round r where v'' is within distance $2R'_V$ of v, then port $\langle k, v'' \rangle$ broadcasts $\bot$.

  Notice that the preceding argument is for a virtual node v'' that is in the interference range of virtual node v. We are interested, in this case, in nodes $k \in I_B$ that are within the interference range of port $\langle i, v \rangle$. Consider some $k \in I_B$ that is within distance $2R'_B$ of i, and some $v'' \in I_V$, where $v'' \neq v$. First, port $\langle k, v'' \rangle$ broadcasts a non-$\bot$ message only if k is within distance $R_B/4$ of v''. And we know that i is within distance $R_B/4$ of v. Thus, if port $\langle k, v'' \rangle$ were to broadcast a non-$\bot$ message, then we could conclude that v'' is within distance $2R'_B = 2R'_V - R_B/2$ of v', resulting in a contradiction.

– Ports of the form $\langle k, 0 \rangle$: For all k $in I_B$, port $\langle k, 0 \rangle$ of the multiplexer does not broadcast in the vn phase.

As argued above, there is only message broadcast by a port $p \in \mathit{bcast\text{-}ports}_B$ in the interference radius of $\langle i, v \rangle$, that is, within distance $2R'_B$. We conclude from the eventual collision freedom property of the basic broadcast service, that port $\langle i, v \rangle$ receives message $\langle \mathsf{vn}, v, m \rangle$ in the vn phase of round r, and adds this message to *nearby-msgs*.

We thus conclude that $E(v).vnM_i \leftarrow$ *nearby-msgs*: Since no other ports near to i broadcast a message in the vn phase of round r, we can conclude that $|\mathit{nearby\text{-}msgs}| = 1$ (line 264), that is, $\langle \mathsf{vn}, v, m \rangle$ is the only message is *nearby-msgs*. By the eventual accuracy of the basic broadcast service, we conclude the $cd = \mathsf{null}$ in the vn phase of round r, as the only message broadcast by a neighboring replica was, in fact, received (line 266). Finally, since the only message in *nearby-msgs* is $\langle \mathsf{vn}, v, m \rangle$, and since v is scheduled, we can conclude that no message is received from an unscheduled, nearby virtual node (line 268). Thus we conclude that $E(v).vnM_i \leftarrow$ *nearby-msgs*, and hence message $\langle \mathsf{vn}, v, m \rangle \in E(v).vnM_i$ when ballot b is formed.

Thus we conclude that message $\langle \mathsf{vn}, v, m \rangle \in E(v).\mathit{ballot}[r].vnM_j$ at the end of virtual round r. We now examine the preconditions for the vn-client-output transitions (Figure 10-4): $E(v).\mathit{round\text{-}status}[r]_i = \mathsf{green}$, $v \in \mathit{schedule}[r \mod \mathsf{SMAX}]$, and there exists $\langle \mathsf{vn}, v, m \rangle \in E(v).\mathit{ballot}[r]_i.vnM$. Thus, there is an output event $\mathsf{vn\text{-}client\text{-}output}(\langle \mathsf{vn}, v, m \rangle, \mathsf{null})_{i,v}$ at the end of round r.

It follows that $\langle \mathsf{vn}, v, m \rangle$ is added to the set $E(\mathit{multiplexer}).\mathit{inM}_i$, and thus message m is delivered to port $\langle i, * \rangle$ during the $\mathsf{recv}_{i,*}$ event that follows immediately, implying that $m \in M$, as desired.

- Let $v, v' \in I_V$ be two virtual nodes, where $v' \in \mathsf{near}(\mathit{location}[v], r, I_V \times \{*\}$. Assume that a round r $\mathsf{bcast}(m, \cdot)_{v,*}$ event occurs in γ_B, and that for all $k \in I_V$, $k \neq v$, if $k \in \mathsf{interfere}(\mathit{location}[v], r, I_V \times \{*\})$, then a round r $\mathsf{bcast}(\perp, \cdot)_{k,*}$ event occurs in γ_B. Moreover, assume that port $\langle v, * \rangle$ is advised to be active in round r. We need to show that a round r $\mathsf{recv}(M, \ldots)_{v',*}$ event occurs in γ_B where $m \in M$.

  Since port $\langle v, * \rangle$ is advised to be active in round r, by the construction of $\gamma_{\mathit{virtual}}$ we can that v is scheduled for round r. If v is up in round r, we can conclude (by Lemma 11.8.12 that round r is green for v, and hence good. We conclude by Lemma 11.12.2 that for some $j \in I_B$ there is a $\mathsf{bcast}(\langle \mathsf{vn}, v, m \rangle, \cdot)_{j,v}$ in the vn phase of round r, and a $\mathsf{bcast}(\langle \mathsf{vn}, v, b \rangle, \cdot)_{j,v}$ in the $\mathsf{scheduled\text{-}ballot}$ phase of round r.

  Since there is a $\mathsf{recv}_{v',*}$ event in $\gamma_{v'}$, we can conclude that v' is up in round r. Since $r \geq r_{gst} + 1$, we can conclude by Lemma 11.8.12 that round r is good v'. Let i be the round r proposer for v', and let b be the round r ballot.

  We now argue that all other ports near to $\langle i, v' \rangle$—with the exception of port $\langle j, v \rangle$—broadcast $\perp$ in the vn phase. We first consider ports of the form $\langle k, v \rangle$, for all $k \in I_B$, ports of the form $\langle k, v'' \rangle$, for $k \in I_B$, $v'' \in I_V$, and ports of the form $\langle k, 0 \rangle$, for $k \in I_B$.

  - Ports of the form $\langle k, v \rangle$: By Lemma 11.8.3, there is only one port that has $E(v).\mathit{roundCM} = \mathsf{active}$ at the end of the client phase; moreover we know that $E(v).\mathit{roundCM}_j = \mathsf{active}$, since j broadcasts a message in the vn phase. Thus, we can conclude that for all $k \in I_B$, $k \neq j$, port $\langle k, v \rangle$ does not broadcast a message in the vn phase of round r, that is, there is a $\mathsf{bcast}(\perp, \cdot)_{k,v}$ transition in the vn phase of round r.
  - Ports of the form $\langle k, v'' \rangle$: For all $k \in I_B$, for all $v'' \in I_V$, $v'' \neq v$, we argue that if $\langle v'', * \rangle \in \mathsf{interfere}(\mathit{location}[v], r, I_V \times \{*\})$, then either there is a round r $\mathsf{bcast}(\perp, \cdot)_{v'',*}$ in $\gamma_{v''}$, or there is no round r $\mathsf{bcast}_{v'',*}$ event.

    We have assumed that no nearby port broadcasts a non-$\perp$ message in round r in γ_B. For the sake of contradiction, consider some $k \in I_B$ such that there is a $\mathsf{bcast}(\langle \mathsf{vn}, v'', m' \rangle, \cdot)_{k,v''}$ transition in the vn phase of round r. Then by Lemma 11.12.12, there is a round r $\mathsf{bcast}(m', \cdot)_{v'',*}$ event in $\gamma_{v''}$. However, by assumption $m' = \perp$. Since, as per line 238, we know that $m' \neq \perp$, this implies a contradiction. We conclude that if there is a $\mathsf{bcast}_{k,v''}$ event in the vn phase of round r where v'' is within distance $2R'_V$ of v, then port $\langle k, v'' \rangle$ broadcasts $\perp$.

    Notice that the preceding argument is for a virtual node v'' that is in the interference range of virtual node v. We are interested, in this case, in nodes $k \in I_B$ that are within the interference range of port $\langle i, v' \rangle$. Consider

some $k \in I_B$ that is within distance $2R'_B$ of i, and some $v'' \in I_V$, where $v'' \neq v$. First, port $\langle k, v'' \rangle$ broadcasts a non-$\perp$ message only if k is within distance $R_B/4$ of v''. And we know that i is within distance $R_B/4$ of v'. Thus, if port $\langle k, v'' \rangle$ were to broadcast a non-$\perp$ message, then we could conclude that v'' is within distance $2R'_B = 2R'_V - R_B/2$ of v', resulting in a contradiction.

– Ports of the form $\langle k, 0 \rangle$: For all k $in I_B$, port $\langle k, 0 \rangle$ of the multiplexer does not broadcast in the vn phase.

As argued above, there is only message broadcast by a port $p \in \mathit{bcast\text{-}ports}_B$ in the interference radius of $\langle i, v \rangle$, that is, within distance $2R'_B$. We conclude from the eventual collision freedom property of the basic broadcast service, that port $\langle i, v \rangle$ receives message $\langle \mathsf{vn}, v', m \rangle$ in the vn phase of round r, and adds this message to *nearby-msgs*.

Notice that i is the proposer for v', and hence i is within distance $R_B/4$ of $\mathit{loc}(v')_V$. Thus, the condition on line 261 is satisfied, and we thus conclude that $E(v').vnM_i \leftarrow \mathit{nearby\text{-}msgs}$: Since no other ports near to i broadcast a message in the vn phase of round r, we can conclude that $|\mathit{nearby\text{-}msgs}| = 1$ (line 264), that is, $\langle \mathsf{vn}, v', m \rangle$ is the only message is *nearby-msgs*. By the eventual accuracy of the basic broadcast service, we conclude the $cd = \mathsf{null}$ in the vn phase of round r, as the only message broadcast by a neighboring replica was, in fact, received (line 266). Finally, since the only message in *nearby-msgs* is $\langle \mathsf{vn}, v', m \rangle$, and since v is scheduled, we can conclude that no message is received from an unscheduled, nearby virtual node (line 268). Thus we conclude that $E(v').vnM_i \leftarrow \mathit{nearby\text{-}msgs}$, and hence message $\langle \mathsf{vn}, v', m \rangle \in E(v').vnM_i$ when ballot b is formed.

The final step of the proof in this case is to argue that message m is received in the round r $\mathsf{recv}_{v,*}$ event in γ_v, and hence in γ_B. Let γ' be the execution including round r used in the construction of γ_v, and choose k', r_1, r_2 such that $\gamma' = \mathit{exec}(r_1, r_2, k')_v$. (By the way in which γ_v is constructed, we know that $r_2 \neq \infty$.) Since b is the round r ballot, and since round r is good for v, we know that at the end of round r, $E(v).\mathit{ballot}[r]_{k'} = b$. Thus, it follows that the calculate-state function which constructs γ' includes message m in the round r recv event, as desired. From this we can conclude that there is a $\mathsf{recv}(M, \cdot)_{v',*}$ event in round r of $\gamma_{v'}$ where $m \in M$, as desired.

