A One-Round Algorithm for Virtually Synchronous Group Communication in Wide Area Networks

by

Roger I Khazan

M.S. in Electrical Engineering and Computer Science, MIT (1998)
B.A., Computer Science and Mathematics, Brandeis University (1996)

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2002

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

Certified by . . . .
Doctor Idit Keidar
Postdoctoral Research Associate, MIT
Senior Lecturer, The Technion – Israel Institute of Technology
Thesis Supervisor

Accepted by . . . .
Professor Arthur C. Smith
Chairman, Department Committee on Graduate Theses, MIT

A One-Round Algorithm for Virtually Synchronous Group Communication in Wide Area Networks

by

Roger I Khazan

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2002, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Group communication services, and especially those that implement Virtual Synchrony semantics, are powerful middleware systems that facilitate the development of fault-tolerant distributed applications.

In this thesis, we present a high quality, theoretical design of a group communication service that implements Virtual Synchrony semantics and is aimed for deployment in wide-area networks (WANs). The design features a novel algorithm for implementing Virtual Synchrony semantics; the algorithm is more appropriate for WANs than the existing solutions because it involves fewer rounds of communication and operates in a scalable WAN-oriented architecture. The high quality of the design refers to the level of formality and rigor at which it is done: The design includes formal and precise specifications, algorithms, correctness proofs, and performance analyses.

We develop the necessary supporting theory and methodology required for producing and evaluating this design. In particular, we develop a formal, inheritance-based, methodology that supports incremental construction of specifications, models, and proofs. This methodology helps us manage the complexity of the design and makes it evident which part of the algorithm implements which property of the system. We also develop new, formal approaches in the area of performance evaluation.

Thesis Supervisor: Professor Nancy A. Lynch
Title: NEC Professor of Software Science and Engineering, MIT

Thesis Supervisor: Doctor Idit Keidar
Title: Postdoctoral Research Associate, MIT
Senior Lecturer, The Technion – Israel Institute of Technology

Acknowledgments

First, I would like to thank the following five people directly related to this dissertation:

- Nancy Lynch — my dissertation supervisor, research advisor, mentor, teacher, and research collaborator. Nancy, thank you for the six wonderful years in the TDS group, for your support, guidance, and encouragement throughout these years. I have learned a great deal from you, and I thank you for the opportunity to be your student.

- Idit Keidar — my dissertation supervisor, mentor, research collaborator, and (I hope this is not out of line) friend. Idit, thank you for the exciting times we had doing research together, and for being there for me every step of the way! I am honored to be your first Ph.D. student (of the many to come).

- Alex Shvartsman — my mentor, research collaborator, and teacher. Alex, I am very fortunate to have met you. Thank you for introducing me to Distributed Systems, steering me to join the TDS group, and for your guidance and wisdom throughout the years. I am grateful to you for always keeping my best interests at heart.

- Butler Lampson — my Ph.D. committee member and teacher. Prof. Lampson, thank you for keeping me on my toes with your tough questions; I have learned a lot trying to answer them. I also thank you for teaching me some of the key principles of designing sound computer systems (6.826). I hope this dissertation gives them justice.

- Alan Fekete — my Ph.D. committee member. Alan, thank you for always finding time to meet with me whenever you were in town, for helping me sort out the application example, and, very importantly, for helping me focus on the immediate goals.

I also thank my mentor, Martin Cohn, for his academic and professional advice.

This dissertation brings to a conclusion six wonderful years in which I have been a member of the TDS group at MIT. I thank my fellow graduate students: Carl Livadas, Roberto De Prisco, Victor Luchangco, Rui Fan, and Sayan Mitra for their companionship and insightful discussions on variety of topics. I thank Roberto for graduating early and bequeathing onto me the best office corner on the floor ⌣̈. I am especially grateful to Carl for his friendship. I know I can always count on you, Carl. (I hope you feel you can count on me too.)

I also would like to acknowledge several other people on our third floor at LCS: our administrative assistant, Joanne Talbot, for making TDS run so smoothly; our 'administrative mom', Be Blackburn, for chocolate (my mother would be very happy to know about this), other treats, and fun get-togethers; our system administrators, William Ang, Greg Shomo, Matt McKinnon, and Michael Vezza — these guys are amazing; and finally the members of the WWW Consortium (neighboring my office) for their espresso machine, spring water fountain, and steady supply of snacks. To all of you, Thank You!

In my free time I enjoyed the company of my Boston-area friends: Daniil Utin, Jane Silver, Mark and Anna Gurevich, Mike and Irina Fazio, and Eddie and Debbie Bruckner. Life would have been rather dull without you, guys.

Finally, I thank my dear parents, Dr. Leonard Khazan and Lana Brodsky; my grandpa, Isaak Brodsky; my grandma, Mara Khazan; and my parents-in-law, Dr. Simon and Sima Zaslavsky; as well as the rest of the meshpucha for their love and belief in my abilities. These have been the driving force behind my accomplishments. Special thanks to my Dad for keeping my wardrobe up-to-date, to my Mom for the monthly checks, and to Simon for helping edit my dissertation proposal.

Above all, I am indebted to my wife, Inna Zaslavsky Khazan, for her love, emotional support, and encouragement (not to mention helping me with the editing and the daily chauffeuring). Inna, you are and always will be my true inspiration.

Roger I. Khazan
Cambridge, Massachusetts

To my parents, Dr. Leonard Khazan and Lana Brodsky.

Contents

List of Figures

Chapter 1

Introduction

We present a high quality, theoretical design of a group communication service (GCS) that implements Virtual Synchrony semantics and is aimed for deployment in wide-area networks (WANs). Section 1.1, below, presents some basic background on GCSs and Virtual Synchrony; the section also explains the challenges that arise in WANs. After this introductory information, we summarize the key contributions made by our work; this is done in Section 1.2. Then, in Sections 1.3 and 1.4, we overview how each of these contributions is accomplished: Section 1.3 discusses different aspects of the design, in particular, how the design addresses the challenges outlined in Section 1.1 and what characteristics make it high quality. Section 1.4 overviews the supporting theory and methodology that we develop to help us produce the design.

1.1 Background and Motivation

Modern distributed applications often involve large groups of geographically distributed processes that interact by sending messages over an asynchronous fault-prone network. Many of these applications maintain a replicated state of some sort. In order for these applications to be correct, the replicas must remain mutually consistent throughout the execution of the application. For example, in an online game, the states of the game maintained by different players must be mutually consistent in order for the game to be meaningful to the players. Designing algorithms that maintain state consistency is difficult however: different application processes may perceive the execution of the application inconsistently because of asynchrony and failures. For example if Alice, Bob, and Carol are playing an online game, the following asymmetric scenario is possible: Alice and Bob perceive each other as alive and well, but they differ in the way they perceive Carol; one sees Carol as crashed or disconnected, while the other sees her as alive and well. Middleware systems that hide from the application some of the underlying inconsistencies and instead present them with a more consistent picture of the distributed execution facilitate development of distributed applications.

Group communication services, such as [6, 8, 93, 19], are examples of such middleware systems. They are particularly useful for building applications that require reliable multi-point to multi-point communication among a group (or groups) of processes. Examples of such applications are data replication (for example, [57, 7, 39, 63, 42], and [30] Ch. 7), highly-available servers (for example, [11]), and online games.

GCSs provide a notion of *group abstraction*, which allows application processes to easily organize themselves into multicast groups. Application processes can communicate with the members of a group by addressing messages to the group. The semantics of the group abstraction is such that different members of the group have consistent perceptions of the communication done in the group. The abstraction is typically implemented through the integration of two types of services: membership and reliable multicast.

Membership services maintain information about membership of groups. The membership of a group can change dynamically due to new processes joining and current members departing, failing, or disconnecting. The membership service tracks these changes and reports them to group members. The report given by the membership service to a member is called a *view*. It includes a unique identifier and a list of currently active and mutually connected members. Failures can partition a group into disconnected components of mutually connected members. Membership services strive to form and deliver the same views to all mutually connected members of the group. While this is not always possible, they typically succeed once network connectivity more or less stabilizes (see, for example, [58, 27]).