□

11.16 Virtual Node Failures

In this section, we examine when a virtual node is failed and when a virtual node is non-failed. Intuitively, a virtual node is non-failed as long as there is some mobile node nearby acting as an emulator. A nearby node, however, can act as an emulator only after it completes the join protocol; the success or failure of the join protocol

depends on whether the network has stabilized. Our goal in this section, then, is to describe a precise situation in which we can be sure that a virtual node is non-failed.

We consider some range of virtual rounds $[r_1, r_2]$, where $r_1 \geq r_{gst} + 1$, and $r_2 \geq r_1 + \mathsf{SMAX}$. We assume that in each virtual round $r \in [r_1, r_2]$ there is some node $i_r \in I_B$ that has not failed and is near to virtual node v at the beginning of round r. We also assume that i_r remains non-failed and near for long enough to complete the join or reset protocol, and also long enough to pass on the state information to a later emulator. Since it may require SMAX rounds before a node can complete the join protocol, we therefore assume that each i_r remains nearby and non-failed through round $r + \mathsf{SMAX}$. In this case, virtual node v is non-failed in virtual rounds $[r_1 + \mathsf{SMAX}, r_2]$.

We first prove two preliminary lemmas. The first lemma shows that if some node $i \in I_B$ begins round r for v and remains nearby and non-failed through the end of round r, then v is up in round r. The main fact that must be proved in this lemma is that i successfully prevents any node from resetting v.

Lemma 11.16.1. *Let $v \in I_V$ be a virtual node, and $r > 0$ a virtual round. Assume that $i \in I_B$ begins round r for v and remains non-failed and within distance $R_B/4$ of $loc(v)_V$ through the end of round r. Then v is up in round r.*

Proof. By assumption v completes round r for v. Hence it remains only to show that v is not reset in round r. We argue that i broadcasts a veto message in the join-veto phase, thus preventing any node from resetting v in round r. We now proceed in more detail.

Assume for the sake of contradiction that $j \in I_B$ resets v in round r. We know in this case that j is within distance $R_B/4$ of $loc(v)_V$ at the end of the join-veto phase (line 480, Figure 10-9), and hence is within distance $3R_B/4$ of $loc(v)_V$ at the beginning of the join-veto phase. Thus at the beginning of the join-veto phase nodes i and j are within distance R_B of each other. We can also conclude, as per line 480 (Figure 10-9) that v is scheduled in round r.

Since i is joined and not failed, and since v is scheduled for round r, we can conclude that i broadcasts a veto message in the join-veto phase (lines 467–469, Figure 10-9). By the completeness of the basic broadcast service, we can conclude that node j either receives the veto message or detects a collision in the join-veto phase, and hence j does not reset v, resulting in a contradiction. □

The second preliminary lemma shows that the join protocol successfully allows nodes to join, after the network stabilizes. We assume that there is some node $i' \in I_B$ that is joined and not failed during the beginning of the join protocol, i.e., through the beginning of the join-ack phase. Thus there is at least one node available to send a join response in the join-ack phase. The regional contention manager ensures that there is exactly one node that sends a join response. Eventual collision freedom ensures that this message is received. Eventual accuracy shows that no collisions are detected in the join-ack phase. The result, then, is that any node trying to join v after the network stabilizes will complete the join protocol successfully.

Lemma 11.16.2. *Let $v \in I_V$ be a virtual node, and $r \geq r_{gst} + 1$ a virtual round such that v is scheduled for round r. Let $i' \in I_B$ be a node that is joined and not failed at the beginning of the* join-ack *phase of round r. Let $j \in I_B$ be a node that is non-failed through the end of round r, and is within distance $R_B/4$ of $loc(v)_V$ throughout the round. Then j completes round r for v.*

Proof. In order to show that j completes round r for v, we must show that the end of round r, $E(v).joined_j = \mathsf{true}$ and $E(v).failed_j = \mathsf{false}$. Notice that the latter is true by assumption.

If j begins round r for v, i.e., if $E(v).joined_j = \mathsf{true}$ at the beginning of the round, then it clearly remains joined throughout the round, as it remains within distance $R_B/4$ of $loc(v)_V$ throughout the round, by assumption.

Assume, then, that j does not begin round r for v, i.e., $E(v).joined_j = \mathsf{false}$ at the beginning of round r. We show that in this case j joins v in round r. The argument goes as follows: First, node j broadcasts a join request in the join phase of the protocol. Second, we identify some node $i \in I_B$ that broadcasts a join response in the join-ack phase. The key step is showing that exactly one node broadcasts a join response. (This is the second place in this paper that we use the properties of the regional contention manager; the first is in Lemma 11.8.3.) Third, it follows from eventual collision freedom and eventual accuracy that j receives this join information and completes the join protocol. We now proceed in more detail.

Step 1. First, we argue that port $\langle j, v\rangle$ broadcasts a join response in the join phase of round r. Consider the $\mathsf{recv}_{j,v}$ event in the unscheduled-veto-2 phase. Since j is near to v (line 431, Figure 10-8), and j is not joined (line 432, Figure 10-8), and v is scheduled for round r (line 432, Figure 10-8), we can conclude that on line 433, node j chooses to broadcast message $\langle \mathsf{vn}, v, \mathsf{join}\rangle$ in the join phase of round r.

Step 2. We now argue that exactly one node responds to the join request. We begin by showing that the requirements of the regional contention manager CM_v are met, which implies that it designates exactly one node to be active in the join-ack phase. By Lemma 11.8.2, we know that α satisfies $loc(v)_V$-restricted contention. By assumption, we know that some $i' \in I_B$ is joined and not failed at the beginning of the join-ack phase. Therefore, $\langle i', v\rangle \in \mathsf{near}(loc(v)_V, [r_a, r_a + 1], bcast\text{-}ports_B)$, where r_a is the join phase and $r_a + 1$ is the join-ack phase. (Otherwise, if i' is too far away, it sets $E(v).joined_{i'} = \mathsf{false}$.) Moreover, port $\langle i', v\rangle$ contends for CM_v when (and only when) it has joined v (lines 164–167, Figure 10-4). We can therefore conclude that port $\langle i', v\rangle$ contends for the join (and join-ack) phases of round r. As a result, the regional contention manager guarantees that there exists some port $\langle i, v\rangle$, $i \in I_B$, where the following properties hold:

- Port $\langle i, v\rangle$ contends for CM_v in the join phase of round r. This implies that $E(v).joined_i = \mathsf{true}$ at the beginning of the unscheduled-veto-2 phase.
- Port $\langle j, v\rangle$ does not fail prior to the beginning of the join-ack phase of round r.
- Port $\langle j, v\rangle \in \mathsf{near}(loc(v)_V, [r_a, r_a + 1], bcast\text{-}ports_B)$, i.e., i is near to v at the beginning of the join and join-ack phases. Together with the previous two points, this implies that i remains joined through the beginning of the join-ack phase.

- Contention manager CM_v advises port $\langle i, v\rangle$ to be active in the join and join-ack phases of round r. This means that in the join phase $\mathsf{recv}_{i,v}$ event, $cm = \mathsf{active}$.

- At most one port is advised by CM_v to be active in the join-ack phase of round r. Thus for every port $\langle k, v\rangle$, $k \neq i$, CM_v does not advise port $\langle k, v\rangle$ to be active in the join-ack phase. Since all contention managers are conservative, and since port $\langle k, v\rangle$ does not contend on any other contention manager, we can conclude that in the join phase $\mathsf{recv}_{k,v}$ event, $cm = \mathsf{passive}$.

It remains to show one final property of port $\langle i, v\rangle$: we need to show that $\langle i, v\rangle$ either receives the join request from node j, or detects a collision, in the join phase of round r: since i and j are both within distance $R_B/4$ of $loc(v)_V$, we can conclude that i and j are within distance R_B of each other, and hence this fact follows by the completeness of the basic broadcast service.

Thus all the conditions are satisfies for node i to broadcast a join response (see Figure 10-9):

- line 442: Virtual node v is scheduled for round r.

- line 443: Port $\langle i, v\rangle$ either receives the join request from j, or detects a collision.

- line 445: As argued above, $E(v).joined_i = \mathsf{true}$ at the end of the join phase (and the beginning of the join-ack phase), and $cm = \mathsf{active}$ in the join phase recv event for port $\langle j, v\rangle$.

Thus on line 446 (Figure 10-9) port i sets $E(v).outgoing\text{-}msg_i$ to an appropriate join response message. Since i does not fail prior to the $\mathsf{bcast}_{i,v}$ event in the join-ack phase of round r, this message is broadcast in the join-ack phase.

Step 3. It remains only to show that port $\langle j, v\rangle$ receives this message. Recall that for all $k \in I_B$, $k \neq j$, in the $\mathsf{recv}_{k,v}$ transition in the join phase, $cm = \mathsf{passive}$, due to the operation of regional contention manager CM_v and the fact that all contention managers are conservative. Moreover, for all $v' \in I_V$, $v' \neq v$, if v' is within distance $R_B + 2R'_B$ of v, then v' is not scheduled, since the *schedule* is non-conflicting. From these two facts, we conclude that for all $k \in I_B$, $k \neq i$, for all $v' \in I_V$, if port $\langle k, v'\rangle$ is within distance $2R'_B$ of port $\langle j, v\rangle$, then port $\langle k, v'\rangle$ does not broadcast a message in the join-ack phase of round r: either $cm = \mathsf{passive}$, if $v = v'$, or v' is not scheduled.