Reliable multicast services allow application processes to send messages to the entire membership of a group. GCSs guarantee that message delivery satisfies certain properties. For example, one property can be that messages sent by the same sender are delivered in the order in which they were sent. Different GCSs differ in the specific message delivery properties they provide, but most of them provide some variant of *Virtual Synchrony* semantics. We refer to a GCS providing such semantics as a *Virtually Synchronous GCS*, and to an algorithm implementing this semantics as a *Virtual Synchrony algorithm.*

Virtual Synchrony semantics specifies how GCSs integrate membership and reliable multicast services; in particular, they specify how message deliveries are synchronized with view deliveries. This synchronization is done in a way that simulates a "benign" world in which message delivery is reliable within each view. Many variants of Virtual Synchrony have been suggested (for example, [77, 43, 27, 19, 82, 12]). In addition to other properties, nearly all of them include a key property, called *Virtually-Synchronous Delivery*, which guarantees that *processes that receive the same pair of views from the GCS receive the same sets of messages in between receiving the views.* Henceforth, when we refer to Virtual Synchrony, we assume the semantics includes Virtually-Synchronous Delivery.

Figure 1.1 illustrates a typical interface between application clients and the underlying GCS. It shows three clients, Alice, Bob, and Carol, using a Virtually Synchronous GCS to communicate with each other. In addition to the send-deliver events, the interface also has view events: whenever a client's view changes, the GCS informs the client of the new view.

The following example illustrates how Alice, Bob, and Carol may exploit the Virtually-Synchronous Delivery property while playing an online game.

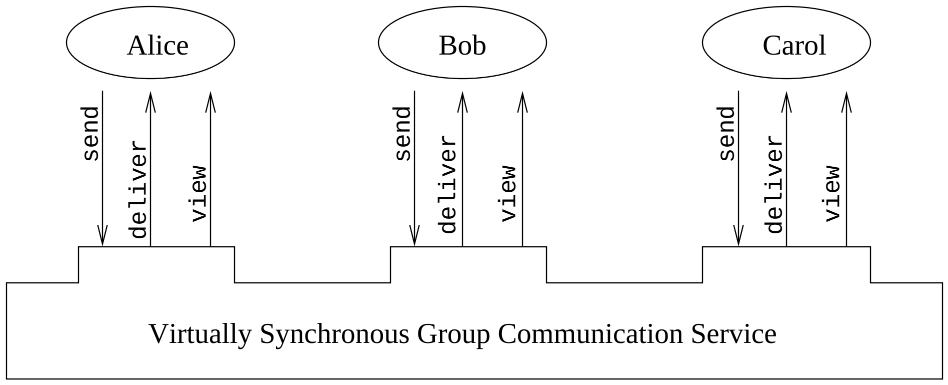


Figure 1.1: Typical interface between application clients and a Virtually Synchronous GCS. Arrows represent direction of the interaction.

Example 1.1.1 *Assume that, initially, each of the three clients, Alice, Bob, and Carol, is given a view* $\langle\{\textit{Alice, Bob, Carol}\}, 1\rangle$*, where* $\{\textit{Alice, Bob, Carol}\}$ *is a set of members and* 1 *is a view id. Then Carol disconnects, and Alice and Bob are given a new view* $\langle\{\textit{Alice, Bob}\}, 2\rangle$*. The Virtually-Synchronous Delivery property guarantees that both Alice and Bob receive the same messages before receiving the new view. In particular, if Bob receives a message from Carol before it receives the new view, then Alice also receives this message before the new view. Therefore, if Alice and Bob modify their game states only when they receive messages, they remain in consistent states and can safely continue playing the game after they receive the new view.*

In general, Virtually Synchronous GCSs are especially useful for building applications that maintain a replicated state of some sort using a variant of the well-known *state-machine/active replication* approach [66, 83]. With such approach, processes that maintain state replicas are organized into multicast groups. Actions that update the state are sent using a multicast primitive that delivers messages to different processes in the same order. When processes receive these actions, they apply them to their local replicas. Virtual Synchrony guarantees that processes that remain connected receive the same messages. This implies that processes that remain connected apply the same sequences of actions to their replicas. Hence, their replicas remain mutually consistent. Examples of GCS applications that use this technique are [4, 7, 57, 91, 42, 11].