Thus, by the eventual collision freedom property of the basic broadcast service, we conclude that port $\langle j, v\rangle$ receives the message broadcast by port $\langle i, v\rangle$ in response to the join request. Moreover, since only one message was sent by a node within the interference range of $\langle j, v\rangle$, and since that message was received, we conclude by the eventual accuracy of the basic broadcast service (Lemma 8.1.42) that port $\langle j, v\rangle$ does not detect a collision in the join-ack phase of round r. Thus, on lines 460–463 (Figure 10-9), node j sets $E(v).joined_j = \mathsf{true}$, as desired. Since j remains near to v and non-failed through the end of round r, j completes round r for v. □

Finally, we can show that virtual node v is non-failed as long as in each round there exists some node available that remains nearby for sufficiently long.[2]

Theorem 11.16.3. *Let $v \in I_V$ be a virtual node. Let $r_1 \geq r_{gst}+1$ and $r_2 \geq r_1+\mathsf{SMAX}$ be two virtual rounds. Assume that for every $r \in [r_1 r_2]$ there exists some node $i_r \in I_B$ with the following properties:*

- *Node i_r does not fail prior to the end of round $r + \mathsf{SMAX}$.*
- *Node i_r is within distance $R_B/4$ of $loc(v)_V$ from the beginning of round r until the end of round $r + \mathsf{SMAX}$.*

Then virtual node v is non-failed in rounds $[r_1 + \mathsf{SMAX}, r_2]$.

Proof. Recall from Section 11.9 that, as per the construction of γ_v, virtual node v fails in round r when v is down in round r. Thus our goal is to show that virtual node v is up in rounds $[r_1 + \mathsf{SMAX}, r_2]$. In fact, choose $r'_1 \in [r_1, r_1 + \mathsf{SMAX}]$ such that $v \in \mathit{schedule}[r'_1]$; by the completeness of the schedule, we know that there is some such r'_1. We in fact show that v is up in rounds $[r'_1 + 1, r_2]$: v is reset in round r'_1, if it is not already up, and remains up throughout the remaining rounds.

We begin by showing that either virtual node v is up in round r'_1, or v is reset in round r'_1. We then proceed to show that virtual node v remains up in each of the following rounds through round r_2. Specifically, for each virtual round we identify a node that begins that round and remains joined and not failed through the round. That is, we say that node i is **responsible** for round r (with respect to v) if i begins round r for v and remains joined and not failed through the end of round r. We argue that if there is some node responsible for each round, then v is up in that round: the responsible node broadcasts in the join-veto round, preventing a reset, and also satisfies the requirement that some node complete the round.

For round $r'_1 + 1$, this "responsible" node is the node that performed the reset in round r'_1, if v is reset in round r'_1, or $i_{r'_1}$, otherwise. For each ensuing round r, the responsible node is either the same as the responsible node for round $r - 1$, or alternatively node i_r. In this way, by identifying a node responsible for each round, and thus we ensure that the virtual node is up in each round. In particular, Lemma 11.16.1 shows that if some node is responsible for round r, then v is up in round r.

We now proceed in more detail. We begin by examining round r'_1; we consider two subcases, depending on whether there is some node $i' \in I_B$ that is available to broadcast a message in the join-ack phase. If there is some node that is joined and not failed at the beginning of the join-ack phase, we argue that $i_{r'_1}$ successfully joins v in round r; otherwise, we argue that $i_{r'_1}$ resets v in round r'_1. In each case, we argue that $i_{r'_1}$ is responsible for rounds $[r'_1, r'_1 + \mathsf{SMAX}]$.

[2]Notice that the hypothesis of this lemma can be slightly weakened; in fact, it is only important that nodes are available in rounds in which v is scheduled; in unscheduled rounds, new nodes cannot join v.

- Assume there exists some node $i' \in I_B$ that is joined and not failed at the beginning of the join-ack phase of round r. By Lemma 11.16.2 we conclude that node $i_{r'_1}$ completes round r for v. Since, by assumption, node $i_{r'_1}$ remains near to v and not failed for rounds $[r'_1, r'_1 + \mathsf{SMAX}]$, we can conclude that $i_{r'_1}$ is responsible for rounds $[r'_1 + 1, r'_1 + \mathsf{SMAX}]$.

- Assume there is no node $i' \in I_B$ that is joined and not failed at the beginning of the join-ack phase of round r. In this case, node $i_{r'_1}$ resets v in round r'_1. In particular, since there is no node that is joined and not failed at the beginning of the join-ack phase, we can conclude that there is no node joined and not failed at the beginning of the join-veto phase: it is impossible for any node to join during the join-ack phase, since no node broadcasts a join response in the join-ack phase.

  Moreover, since there is no node joined and not failed at the beginning of the join-veto phase, no node broadcasts a message during the join-veto phase. Thus we can conclude that node $i_{r'_1}$ does not receive any messages during the join-veto phase. By the eventual accuracy of the basic broadcast service (Lemma 8.1.42) we can conclude that port $i_{r'_1}$ does not detect a collision in the join-veto phase of round r. Since v is scheduled, and since $i_{r'_1}$ is close to v, all the conditions are therefore met for $i_{r'_1}$ to reset v (see lines 480–481, Figure 10-9).

  We therefore conclude that $i_{r'_1}$ completes round r'_1 for v. As in the previous case, we can conclude that $i_{r'_1}$ is responsible for rounds $[r'_1 + 1, r'_1 + \mathsf{SMAX}]$.

We have now shown that there exists some node that is responsible for rounds $[r'_1 + 1, r'_1 + \mathsf{SMAX}]$. By Lemma 11.16.1, we can conclude that v is up in rounds $[r'_1 + 1, r'_1 + \mathsf{SMAX}]$. Assume inductively that for some $c > 0$, we have shown that there exists a node responsible for each round in the round $[r'_1 + 1, r'_1 + c \cdot \mathsf{SMAX}]$, and that v is up in these rounds. (We have already established this for $c = 1$.) If $r'_1 + c \cdot \mathsf{SMAX} < r_2$, we now identify a node to be responsible for the rounds $[r'_1 + c \cdot \mathsf{SMAX} + 1, r'_1 + (c+1) \cdot \mathsf{SMAX}]$, and conclude by Lemma 11.16.1 that v is up in these rounds. Let $r = r'_1 + c \cdot \mathsf{SMAX}$.

Specifically, we argue that node i_r is responsible for rounds $[r+1, r+\mathsf{SMAX}]$, which is the desired range. Since there exists some node $i' \in I_B$ that is responsible for round r, and since v is up in round r, Lemma 11.16.2 shows that node i_r completes round r for v. Since i_r, by assumption, remains near v and non-failed through the end of round $r + \mathsf{SMAX}$, we conclude that i_r is responsible for v through round $[r + \mathsf{SMAX}]$, as desired. We thus conclude that v is up in rounds $[r'_1, r_2]$, which implies that v is non-failed in rounds $[r'_1, r_2]$. □

Concluding Material

Chapter 12

Conclusion

In this thesis we have introduced the idea of *virtual infrastructure*, a set of new techniques for simplifying the development of reliable applications for unreliable and unpredictable wireless ad hoc networks. We have shown how to emulate virtual infrastructure in fault-prone mobile networks, considering both networks with reliable communication (Part I), and unreliable communication (Part II). In this chapter, we begin (Section 12.1) by briefly reviewing the main contributions of this thesis. We then discuss in Section 12.2 a protype implementation based on some of the ideas in this thesis. Finally, we conclude in Section 12.3 with a discussion of open questions and ongoing research.

12.1 Contributions

The first main contribution of this thesis is the introduction of virtual infrastructure, that is, the idea of emulating reliable fixed infrastructure in wireless networks that have no preexisting infrastructure. We describe three different infrastructure abstractions, each with different functionalities and capacities.

- The first of these abstractions, the Virtual Object Layer (Section 3.1), provides clients with reliable data storage, known as virtual objects. The virtual objects are (relatively) reliable, and communication between the clients and the virtual objects is reliable. We show how to use these virtual objects to build a reconfigurable, atomic distributed shared memory (Chapter 6).

- The second of these abstractions, the Virtual Node Layer (Section 3.2), includes virtual computation; the abstraction consists of both clients and virtual nodes, which communicate reliably. Communication can be long-distance, via a virtual GeoCast service (as discussed in Section 3.2), or via a simple local broadcast (as discussed in Chapter 7). The virtual nodes can be used to form an overlay network that facilitates communication and coordination among the clients.

- The third virtual infrastructure abstraction is designed for collision-prone networks that do not support reliable communication. The abstraction consists

of clients and virtual nodes that communicate via *synchronous* local wireless broadcast. Communication, however, is unreliable: messages can be lost due to interference, collisions, and other unpredictable anomalies. The clients and virtual nodes are equipped with collision detectors, which help to detect when messages are lost, and contention managers, which help to determine how to reduce contention on the wireless channel.

By designing simple abstractions, we facilitate the design of reliable software for wireless networks.

The second main contribution of this thesis is a set of algorithms for emulating virtual infrastructure in fault-prone networks that support reliable and timely communication. We show how to extend standard replicated-state-machine techniques to implement Virtual Object Layers and Virtual Node Layers. The basic underlying idea is that each mobile node in the vicinity of a virtual node participates in replicating that virtual node. The key challenge here lies in supporting nodes that join and leave the emulation, and in handling the situation when a virtual node (or object) is reset.

The third main contribution of this thesis is a protocol for emulating virtual infrastructure in fault-prone and *collision-prone* networks. We introduce (in Section 8.2) a more realistic model for wireless networks that accounts for the real unpredictable behavior of wireless radios: messages are lost due to interference, collisions, and other anomalies. As a result, algorithms for these networks are much more involved. We develop an efficient emulation algorithm that emulates one round of the virtual infrastructure abstraction in a bounded number of real rounds of communication; moreover, the size of each message is bounded. As in the earlier emulation algorithms, the basic underlying idea is to replicate the virtual node at mobile nodes in the nearby region. While inspired by techniques such as three-phase commit, this algorithm relies on a new veto-based paradigm for achieving partial-agreement among the replicas.

To summarize, we introduce in this thesis the idea of virtual infrastructure; we present three virtual infrastructure abstractions; and we develop two different techniques for emulating virtual infrastructure in wireless networks, considering networks with both reliable and unreliable communication.

12.2 Building a Prototype Virtual Infrastructure

In order to better understand the trade-offs and engineering considerations involved in the deployment of virtual infrastructure, we have begun to develop a prototype implementation based on the ideas presented in this thesis (see [14]).

12.2.1 Virtual Node Emulator

We focus on a virtual node layer that is specifically optimized for practical implementation. The virtual nodes' in this layer are event driven, taking (atomic) steps in which they (1) receive a message, (2) update their state, and then (optionally) (3) send a message. The clients and virtual nodes communicate via (mostly-reliable) local broadcast.

Much like the emulation algorithms presented in this thesis, the mobile nodes replicate each nearby virtual node. The emulation is implemented by three components: (1) the Consistency Manager, which attempts to ensure that each of the replicas has a consistent view of the virtual node's state; (2) the Leader Manager, which attempts to select a leader to manager the emulation; and (3) the Membership Manager, which facilitates the process of nodes' joining the emulation. Unlike the algorithms described in this thesis, the emulation is managed by a leader. The leader is responsible for sending messages on behalf of the virtual node, and in ensuring that nodes receive an up-to-date replica of the state when they join the emulation. The main algorithmic issues occur during transitions in leadership, for example, when a leader fails or leaves the emulation.

The emulator was implemented in Python and deployed on a testbed of HP iPAQ Pocked PCs. The iPAQs run linux and communicate via an 802.11 wireless adaptor. The devices were carried through the hallways of an office building, providing mobility. Each device determined its approximate location by querying the closest 802.11 access point. (The access points were used only to determine the device's location; all communication was performed in ad hoc mode among the mobile devices.)

12.2.2 Virtual Traffic Control

Using this virtual infrastructure platform, we built a simple traffic control application that implements a virtual traffic light at a busy intersection. Each user carries an HP iPAQ, which presents the user with a visual indication as to whether the light is green or red, i.e., whether she can safely proceed through the intersection. Depending on the number of users waiting in each direction, the virtual traffic light chooses one of the four directions and allows traffic to proceed, while indicating that the other users should wait. A screenshot of the application can be found in Figure 12-1.

While the virtual traffic light application itself is clearly quite simple, such applications can be non-trivial to develop in wireless ad hoc networks. In fact, the main difficult of a traffic light reduces to the problem of *mutual exclusion*, and there are many subtleties involved in implementing a *reliable*, fault-tolerant mutual exclusion protocol in a wireless ad hoc network. Yet in the context of a virtual node abstraction, the virtual traffic light application becomes quite simple, requiring fewer than 300 lines of Python code. Abbreviated Python code executing on the clients and virtual nodes can be found in Figure 12-2. (In both cases, we have removed the error checking and user-interface code for clarity and succinctness.)

12.3 Open Questions and Ongoing Research

In this section, we discuss some of the open questions raised by the research in this thesis, and some of the ongoing research that is beginning to address these questions. First, in Section 12.3.1 we discuss some of the applications that we have considered developing on the virtual infrastructure platform. We then discuss in Section 12.3.2 some of the issues and trade-offs that arise in implementing virtual infrastructure.

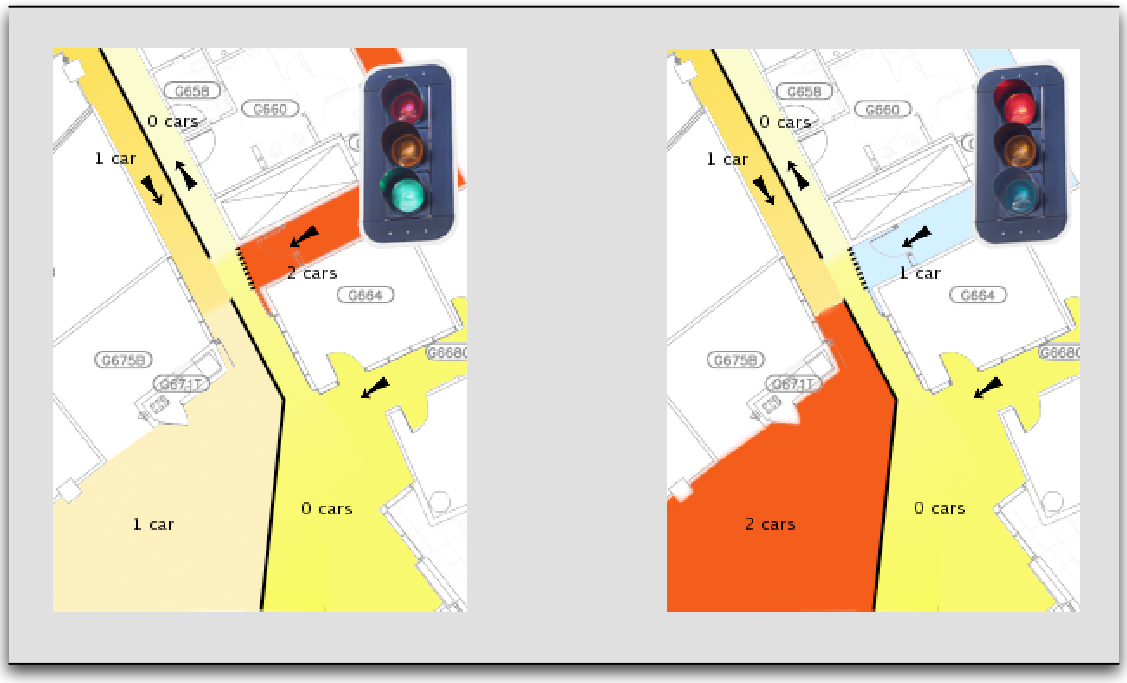

Figure 12-1: Two screenshots from the Virtual Traffic Light demo. The traffic light is controlling access to an intersection, with users arriving from four different regions. In the screenshot on the left, the user is in the top right region (dark shaded), and has a green light. In the screenshot on the right, the user is in the bottom right region (dark shaded), and has a red light. In each region, the interface indicates the number of users waiting to access the intersection.

Figure 12-2: Virtual Traffic Light: Virtual Node and Client Routines

Main routine executing on the virtual node:

```
def msgReceived(self, msg):
    replyArray = []
    if msg.msgtype(msg) == "UPDATE":
        self.UpdateVehInfo(msg)
        self.UpdateLightState()
        reply = "STATUS:" + self._statusSTR_()
        replyArray.append(reply)
    return replyArray
```

Main routine executing on the client:

```
while (self.running):
    transmit-msg = self.createUpdateMessage()
    self.sock.bcast(transmit-msg)
    while self.sock.select() :
        rcv-msg = self.sock.recv()
        self.UpdateUI(rcv-msg)
    if self.location.moved() :
        self.UpdateUI(self.location.position)
```

Finally, in Section 12.3.3 we discuss some of the interesting algorithmic questions that remain.

12.3.1 Virtual Infrastructure Applications

In this section, we briefly overview some of the applications that we have considered developing on the virtual infrastructure platform.

Traffic Control. Recall from Section 12.2.2 that we have already begun to implement a simple traffic control system on our prototype virtual infrastructure. As currently implemented, the system supports virtual traffic lights that can control the flow of traffic at specific intersections. There are many possible extensions to this simple system. For example, the virtual traffic lights at different intersections could coordinate to improve the flow of traffic. Moreover, the traffic control system could notify cars of upcoming traffic, and could help to route cars around congested regions.

Air Traffic Control. A natural extension to the problem of *highway* traffic control is the problem of *air traffic control*. Currently, airplane traffic follows a strict set of rules: flight plans must be filed in advance, deviation from flight plans is not allowed, and problems of congestion—for example, during arrival and departure from airports—are handled via human air traffic controllers. As air traffic has grown, however, this system has become both increasingly inefficient and increasingly susceptible to human error; we focus here on the problem of inefficiency. (Other work, e.g., [100], has considered solutions to minimize the problems of human error, particularly with respect to take-off and landing.) As an example, pilots are often requires to fly longer (and hence more expensive) routes, since routes are statically assigned prior to flight time; they are not allowed to switch to a shorter unoccupied route.

Thus the idea of *free flight* has been proposed: each airplane chooses its own route independently, and coordinates with nearby planes to avoid collisions. The challenge lies in facilitating this coordination in a manner that allows for pilots to choose efficient routes, while at the same time avoiding the potential for in-air collisions. It has been proposed in [15] that an air-traffic control system could be readily built on a virtual infrastructure platform. The three-dimensional space in which planes fly can be divided into regions, each of which is associated with a virtual node. The virtual nodes are responsible for coordinating the flight plans of airplanes within their region, and for coordinating with nearby virtual nodes to ensure a safe flight plan.

Emergency Management and Rescue Worker Coordination. A third coordination scenario involves the management of an emergency situation, for example, an earthquake, a power blackout, or a terrorist attack. In such situations, emergency workers must coordinate to rescue survivors, restore order, and otherwise ameliorate the problems—while simultaneously ensuring the safety of the rescuers themselves. Moreover, the catastrophic event itself may have destroyed the infrastructure required

for the rescue workers to coordinate. Thus virtual infrastructure seems ideal for facilitating the rescue effort.

There are a variety of other benefits provided by virtual infrastructure in this context. First, it is possible that some infrastructure may have survived the catastrophe; the virtual infrastructure implementation can take advantage of any remaining infrastructure, while ensuring that virtual infrastructure is available even in regions where the infrastructure has been destroyed. Moreover, this process occurs transparently, and the rescue workers need not be aware of whether the previous infrastructure remains or was destroyed. Second, in the future, rescue workers may make use of robots and other mechanical devices to avoid putting the rescuers themselves in dangerous situations. Lynch et al. [69] have already begun to study the problem of using virtual infrastructure to coordinate mobile robots. Third, virtual infrastructure is closely tied to geography, which may be particularly useful in coordinating rescue efforts. For example, rescue workers may want to associate certain information with particular locations, say, warning any rescue worker that approaches of some danger.

Monitoring and Data Collection. Sensor and mobile networks are often deployed for the purpose of environmental monitoring and data collection. For example, in the ZebraNet project (see, e.g., [43]), mobile devices were attached to zebras in Kenya and used to monitor the zebra behavior. Others have proposed the idea of using submergible and floating sensors to monitor aquatic environments. In these situations, little infrastructure is available, and it remains difficult to aggregate and collect the data that has been collected by these sensors. Virtual infrastructure may provide a simple method to accomplish these tasks.

Internet Routing. Finally, we have recently begun to consider whether virtual infrastructure can be used to facilitate internet routing in mobile networks. Specifically, virtual nodes can be used to execute simple IPv6 routers, transforming an unstructured wireless network into a well-defined network with a more traditional structure.