Let us consider what is involved in implementing the Virtually-Synchronous Delivery property. Imagine that GCS processes are forming a new view because someone has disconnected from their current view. The GCS processes must make sure that they deliver the same messages to their application clients before delivering to them the new view. However, it may be the case that some of these GCS processes received messages that others did not. In the scenario illustrated in Example 1.1.1, the last messages from Carol may have reached the GCS process of only Bob, and not of Alice; Bob and Alice need to agree on whether or not to deliver these messages. To ensure such agreement, GCS processes invoke a *synchronization* protocol whenever a new view is forming; the protocol involves the processes exchanging special *synchronization messages*.

Designing correct and efficient algorithms that implement Virtual Synchrony is not trivial. Different GCS processes may perceive connectivity changes inconsistently. Since the desired synchronization depends on who the members of the new view are, the algorithm has to tolerate transient inconsistent views and cascading connectivity changes.

In particular, a Virtual Synchrony algorithm needs to know which synchronization messages sent by different processes pertain to the same view formation attempt. Existing algorithms, such as [43, 6, 82, 12, 47, 8], identify such synchronization messages by tagging them with a common identifier. Some initial communication is performed first, before synchronization messages are communicated, in order to agree upon a common identifier and to distribute it to the members of the forming view.

While a view is forming and a synchronization protocol is executing, there may be changes in connectivity that call for views with altogether different memberships. When such situations happen, existing Virtual Synchrony algorithms, for example [43, 47, 82, 12, 8], continue executing their current synchronization protocol to termination and then deliver to the application a view that does not reflect the already detected changes in connectivity. Afterwards, the algorithm is invoked anew to incorporate the new changes.

We refer to a view as *obsolete* [58] when it is delivered by a GCS even though the GCS already has information that the view's membership has changed. Obsolete views cause an overhead not just for the GCS, but also for applications. Since application processes do not know when the views delivered to them are obsolete, they handle such views just as they do any other view, for example by running state-synchronization protocols [57, 39, 63].

Even though the existing algorithms are relaxed about handling obsolete views, they perform well in typical local-area networks. Such networks have fairly stable connectivity and short message latency, so the synchronization protocols are invoked very infrequently and for very short periods of time.

WANs differ from LANs. WANs may have frequent message loss and changes in connectivity, as well as high and unpredictable message latency. These characteristics imply that, in order for a Virtual Synchrony algorithm to be appropriate for a WAN, it has to execute fewer communication rounds and respond to connectivity changes promptly, without wasting resources on handling obsolete memberships. Such improvements to the existing algorithms would reduce the periods of time when application processes are waiting for a new view to be delivered to them so they may resume their application-related communication in the new view.

1.2 Our Contributions

In this thesis, we present a novel high quality design of a GCS targeted for WANs. The design provides a variant of the Virtual Synchrony semantics that combines a core set of properties that is commonly provided by GCSs. This set captures the properties that are useful for facilitating implementations of typical GCS applications; it can also support implementations of other, stronger, GCS properties. The contributions made by our design are summarized in the following list:

- The design features a new algorithm for implementing Virtual Synchrony. The algorithm neither processes nor delivers views with obsolete memberships. Moreover, the synchronization protocol run by our algorithm involves just a single message exchange round among members of the new view. We are not aware of any other algorithm for implementing Virtual Synchrony that has these two features.

- The design demonstrates how to more effectively decouple the algorithm for achieving Virtual Synchrony from the algorithm for maintaining membership. As suggested in [9, 58], such efficient decoupling is important for providing scalable GCSs in WANs.