12.3.2 Trade-offs in Implementing Virtual Infrastructure

There are a variety of trade-offs that arise in implementing virtual infrastructure. In this section, we discuss some of these issues, along with some of the other challenges that arise in practical implementations.

Notice that some of the applications discussed in Section 12.3.1 require high reliability—for example, air traffic control—while others can tolerate more faults—for example, data collection. Similarly, some applications—again, particularly the control-related applications—require strong semantics, while others—for example, IPv6 routing—can tolerate weaker best-effort semantics. When implementing virtual infrastructure, there is an inherent trade-off between (1) the efficiency (and simplicity) of the implementation, (2) the semantics of the virtual devices, and (3) the reliability and availability of the virtual infrastructure.

In the first part of this thesis, we described virtual infrastructure layers that have strong semantics, and also provide good reliability. Specifically, the semantics of the virtual nodes emulated in Chapter 5 are defined simply in terms of traditional computing devices: the virtual nodes behave exactly as real computing devices in which each step of computation is executed in order. In addition, the virtual nodes are available as long as there are any clients in the virtual node region, with only minimal overlap required when clients enter and leave the region. In order to ensure these strong semantics and good reliability, the emulation sacrifices some efficiency: it depends on a totally-ordered broadcast service that guarantees reliable message delivery, which can be costly to implement; and it involves broadcasting redundant messages to coordinate the replicas.

In the second part of this thesis, we describe a virtual infrastructure emulation that also provides strong semantics, but provides more limited availability. Specifically, the virtual nodes are themselves prone to collisions and lost messages, and during times when the network is congested, the virtual nodes may not be available. As a result, the virtual infrastructure emulation presented in Chapter 10 is more efficient: the underlying broadcast service is less powerful, subject to congestion and lost messages; we can better bound the size of messages and the overhead of the emulation after stabilization; we can bound the length of time necessary to emulate a virtual round, even during unstable portions of the execution. Moreover, there may be further optimizations (as discussed in Section 12.3.3) that improve the performance of this emulation algorithm.

For some applications, such as implementing an IPv6 router, however, weaker semantics are sufficient. In particular, IP routing guarantees only best effort message delivery, and hence some unexpected behavior may be acceptable. Moreover, lower levels of reliability may be sufficient: if a virtual router resets itself on occasion, there is little effect on the resulting routing performance. Thus it seems likely that there are quite efficient emulation algorithms that are sufficient for implementing virtual routers. It remains an interesting question to consider virtual infrastructure with weaker semantics.

Another interesting trade-off that arises in the use of virtual infrastructure is the question of how much infrastructure is needed. Each piece of infrastructure has some cost—in terms of emulation—and provides some benefit—in terms of simplifying coordination among the clients.

For example, in the case of virtual traffic control, it may be necessary to have a virtual traffic light at each intersection, and it may optionally be useful to have some additional virtual nodes in between intersections to help coordinate the traffic and provide information to passing vehicles. Similarly, as discussed in Section 3.3, when using virtual nodes for routing or data collision, there is a trade-off between the number of virtual nodes and the performance of the routing and data collection: the more virtual nodes, the lower the latency for delivering messages or aggregated data, but the higher the cost.

Clearly, optimizing this trade-off depends significantly on the efficiency of the virtual infrastructure emulation—which itself depends on the desired semantics and reliability of the virtual entities.

Another interesting problem is the dynamic control of the amount of virtual infrastructure. This would allow the system to dynamically adjust this trade-off, changing the number of virtual nodes or objects, depending, perhaps, on the desired performance. In Chapter 6, the quorum-based protocol is reconfigurable, meaning that the set of virtual objects that the protocol uses may be dynamically changed; in this case, there is no reason to emulate the virtual objects that are currently unused. We examine one aspect of this question in [34], where (autonomous) virtual nodes may be generated and destroyed on the fly.

Finally, it remains an interesting open question—and an important area of ongoing research—to compare the performance and complexity of algorithms based on virtual infrastructure with the algorithms implemented directly on mobile ad hoc networks.

12.3.3 Algorithmic Open Questions

There are a variety of algorithmic questions involved in designing better and more efficient virtual infrastructure. In this section, we discuss several of these possibilities.

First, in the context of collision-prone networks, there are several optimizations that can further improve the performance of the emulation algorithm in Chapter 10. It seems likely that the overhead of the emulation can be further reduced: there is no need to include the client and virtual node messages (from the client and vn phases) in the ballot (send in the scheduled-ballot and unscheduled-ballot phase); in this case, the overhead of the emulation after stabilization—aside from nodes joining and leaving—is reduced to an additive constant per round. It also remains open as to whether it is possible to reduce the number of rounds required to implement one virtual round: as presented in this thesis, the length of a virtual round depends on the size of the schedule; it may be possible to implement one virtual round in a constant number of underlying rounds, independent of the size of the schedule. Finally, it remains an open question as to whether there is a simpler algorithm for emulating virtual infrastructure in collision-prone networks.

Second, there remain a variety of questions related to the problem of collision detection. In this thesis, we have assumed that mobile nodes have access to complete, eventually accurate collision detectors. In other work [23, 79], we have considered *majority*-complete collision detectors and *zero*-complete collision detectors. We believe that it is possible to adapt the emulation algorithm in Chapter 10 to tolerate majority-complete collision detection. By contrast, we believe that it is impossible to emulate virtual infrastructure with bounded round length using zero-complete collision detectors. It also remains an open question as to which collision detectors can be reliably built in wireless radio hardware. Perfect completeness seems unrealistic, while zero-complete collision detectors appear more plausible. It is possible, however, that all real collision detectors may occasionally fail, and hence it may be interesting to consider probabilistic completeness conditions. There remains much work in implementing and experimenting with various collision detectors.

Third, there also remain a variety of questions related to the problem of contention management. Most contention managers depend on backoff protocols to reduce contention, and there has been a large body of literature studying the problem of backoff.

It remains an interesting question to consider the performance of backoff protocols in the context of fault-prone collision detectors. Specifically, most backoff protocols assume that the mobile nodes can accurate detect collisions. When the collision detectors may be unreliable, the problem of contention management may be more involved. It also remains an open question whether we can emulate virtual infrastructure using even weaker contention managers.

Fourth, mobile nodes tend to have very stringent energy limitations. They have small batteries, and a limited lifetime that is further shortened by broadcast transmissions that are costly, in terms of energy. It is possible to take advantage of the redundancy already built into virtual infrastructure emulation to increase the lifetime of a wireless network. Specifically, the various replicas for a virtual node can share the energy cost of the emulation, potentially improving the lifetime of the entire network. Designing an energy-optimal virtual infrastructure emulation remains an open area of research, as does determining the potential energy costs and benefits of using virtual infrastructure.

Finally, the problem of malicious (or Byzantine) devices remains an area of ongoing research. Specifically, is it possible to emulate virtual infrastructure in the presence of non-cooperative devices? If the malicious devices themselves can cause arbitrary collisions, then clearly it is impossible for replicas to coordinate. Malicious devices, however, may have some limited capacity to disrupt the network, either due to energy limitations, or due to the existence of multiple disjoint frequencies that can be used by the replicas to communicate. In this context, it seems likely the replicas can communicate sufficiently to emulate virtual infrastructure. Further, however, it may desirable that the malicious devices learn nothing of the state of the virtual system. Cryptographic techniques related to secret sharing and secure function evaluation may enable such scenarios.

Bibliography

[1] IEEE 802.11. Wireless LAN MAC and physical layer specifications, June 1999.

[2] Ittai Abraham, Danny Dolev, and Dahlia Malkhi. LLS: a locality aware location service for mobile ad hoc networks. In *DIALM-POMC*, pages 75–84, 2004.

[3] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986.

[4] N. Alon, A. Bar-Noy, N. Linial, and D. Peleg. On the complexity of radio communication. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 274–285. ACM Press, 1989.

[5] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Hebrew University, 1995.

[6] Paul Attie, Nancy A. Lynch, and Sergio Rajsbaum. Boosting fault-tolerance in asynchronous message passing systems is impossible. Technical Report LCS-TR-877, MIT, 2002.

[7] Ziv Bar-Joseph, Idit Keidar, and Nancy A. Lynch. Early-delivery dynamic atomic broadcast. In *DISC*, pages 1–16, London, UK, 2002. Springer-Verlag.

[8] Ziv Bar-Joseph, Idit Keidar, and Nancy A. Lynch. Early-delivery dynamic atomic broadcast. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 1–16, London, UK, 2002. Springer-Verlag.

[9] R. Bar-Yehuda, O. Goldreich, and A. Itai. Efficient emulation of single-hop radio network with collision detection on multi-hop radio network with no collision detection. *Distributed Computing*, 5:67–71, 1991.

[10] R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.

[11] Jacob Beal. Persistent nodes for reliable memory in geographically local networks. Technical Report AIM-2003-11, MIT, 2003.

[12] Jacob Beal. A robust amorphous hierarchy from persistent nodes. In *Proc. of Communication Systems and Networks*, 2003.

[13] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFC 1349.

[14] Matthew Brown, Seth Gilbert, Nancy Lynch, Calvin Newport, Tina Nolte, and Michael Spindel. The virtual node layer: A programming abstraction for wireless sensor networks. In *International Workshop on Wireless Sensor Network Architecture*, April 2007.

[15] Matthew D. Brown. Air traffic control using virtual stationary automata. Master's thesis, MIT, September 2007.

[16] Costas Busch, Malik Magdon-Ismail, Fikret Sivrikaya, and Bulent Yener. Contention-free MAC protocols for wireless sensor networks. In Rachid Guerraoui, editor, *Proceedings of 18th International Symposium on Distributed Computing*, volume 3274/2004 of *Lecture Nodes in Computer Science*, pages 245–259, Oct 2004.

[17] T. Camp and Y. Liu. An adaptive mesh-based protocol for geocast routing. *Journal of Parallel and Distributed Computing: Special Issue on Mobile Ad-hoc Networking and Computing*, pages 196–213, 2002.

[18] Paz Carmi, Shlomi Dolev, Sariel Har-Peled, Matthew J. Katz, and Michael Segal. Geographic quorum system approximations. *Algorithmica*, 41(4):233–244, 2005.