  Existing designs typically have a single algorithm handling both Virtual Synchrony and membership. The few designs that do employ two separate algorithms [82, 18] still have the two algorithms tightly coupled. In particular, the Virtual Synchrony algorithms control the membership algorithms: the membership algorithms are not allowed to incorporate newly joining members while the synchronization protocols are running. In addition to the control issue, such tight coupling requires two-directional communication between the two algorithms. In contrast to these two observations, our design allows the membership algorithm to freely change memberships of forming views at any time; the interaction between the membership and Virtual Synchrony algorithms is only in one direction, from the former to the latter, and it has low overhead. The decoupling is such that the synchronization protocol can execute in parallel with the view formation protocol.

- The design is well-documented and is carried at a high level of formality and rigor, much higher than that of most previous designs of Virtually Synchronous GCSs. The presented specifications of our GCS and its environment, description of the algorithm, and proof of correctness are all precise and formal. Our project is the first to use formal methods for modeling a Virtually Synchronous GCS and to provide an assertional proof of its correctness.

  We evaluate the performance claims that we make about our design by doing a formal analysis. In particular, we investigate how long it takes the GCS to recover and resume its normal operation after network events occur. We express our results as a collection of time-bounds depending on the specific types and timing of network events and on the behavior of underlying services used by the GCS.

In order to manage the complexity of the design, we develop new theory and methodology that facilitates incremental construction of formal specifications, algorithms, and, very importantly, proofs. In addition to making the design tractable, the use of this methodology makes it evident which part of the algorithm implements which property. This theory and methodology are not specific to our GCS design. They are general contributions to the field of formal modeling and verification, as well as software engineering. Our thesis contains a comprehensive description of this general approach; an overview of this approach is given in Section 1.4 below.

The thesis also contains general formalizations of the new approaches that we develop to evaluate the results of our work, in particular, new approaches to studying the performance characteristics of our design.

1.3 Design Overview

We now discuss each of the different aspects of our design in more detail.

Algorithm for Virtual Synchrony

The novelty of our algorithm for achieving Virtual Synchrony is concentrated in its synchronization protocol. Recall that this protocol is run among GCS processes in order for those that remain connected to agree upon a common set of messages each of them must deliver before moving into the new view. The protocol depends on a simple, yet powerful idea. Instead of using common identifiers to designate which synchronization messages pertain to the same view formation attempt, we use locally generated identifiers. These identifiers are then included as part of the formed views[1]. Once a view formation completes at a GCS process, the process knows which synchronization messages of other members to consider for the view – the messages tagged with the identifiers that are included in the view.

Example 1.3.1 *Consider a view* $\langle\{Alice, Bob, Carol\}, [4,3,7], 8\}\rangle$. *This view has membership* $\{Alice, Bob, Carol\}$*, vector of local identifiers* $[4,3,7]$*, and view identifier* 8. *When a GCS process forms this view, it uses the synchronization messages from Alice, Bob, and Carol tagged respectively with* 4*,* 3*, and* 7 *to decide on the set of messages it must deliver before delivering this view to its application. Thus, if Alice, Bob, and Carol form the same view, they use the same synchronization messages, and thus agree on which application messages each of them needs to deliver.*

The use of local identifiers eliminates the need to pre-agree on common identifiers and allows the synchronization protocol to complete in a single message exchange round. It also allows the algorithm to promptly and efficiently react to connectivity changes, without wasting resources on obsolete views. The protocol works correctly even if, because of network instability, GCS processes send multiple synchronization messages during the same synchronization protocol.

Decoupling of Virtual Synchrony and Membership

Our design decouples the algorithm for implementing Virtually Synchronous multicast from the algorithm for maintaining membership; see Figure 1.2. The membership algorithm handles generation of local identifiers and formation of views. The algorithm for implementing Virtually Synchronous multicast is run by GCS end-points; these end-points synchronize views and application messages to implement the Virtual Synchrony semantics. In particular, they handle multicast requests submitted by the application, deliver application messages and views back to the application, and run the synchronization protocol to synchronize processes that transition together into new views. The decoupling involves low-cost,

[1]A similar view structure is suggested in [82], for the purpose of not having concurrent views intersect.

one-directional communication from the membership to the Virtually Synchronous multicast algorithm. It also allows the synchronization protocol to execute in parallel with the view formation protocol.