[19] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[20] Ioannis Chatzigiannakis, Sotiris Nikoletseas, and Paul Spirakis. An efficient communication strategy for ad-hoc mobile networks. In *Proc. 15th International Symp. on Distributed Computing*, pages 285 – 299, 2001.

[21] B. Chlebus and D. Kowalski. A better wake-up in radio networks. *PODC*, pages 266–274, 2004.

[22] B. S. Chlebus, L. Gasieniec, A. Gibbons, A. Pelc, and W. Rytter. Deterministic broadcasting in ad hoc radio networks. *Distributed Computing*, 15(1):27–38, 2002.

[23] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *PODC*, 2005. To appear.

[24] G. V. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 237–246, New York, NY, USA, 1998. ACM Press.

[25] A. Clementi, A. Monti, and R. Silvestri. Round robin is optimal for fault-tolerant broadcasting on wireless networks. *J. Parallel Distributed Computing*, 64(1):89–96, 2004.

[26] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[27] J. Deng, P. K. Varshney, and Z. J. Haas. A new backoff algorithm for the IEEE 802.11 distributed coordination function. In *Communication Networks and Distributed Systems Modeling and Simulation*, January 2004.

[28] S. Dolev, E. Schiller, and J. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *Proc. of the 21st IEEE Symp. on Reliable Distributed Systems*, pages 70–79, 2002.

[29] Shlomi Dolev, Seth Gilbert, Limor Lahiani, Nancy A. Lynch, and Tina Nolte. Timed virtual stationary automata for mobile networks. In *Proceeding of the 43rd Allerton Conference on Communication, Control, and Computing*, September 2005. Invited.

[30] Shlomi Dolev, Seth Gilbert, Limor Lahiani, Nancy A. Lynch, and Tina Nolte. Timed virtual stationary automata for mobile networks. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, December 2005.

[31] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Elad Schiller, Alex A. Shvartsman, and Jennifer L. Welch. Virtual mobile nodes for mobile adhoc networks. In *Proceeding of the 18th International Conference on Distributed Computing (DISC)*, October 2004.

[32] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Alex A. Shvartsman, and Jennifer Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceeding of the 17th International Conference on Distributed Computing (DISC)*, volume 2848 of *Lecture Notes in Computer Science*, October 2003.

[33] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Alex A. Shvartsman, and Jennifer Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *Distributed Computing*, 18(2):125–155, November 2005.

[34] Shlomi Dolev, Seth Gilbert, Elad Schiller, Alex A. Shvartsman, and Jennifer L. Welch. Autonomous virtual mobile nodes. In *Proceeding of the 3rd Workshop on Foundations of Mobile Computing (DIAL-M-POMC)*, September 2005.

[35] Shlomi Dolev, Limor Lahiani, Nancy Lynch, and Tina Nolte. Self-stabilizing mobile node location management and message routing. In *SSS*, 2005.

[36] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. In *PODC*, pages 53–62, New York, NY, USA, 1997. ACM Press.

[37] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex behavior at scale: An experimental study of low-power wireless sensor networks. *UCLA Computer Science Technical Report UCLA/CSD-TR*, 2003.

[38] Seth Gilbert, Nancy A. Lynch, and Alex A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2003.

[39] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *2nd international conference on Embedded networked sensor systems*, 2004.

[40] Kostas P. Hatzis, George P. Pentaris, Paul G. Spirakis, Vasilis T. Tampakas, and Richard B. Tan. Fundamental control algorithms in mobile networks. In *Proc. of the 1st ACM Symp. on Parallel Algorithms and Architectures archive*, pages 251 – 260, Saint Malo, France, 1999.

[41] Ted Herman and Sebastien Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *Proceedings of the First International Workshop on Algorithmic Aspects of Wireless Sensor Networks*, volume 3121 of *Lecture Notes in Computer Science*, 2004.

[42] David B. Johnson, David A. Maltz, and Josh Broch. *DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[43] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS*, pages 96–107, 2002.

[44] Brad Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Mobile Computing and Networking*, pages 243–254, 2000.

[45] Dilsun Kirli Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed I/O automata. Technical Report MIT-LCS-TR-917a, MIT, apr 2004.

[46] Dilsun Kirli Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan & Claypool, 2006.

[47] Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit at no additional cost. In *PODS*, pages 245–254, 1995.

[48] Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 68–76, New York, NY, USA, 1996. ACM Press.

[49] Idit Keidar and Danny Dolev. Broadcast in the face of network partitions: Exploiting group communication for replication in partitionable networks. In D. Avresky, editor, *Dependable Network Computing*, chapter 3. Kluwer Academic Publications, 2000.

[50] Young Bae Ko and Nitin Vaidya. Geotora: A protocol for geocasting in mobile ad hoc networks. In *Proc. of the IEEE Intl. Conference on Network Protocols*, pages 240–249, November 2000.

[51] D. Kotz, C. Newport, R. S. Gray, J. Liu, Y. Yuan, and C. Elliott. Experimental evaluation of wireless simulation assumptions. In *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 78–82, 2004.

[52] D. Kotz, C. Newport, R. S. Gray, J. Liu, Y. Yuan, and C. Elliott. Experimental evaluation of wireless simulation assumptions. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, pages 78–82, October 2004.

[53] E. Kranakis, D. Krizanc, and A. Pelc. Fault-tolerant broadcasting in radio networks. In *ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 283–294, 1998.

[54] F. Kuhn, R. Wattenhofer, and A. Zollinger. Asymptotically optimal geometric mobile ad-hoc routing. In *Proceedings of the International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (Dial-M)*, pages 24–33, 2002.

[55] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Radio network clustering from scratch. In *12nd Annual European Symposium on Algorithms*, September 2004.

[56] Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Worst-case optimal and average-case efficient geometric ad-hoc routing. In *Proceedings of the 4th International Symposium on Mobile Ad-Hoc Networking and Computing*, 2003.

[57] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[58] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, 1978.

[59] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, july 1978.

[60] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[61] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[62] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[63] Butler W. Lampson. How to build a highly available system using consensus. In *WDAG*, pages 1–17, London, UK, 1996. Springer-Verlag.

[64] Philip Lewis and David Culler. Mate: a tiny virtual machine for sensor networks. In *International Conference on Mobile Computing and Networks*, 2000.

[65] J. Li, J. Jannotti, D.S.J. De Couto, D.R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of Mobicom*, 2000.

[66] Q. Li and D. Rus. Sending messages to mobile users in disconnected ad-hoc wireless networks. In *Proc. of the 6th MobiCom*, 2000.

[67] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.

[68] J. Luo and J. Hubaux. NASCENT: Network layer service for vicinity ad-hoc groups. In *SECON*, 2004.

[69] Nancy Lynch, Sayan Mitra, and Tina Nolte. Motion coordination using virtual nodes. Technical report, MIT CSAIL, 2005.

[70] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[71] Nancy A. Lynch, Roberto Segala, and Frits Vaandraager. Hybrid I/O automata. *Information and Computation*, 185(1), August 2003.

[72] Nancy A. Lynch, Roberto Segala, and Frits Vaandraager. Hybrid I/O automata. Technical Report LCS-TR-827d, MIT, August 2003.

[73] Nancy A. Lynch and Alex A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of the 16th Intl. Symp. on Distributed Computing*, pages 173–190, 2002.

[74] N. Malpani, J. Welch, and Vaidya. Leader election algorithms for mobile ad hoc networks. In *ACM DIAL-M*, 2000.

[75] M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. In *CONCUR*, pages 408–423, 1991.

[76] Koji Nakano and Stephan Olariu. A survey on leader election protocols for radio networks. In *ISPAN*, page 71. IEEE Computer Society, 2002.

[77] Badri Nath and Dragos Niculescu. Routing on a curve. *ACM SIGCOMM Computer Communication Review*, 33(1):150 – 160, January 2003.

[78] J. C. Navas and T. Imielinski. Geocast – geographic addressing and routing. In *Proc. of the 3rd MobiCom*, pages 66–76, 1997.

[79] Calvin Newport. Consensus and collision detectors in wireless ad hoc networks. Master's thesis, MIT, 2006.

[80] Ryan Newton, Arvind, and Matt Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Fourth International Conference on Information Processing in Sensor Networks*, 2005.

[81] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *First international workshop on data management for sensor networks*, 2004.

[82] Tina Nolte. *Virtual Stationary Timed Automata for Mobile Networks.* PhD thesis, MIT.

[83] Tina Nolte and Nancy Lynch. A virtual node-based tracking algorithm for mobile networks. In *Proceedings of International Conference on Distributed Computing Systems*, June 2007.

[84] V. Park and M.S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *IEEE INFOCOM*, April 1997.

[85] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on-demand distance vector routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[86] J. Polastre and D. Culler. Versatile low power media access for wireless sensor networks. In *Embedded Networked Sensor Systems*, 2004.

[87] Joseph Polastre, Robert Szewczyk, Cory Sharp, and David Culler. The mote revolution: Low power wireless sensor network devices. In *Hot Chips 16: A Symposium on High Performance Chips*, 2004.

[88] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[89] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[90] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Proc. of the 6th MobiCom*, pages 32–43, August 2000.

[91] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *First ACM Workshop on Wireless Sensor Networks and Applications*, 2002.

[92] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771 (Draft Standard), March 1995.

[93] Elizabeth M. Royer and Charles E. Perkins. Multicast operation of the ad hoc on-demand distance vector routing protocol. In *MobiCom*, pages 207–218, August 1999.

[94] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[95] Dale Skeen. Nonblocking commit protocols. In *SIGMOD Conference*, pages 133–142, 1981.

[96] Dale Skeen. A quorum-based commit protocol. In *Berkeley Workshop*, pages 69–80, 1982.

[97] Mike Spindel. Title unknown. Master's thesis, MIT.

[98] Qixiang Sun and Hector Garcia-Molina. Using ad-hoc inter-vehicle networks for regional alerts. Technical report, Stanford, oct 2004.

[99] Daniela Tulone. Is it possible to ensure strong data guarantees in highly mobile networks? In *Proceedings of the Fifth Annual Mediterranean Ad Hoc Networking Workshop*, pages 294–301, June 2006.

[100] Shinya Umeno and Nancy A. Lynch. Safety verification of an aircraft landing protocol: A refinement approach. In *Hybrid Systems: Computation and Control (HSCC)*, pages 557–572, 2007.