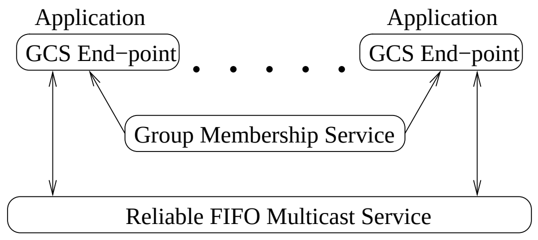


Figure 1.2: The decoupled architecture: GCS end-points provide the Virtual Synchrony semantics by integrating the output of a membership service and a reliable multicast service. Arrows represent interaction between GCS End-points and underlying services.

Efficient decoupling of membership and Virtually Synchronous multicast algorithms allows for an architecture in which the membership service is implemented by a small set of dedicated membership servers maintaining the membership information on behalf of a large set of clients. This architecture was proposed in [9, 58] for supporting scalable membership services in WANs. Our work extends this architecture by specifying how it can be used as a base for a Virtually Synchronous GCS. In particular, we present precise specifications of the interface and semantics that a membership service has to provide in order to be decoupled from the Virtually Synchronous multicast algorithm.

Our interface and membership service specifications allow for straightforward and efficient membership and Virtually Synchronous multicast algorithms. The Virtually Synchronous multicast algorithm presented in this thesis is an example of such an algorithm: the synchronization protocol requires a single message exchange round, which can occur in parallel with the formation of the view. The algorithm has been implemented (in C++) [92] using the scalable one-round membership algorithm of [58]. This membership algorithm was specifically tailored for our design, but other existing membership algorithms (for example, [35, 8]) can be also easily extended to provide the required interface and semantics.

Formal and Rigorous Approach

Our design has been carried out and is presented at a level more formal and rigorous than that of most previous designs of Virtually Synchronous GCSs. We precisely specify the properties satisfied by our Virtually Synchronous multicast algorithm, the external membership service, and the underlying communication substrate. We then give a formal description of the Virtually Synchronous multicast algorithm. The algorithm is accompanied by a careful, formal correctness proof and performance analysis. The safety properties are proved by using invariant assertions and simulation mappings; the liveness properties and performance claims are proved by using invariant assertions and careful operational arguments. We found this level of rigor to be important: in the process of specifying and verifying the algorithm, we uncovered several ambiguities and errors.

Previously, formal approaches were used to specify the semantics of Virtually Synchronous GCSs and to model and verify their applications, for example, in [26, 39, 31, 63, 32, 51]. Existing algorithms implementing Virtual Synchrony are modeled in pseudo-code and proven correct operationally. However, due to their size and complexity, such algorithms were not previously modeled using formal methods nor were they assertionally verified.

1.4 New Modeling Methodology

To manage the complexity of our design we have developed a formal methodology for incrementally constructing specifications, algorithms, and proofs [61, 62]. In addition to making the project tractable, the use of this construct makes clear which parts of the algorithm implement which property. The modularity of this approach facilitates further modifications and alterations of the design. Both the design and the new modeling methodology are developed in the framework of the I/O automaton formalism (see [72] and [71], Ch. 8).

The developed modeling methodology is not specific to our GCS design, but rather is a generic extension of the formal modeling and validation methods. We provide a framework for reuse of proofs analogous and complementary to the reuse provided by object-oriented software engineering methodologies. Specifically, we present a formal technique for incrementally constructing safety specifications (requirements), abstract algorithm descriptions, and *simulation proofs* that algorithms meet their specifications.

Our approach to incremental construction of system models (specifications and algorithms) relies on the following two modification constructs; these constructs produce a new model, called *child*, when they are applied to an existing model, called *parent*.

1. *Specialization:* We allow the child to *specialize* the parent by reusing its state in a read-only fashion, by adding new state components (which are allowed to be modified), and by constraining the set of behaviors of the parent. This corresponds to the *subtyping* view of inheritance [22]. We will show that any observable behavior of the child is *subsumed* (see [2]) by the possible behaviors of the parent, making our specialization analogous to *substitution inheritance* [22]. In particular, the child can be used anywhere the parent can be used.
2. *Signature-Extension:* A child can modify the signature of the parent's actions and introduce new actions not provided by the parent. When the actions of the child are renamed to their parent's names and the new actions are projected out, any behavior of the child is exactly as some behavior of the parent.