[101] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *In Proceedings ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[102] J. Walter, J. Welch, and N.H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 7(6):585–600, 2001.

[103] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *First USENIX Symposium on Networked Systems Design and Implementation*, 2004.

[104] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *International conference on mobile systems, applications, and services*, 2004.

[105] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of multihop routing in sensor networks. *In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems*, 2003.

[106] A. Woo, K. Whitehouse, F. Jiang, J. Polastre, and D. Culler. The shadowing phenomenon: implications of receiving during a collision. *Technical Report UCB//CSD-04-1313, UC Berkeley*, March 2004.

[107] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *In Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, 2002.

[108] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. *In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems*, 2003.

Appendix A

Background Material

In this chapter, we review some basic concepts and notation that are used throughout this thesis. More details can be found in [45, 46, 70].

A.1 Basic Notation

- Let $\perp$ be some distinguished symbol. For any set X, we refer to the set $X \cup \{\perp\}$ as $X_\perp$. For example, if *msgs* is a set of messages, then $msgs_\perp$ is the set of messages plus $\perp$.
- For any execution α, $\ell time(\alpha)$ refers to the last time in the execution.
- For any finite execution α, $\ell state(\alpha)$ refers to the last state in the execution.

A.2 Timed I/O Automata

In this section, we review some of the basic theory of timed automata. See [45, 46] for more detail. A timed I/O automaton (TIOA) is a tuple

$$\langle vars, states, start, actions, input, output, \delta, trajectories \rangle$$

that is input enabled (Definition A.2.4) and time-passage enabled (Definition A.2.5). In more detail, an I/O automaton consists of the following components:

- *vars*, a set of internal variables,
- $states \subseteq vals(vars)$, the set of states for the automaton,
- $start \subseteq states$, a subset of states in which the automaton may begin an execution,
- *actions*, a set of actions that name transitions,
- $input \subseteq actions$, a subset of actions designated as input actions,

- $output \subseteq actions$, a subset of actions designated as output actions; $output \cap input = \emptyset$,
- $\delta \subseteq states \times actions \times states$, a transition relation that determines how the actions modify the state,
- $trajectories \subseteq trajs(states)$, a set of trajectories defining how the analog variables evolve as time passes in between discrete actions. (The set $trajs(states)$ describes all possible trajectories over the set of states.) The trajectories must satisfy the following axioms:
  - (Existence of point trajectories.) If $x \in states$, then the point trajectory with domain $[0, 0]$ mapping 0 to x is in $trajectories$.
  - (Prefix closure.) For every $\tau \in trajectories$ and every $\tau' \leq \tau$, $\tau' \in trajectories$.
  - (Suffix closure.) For every $\tau \in trajectories$ and every $t \in domain(\tau)$, $\tau \trianglerighteq t \in trajectories$, i.e., every extension of trajectory τ is also in the set of trajectories.
  - (Concatenation closure.) Let $\tau_0, \tau_1, \tau_2, \ldots$ be a sequence of trajectories in $trajectories$ such that for each nonfinal index i, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$. Then $\tau_0 \frown \tau_1 \frown \cdots \in trajectories$, i.e., the concatenation of trajectories is also in the set of trajectories.

Definition A.2.1. *An action $a \in actions$ is an **external** action if it is an input or an output actions, that is, $a \in input \cup output$.*

Definition A.2.2. *An action $a \in actions$ is an **internal** action if it is not an external action.*

Definition A.2.3. *An action $a \in actions$ is **locally controlled** if it is an internal or an output action, that is, $a \notin input$.*

By assumption, a timed-I/O automaton is **input enabled**:

Definition A.2.4. *An automaton is **input enabled** if for each state $s \in states$ and $a \in input$, there exists a $s' \in states$ such that $\langle s, a, s' \rangle \in \delta$.*

Also by assumption, a timed-I/O automaton is **time-passage enabled**:

Definition A.2.5. *An automaton is **time-passage enabled** if for each state $s \in states$ there exists $\tau \in trajectories$ such that $\tau.fstate = s$ and either*

1. *$\tau.ltime = \infty$ or*
2. *τ is closed and some locally controlled action ℓ is enabled in $\tau.lstate$.*

One subclass of timed I/O automata is the set of asynchronous automata:

Definition A.2.6. *We say that a timed I/O automaton is* ***asynchronous*** *if all of its variables are discrete and if its set of trajectories consists of all constant-valued mappings from all possible time intervals.*

An asynchronous automaton can have any possible time-passage between any two discrete events.

We will be interested in automata that take only a finite number of steps in a finite interval of time:

Definition A.2.7. *We say that an execution fragment is* ***locally Zeno*** *if it has finite limit time and contains infinitely many locally controlled actions.*

Definition A.2.8. *A timed I/O automaton is* ***progressive*** *if it has no locally Zeno execution fragments.*

In general, a progressive automaton guarantees to only take a finite number of locally controlled actions in a bounded amount of time. While we will be interested in overall systems that guarantee this property, some subcomponents of the system may have a weaker property: they will only provide this guarantee when there are a finite number of input actions during the execution fragment in question. We therefore introduce here a new notion of progessivity:

Definition A.2.9. *We say that a timed I/O automaton is* ***internally progressive*** *if it has no locally Zeno execution fragments containing a finite number of input actions.*

Composition is a key operation that combines two automata into a single automaton. In order to compose two automata, they must be "compatible:"

Definition A.2.10. *We say that timed automata A_1 and A_2 are* ***compatible*** *if (1) the set of internal actions of A_1 is disjoint from the actions of A_2, (2) the set of internal actions of A_2 is disjoint from the actions of A_1, and (3) the variables of A_1 and A_2 are disjoint.*

When two internally progressive TIOAs are composed together, there is no guarantee with respect to the resulting automaton; it may in fact have Zeno executions. When one of the automata is progressive and the other is internally progressive, then we can state a theorem about their composition:

Theorem A.2.11. *If A_1 is a progressive TIOA, and A_2 is an internally-progressive TIOA, and A_1 and A_2 are compatible, then their composition is an internally-progressive TIOA.*

Proof. Assume that the composed automaton has a locally Zeno execution fragment. Therefore it contains infinitely many locally controlled actions. Notice that only a finite number of these locally controlled actions can be actions from automaton A_1, since A_1 is progressive. Thus the execution fragment must contain infinitely many locally controlled actions from A_2. Since A_2 is internally-progressive, this implies that the execution fragment has an infinite number of actions that are either input actions

to the composed automaton or output actions of A_1 that act as hidden inputs to A_2. Since only a finite number of the actions are output actions from A_1, this implies that there are an infinite number of input actions to the composed automaton, satisfying the requirements of an internally-progressive automaton. □

We will often want to talk about **restricting** an execution to a particular automaton, i.e., to a particular set of actions and variables. Given an execution α, a set of actions A, and a set of variables V, we denote by $\alpha|(A,V)$ the (A,V)-restriction of α.

Before defining (A,V)-restriction, we note the following notation: if f is a function and S is a set, we donte by $f|S$ the restriction of f to S, i.e., the function g with $dom(g) = (dom(f) \cap S)$ where $g(c) = f(c)$ for each $c \in dom(g)$; if f is a function whose range is a set of functions, and if S is a set, we indicate by $f \downarrow S$ the function g where $dom(g) = dom(f)$ and $g(c) = f(c)|S$ for each $c \in dom(g)$. (See [46], Section 2.)

For any finite execution α, the (A,V)-restriction of α is defined inductively as follows:

$$\alpha|(A,V) = \begin{cases} \tau \downarrow V & \alpha = \tau \\ \alpha'|(A,V).a.(\tau \downarrow V) & \alpha = (\alpha'.a.\tau), a \in A \\ \alpha'|(A,V).(\tau \downarrow V) & \alpha = (\alpha'.a.\tau), a \notin A \end{cases}$$

For infinite executions, then the (A,V)-restriction is simply the limit over all closed prefixes of α.

A.3 Atomic Objects

Atomic objects play an important role in the Virtual Object Model discussed in Part I of this thesis. In this section, we first define a *variable type* (Appendix A.3.1), which is used to specify an atomic object. As an example, we specify the variable type for a *read/write object*. We then formally describe an *atomic object* (Appendix A.3.2). Finally, we discuss what it means to *implement* an atomic object, and present a key theorem that we later use to prove that the virtual objects are, in fact, atomic objects (Appendix A.3.3).

A.3.1 Variable Types

An atomic object is specified by a *variable type* that describes its sequential behavior. The definition presented here is adapted from [70], Chapter 9, and [6]. A *variable type* τ consists of the following components:

- V, a set of legal values (i.e., states) for the object
- $v_0 \in V$, an initial value (i.e., state) for the object
- *invocations*, a set of invocations
- *responses*, a set of responses

- δ, the transition function, a mapping from:

$$(invocations \times V) \rightarrow (responses \times V)$$

  that maps every invocation and state to a response and a new state.

As an example, we present the variable type for a read/write object. (In Chapter 6, Section 6.2.1 we present a more sophisticated variable type.) A read/write object has the following variable type:

- V, an arbitrary set of values for the atomic object
- $v_0 \in V$, an arbitrary initial value
- $invocations = \{\mathsf{read}\} \cup \{\mathsf{write}(v) : v \in V\}$
- $responses = \{\mathsf{read\text{-}ack}(v) : v \in V\} \ \cup \ \{\mathsf{write\text{-}ack}\}$
- δ is defined as follows:
  - $\delta(\mathsf{read}, v) \rightarrow \langle \mathsf{read\text{-}ack}(v), v \rangle$
  - $\delta(\mathsf{write}(v'), v) \rightarrow \langle \mathsf{write\text{-}ack}, v' \rangle$

In programmatic style (used later in this thesis), the read/write variable type is expressed as the sequential specification presented in Figure A-1.

A.3.2 Canonical Atomic Objects

The variable type specifies the sequential behavior of an object; it does not indicate how the object behaves when it receives concurrent invocations. The canonical atomic object automaton indicates the legal behaviors of an atomic object of a given variable type. Figure A-2 presents the asynchronous automaton for an atomic object of type

$$\tau = \langle V, v_0, invocations, responses, \delta \rangle$$

with ports in Q, using the I/O automata formalism (see [70] for more details).