The combination of signature extension and specialization provides a powerful mechanism for incrementally constructing specifications and algorithms; this combination, which we call *specialized-extension*, corresponds to the *subclassing for extension* form of inheritance [22].

Consider the following two examples. The parent defines an unordered point-to-point messaging service with the send(msg) and recv(msg) interface. Specialization can be used to extend the parent with fifo ordering by restricting recv(msg) actions to deliver messages only according to the sending order. Subclassing for extension can be used to augment the messaging service with an acknowledgment mechanism: The parent's signature can be extended with ack(msg) actions, and then the parent can be specialized to handle acknowledgments by restricting ack(msg) actions to occur only after the corresponding messages were received by the recipient. The specialization and subclassing for extension constructs can be applied at both the specification level and the algorithm level in a way that preserves the relationship between the specification and the algorithm.

The main power of the provided methodology is in supporting incremental construction of simulation proofs when specifications and algorithms are constructed incrementally using the specialization and specialized-extension constructs. Simulation proofs are one of the most important techniques for proving properties of complex systems; such proofs exhibit a *simulation relation* (also known as *abstraction* or *refinement*) between a formal description of a system (algorithm) and its specification [1, 54, 71, 68].

Consider the example in Figure 1.4: Let S be a specification, and A an abstract algorithm description. Assume that we have proven that A *implements* S using a simulation relation $\mathtt{R_p}$. Assume further that we specialize the specification S, yielding a new *child* specification S'. At the same time, we specialize the algorithm A to construct an algorithm A' which supports the additional semantics required by S'.

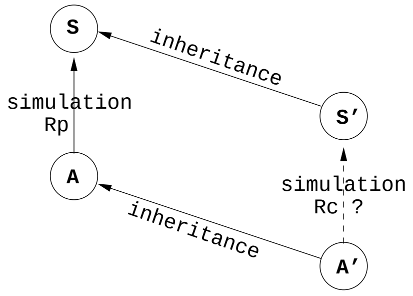


Figure 1.3: Algorithm $\mathtt{A}$ simulates specification $\mathtt{S}$ with $\mathtt{R_p}$. Can $\mathtt{R_p}$ be reused for building a simulation $\mathtt{R_c}$ from a child $\mathtt{A'}$ of $\mathtt{A}$ to a child $\mathtt{S'}$ of $\mathtt{S}$?

When proving that A' implements S', we would like to rely on the fact that we have already proven that A implements S, and to avoid the need to repeat the same reasoning. We would like to reason only about the new features introduced by S' and A'. The proof reuse theorem provides the means for incrementally building simulation proofs in this manner.

We present the formalism of our methodology in the context of the I/O automaton model [71, 72], but the methodology is more general than that particular model. We believe that the essence of our approach is applicable to any state-transition formal model, such as TLA [65], UNITY [76], or various forms of process algebra [53, 75].

Inheritance, as a means for modular system design, has been a subject of extensive research for decades. Many researchers employed formal methods to define various inheritance constructs and study their properties [2, 16, 28, 33, 49, 56, 70, 69, 79, 88, 87, 86, 13, 76, 48]. Our distinguishing contribution is a provision of a formal framework that allows simulation proofs to be constructed incrementally when inheritance is applied at two levels: specification and algorithm. Thus, we extend the applicability of inheritance from the realm of incremental system design to the realm of incremental system verification.

1.5 Roadmap

The dissertation is organized into two parts. The first part is comprised of two chapters: Chapter 2 reviews formal model, proof techniques, and notation. Chapter 3 presents the incremental modeling and verification formalism that we have developed; the presentation is based on the material published in [61, 62]. We employ this formalism for modeling and verifying our GCS design.

The second part covers our design of the Virtually Synchronous Group Communication service; it constitutes the main part of the dissertation. Chapters 4 – 8 are based on the material published in [59, 60].