The input and output actions are of the form $\mathsf{invoke}(inv)_p$ and $\mathsf{respond}(resp)_p$, where $inv \in invocations$, $resp \in responses$, and $p \in Q$ is a port.

Each invocation and response takes place on a port, and each port can support only one operation at a time. Notice that the set of ports Q is a parameter of the canonical automaton; different instantiations will use different sets for Q. In some cases, Q may simply be I, the set of node identifiers, in which case each mobile node has one port on the atomic object. In other cases, Q may be $S = \mathbb{N}^{>0} \times OP \times I$, where OP is some set of operation identifiers (see Figure 2-1), giving each mobile node a countably infinite number of ports on the object, which allows each mobile node more concurrent access to the object. (We see in more detail why this is useful in Chapter 6.)

Figure A-1: Read/Write Sequential Specification.

```
1   State:
2       value, initially v_0
3
4   Operations:
5   read()
6       return read-ack(value)
7
8   write(new-value)
9       value ← new-value
10      return write-ack()
```

Figure A-2: Canonical Atomic Object Specification.
Object Type $\tau = \langle V, v_0, \mathit{invocations}, \mathit{responses}, \delta \rangle$ for the set Q of ports.

Signature:
  Input invoke$(inv)_p$, $inv \in \mathit{invocations}$, $p \in Q$

  Output respond$(resp)_p$, $resp \in \mathit{responses}$, $p \in Q$

  Internal perform$(inv, v, resp, v')_p$, $inv \in \mathit{invocations}$, $resp \in \mathit{responses}$, $v, v' \in V$, $p \in Q$

State:
  $val \in V$, a value, initially v_0
  inv-buffer, a set of pairs $\langle inv, p \rangle$, $inv \in \mathit{invocations}$, by port p, $p \in Q$, initially $\emptyset$
  resp-buffer, a set of pairs $\langle resp, p \rangle$, $resp \in \mathit{responses}$, to port p, $p \in Q$, initially $\emptyset$

Transitions:

Input invoke$(inv)_p$
Effect:
  $\mathit{inv\text{-}buffer} \leftarrow \mathit{inv\text{-}buffer} \cup \{\langle inv, p \rangle\}$

Output respond$(resp)_p$
Precondition:
  $\langle resp, p \rangle \in \mathit{resp\text{-}buffer}$
Effect:
  $\mathit{resp\text{-}buffer} \leftarrow \mathit{resp\text{-}buffer} - \{\langle resp, p \rangle\}$

Internal perform$(inv, v, resp, v')_p$
Precondition:
  $\langle inv, p \rangle \in \mathit{inv\text{-}buffer}$
  $v = val$
  $\delta(inv, v) = (resp, v')$
Effect:
  $val \leftarrow v'$
  $\mathit{inv\text{-}buffer} \leftarrow \mathit{inv\text{-}buffer} - \{\langle inv, p \rangle\}$
  $\mathit{resp\text{-}buffer} \leftarrow \mathit{resp\text{-}buffer} \cup \{\langle resp, p \rangle\}$

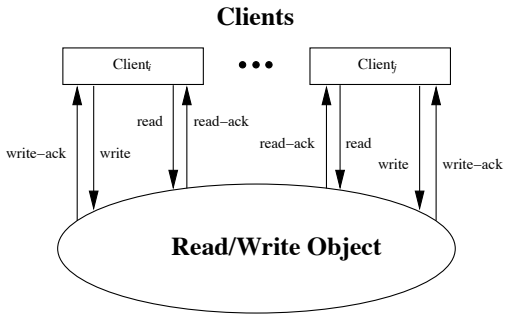

Figure A-3: Abstract Read/Write Object

Figure A-3 depicts the atomic object derived from the read/write variable type. In diagrams like Figure A-3, for clarity of presentation, instead of writing invoke(read) and the corresponding respond(read-ack), we write read and read-ack, as the direction of the arrows makes clear the action involved. We also omit the parameters to the invocations and responses.

Notice that the canonical automaton is not a distributed algorithm; it assumes centralized state that all the nodes can access. It does, however, support concurrent read and write invocations on different ports. The automaton simply performs the operations in some order. This is consistent with the usual notion that an atomic object serializes all of its operations.

Also notice that the canonical automaton is asynchronous, meaning that, formally, it is a timed I/O automaton in which there can be arbitrary time passage between any two discrete events.

The canonical automaton is presented with no liveness conditions. Often (as in [70]), there is an additional fail_p action, for each port in Q, and a "tasks" specification requires that as long as no fail_p action occurs, each invocation on p eventually leads to a response.

A.3.3 Implementing Canonical Objects

We next define what it means for an environment to be well-formed:

Definition A.3.1. *We say that a timed I/O automaton U is a* well-formed environment *for an atomic object if:*

1. *Its outputs are exactly the invocations of the object, and its inputs are exactly the responses of the object.*

2. *In every execution, for every port p, the automaton never performs two consecutive invocations on port p without an intervening response on port p.*

We then say that an automaton S *implements* the canonical (abstract) object, A, if:

1. S has the same input and output actions as A, the canonical object.

2. If U is a well-formed environment, then any trace of $S \times U$ is also a trace of $A \times U$. This implies that S preserves the well-formedness and safety guarantees of A .

(Informally, a trace of an automaton is the sequence of input and output actions occurring in an execution, along with appropriate timing information as to when the inputs and outputs occur. The symbol $\times$ represents the composition of automata, as defined in [70])

The most common way of showing that an algorithm implements an atomic object is to show that in every execution there exists a total ordering of the operations with certain properties. This ordering reflects the order in which the operations are

performed in the canonical automaton. The following theorem is a variant of Lemmas 13.10 and 13.16 in [70][1]. It has been updated to support timed I/O automata, though the argument remains essentially equivalent.

Theorem A.3.2. *Let A be an asynchronous[2] canonical atomic object of some variable type. Let S be a timed automaton with the same inputs and outputs as A. Let U be a timed automaton that is a well-formed environment with respect to S and A. Let α be a timed execution of $S \times A$ in which every operation completes, and assume that the following holds:*

Let Π be the set of operations in α. Assume that there exists a total ordering, $\prec$, on all the operations in Π with the following properties:

1. *The total order is consistent with the external order of invocations and responses. That is, if π completes before π' begins, then $\pi \prec \pi'$.*

2. *Fix some $\pi \in \Pi$. Let $inv_1, inv_2, \cdots, inv_k$ be the invocations of the operations preceding π in the total ordering, indexed according to the total ordering. Let $inv(\pi)$ be the invocation that initiates π, and $resp(\pi)$ be the response that concludes π.*

   Let v be the value of the variable type that results from starting with the initial value, v_0, and processing the following invocations:

   $$inv_1, inv_2, \ldots, inv_k \ .$$

   Then the response to operation π is consistent with the object being in state v. More formally, consider the event respond$(resp(\pi))$ *that occurs in α. Then for some value v' of the variable type,*

   $$\langle resp(\pi), v' \rangle = \delta(inv(\pi), v) \ .$$

Then there exists a timed execution γ of $A \times U$ such that $trace(\alpha) = trace(\gamma)$.

Proof (sketch). The proof is nearly identical to the untimed case, which is described in Lemmas 13.10 and 13.16 in [70]. In particular, we can consider the untimed sequence β of events in α, and show as in the case of an asynchronous system that the invocations and responses are atomic by identifying a serialization point for each operation. This implies that there exists an untimed sequence of events β' of A with the same external events as β and the internal perform events occurring in between the invoke and respond events for each operation in the order specified by the chosen serialization points.

[1] Lemmas 13.10 and 13.16 in [70] are presented for a setting with only finitely many ports, while here we allow there to be a countably infinite number of ports. However, nothing in the lemmas or their proofs depends on the number of ports being finite, so the results carry over for our setting.

[2] *Recall that by* asynchronous *we mean a timed automaton with only discrete variables that supports arbitrary time passage between any two discrete events.*

Finally, we transform β' into an execution of A by specifying the time at which each event occurs: each external event in β' occurs at the identical time as its matching event in α; each internal event in β' occurs at the earliest time that does not preceded any event prior to it in β'. The result is an execution γ of $A \times U$ such that $trace(\alpha) = trace(\gamma)$. □

Since U is an arbitrary environment, this implies that S implements A.

Property 1 requires that the total ordering be consistent with the real-world ordering of operations. Consider the example of a read/write atomic object: this property requires that if a write operation successfully completes and writes some value, *val*, then a later read operation cannot return an earlier value.

Property 2 requires that the total ordering of operations be consistent with the actual responses sent during the execution, since the total ordering is supposed to represent the order in which operations appear to happen. Consider again the example of a read/write atomic object: Property 2 guarantees that if, in the real execution, a read operation returns some value, *val*, then the closest preceding write operation (in the total order) must write that same value *val*.

The proof of Theorem A.3.2 is similar to that of Lemma 13.16 in [70]:

Proof (sketch). The proof involves choosing a serialization point for each operation: the earliest point after which the operation has begun and every operation preceding it in the total order has begun, where ties are ordered consistently with the total order. Property 1 ensures that the serialization point occurs before the operation completes and Property 2 ensures that the serialized execution has the same responses as the real execution. □

In the case of a read/write atomic object, it is necessary to determine only a partial ordering of the operations. The following theorem, then, is the analogue of Theorem A.3.2, and is proved in [70], Lemmas 13.10 and Lemma 13.16[3]:

Theorem A.3.3. *Let A be a canonical atomic read/write object (i.e., an object of the variable type presented in Figure A-1), and assume that S is an automaton with the same inputs and outputs as A, and that U is any well-formed environment. For every execution α of $S \circ U$ in which every operation completes, assume that the following holds:*

Let Π be the set of operations in α. Assume that there exists a partial ordering, $\prec$, on all the operations in Π with the following properties:

1. *All write operations are totally ordered, and every read operation is ordered with respect to all the writes.*

2. *The partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.*

[3]In [70], a fourth property is included, assuming that each operation is preceded by only finitely many other operation. This is unnecessary, as it is implied by Property 2.

3. *Every read operation that is ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns* v_0.

Then $traces(S \circ U) \subseteq traces(A \circ U)$.

Again, since U is an arbitrary environment, this implies that S implements A.

Index