In Chapter 4 we present the client-server architecture of our GCS and formally specify the assumptions we make on the membership service and the underlying communication substrate. Chapter 5 contains precise specifications of the safety and liveness properties satisfied by our GCS.

The algorithm implementing Virtual Synchrony is then given in Chapter 6; it is accompanied by informal correctness arguments. A formal correctness proof that the algorithm of Chapter 6, when it operates in the environment specified in Chapter 4, satisfies the specifications of Chapter 5 is given in the subsequent two chapters: safety – in Chapter 7, and liveness – in Chapter 8.

In Chapter 9 we consider performance characteristics of our algorithm; we formalize and prove our claim that our Virtual Synchrony algorithm involves just a single message exchange round among members of the new view. This chapter builds upon the liveness proof of Chapter 8: The liveness proof establishes that, after the environment starts to behave well, our GCS system *eventually* does what it is supposed to do; the performance analysis quantifies what "eventually" means in terms of various parameters such as message latency and timing of network and membership events.

In Chapter 10 we illustrate the utility of our GCS system by describing a simple application that can be effectively built using GCS. The application implements a variant of a data service that allows a dynamic group of clients to access and modify a replicated data object. The application is prototypical of some collaborative computing applications, such as for example a shared white-board application.

Chapter 11 concludes this dissertation.

Part I

Formal Foundations

Chapter 2

Formal Model and Notation

In this chapter we review the formal model, proof techniques, and notation. Section 2.1 presents background on the I/O automaton model [72] and the precondition-effect notation used for describing I/O automata; the presentation is based on [71, Chapter 8]. In Section 2.2 we describe the main techniques used to prove correctness of I/O automata: invariant assertions, hierarchical proofs, and simulation relations and refinement mappings, as well as the techniques of history and prophecy variables. The material of this section is closely based on [71, pages 216-228] and [68, pages 3,4, and 13]. The incremental modeling and verification formalism developed in the next chapter builds upon the formalism and examples presented here.

2.1 I/O Automaton Model

In the I/O automaton model, a system component is described as a state-machine, called an *I/O automaton*. The transitions of the automaton are associated with named actions, classified as *input*, *output* and *internal*. Input and output actions model the component's interaction with other components, while internal actions are externally unobservable. Note that an action can be either an input or an output, but not both; a function call that returns a value can be modeled using two actions – an input and an output.

Formally, an I/O automaton `A` consists of: a signature `sig(A)`, consisting of input, output, and internal actions; a set of states `states(A)`; a set of start states `start(A)`; a state-transition relation `trans(A)` — a subset of `states(A)` $\times$ `sig(A)` $\times$ `states(A)`; and a partition `tasks(A)` of output and internal actions into *tasks*. Tasks are used for defining fairness conditions. The part of an automaton's signature consisting of the input and output actions is called the automaton's *external signature*; an action is *external* if it is either input or output, but not internal. Using a *composition operation*, one can construct complex automata consisting of "smaller" automata that interact via their input and output actions.

An action π is said to be *enabled* in a state `s` if the automaton has a transition of the form (`s`, π, `s'`); input actions are enabled in every state. An *execution fragment* of an automaton `A` is an alternating sequence of states and actions such that every successive triple of this sequence is an allowable transition; an empty step (s, ϵ, s) is also an execution fragment. An *execution* is an execution fragment that begins with a start state. An infinite execution is *fair* if, for each task, it either contains infinitely many actions from this task or infinitely many occurrences of states in which no action from this task is enabled; a finite execution is *fair* if no action is enabled in its final state. The *trace* of an execution α of A, denoted by trace(α), is a subsequence of α consisting of all the external actions in α. We denote the set of executions of A by execs(A), and the set of traces of A by traces(A). When reasoning about an automaton, we are only interested in its externally-observable behavior as reflected in its traces.

The *composition operation* defines how automata interact via their input and output actions: It matches output and input actions with the same name in different component automata; when a component automaton performs a step involving an output action, so do all components that have this action as an input one. The result of composing an output action with an input action is classified as an output to allow for future compositions with other automata. A *hide* operator can re-classify an output action as an internal one to prevent it from being externally observable.

I/O automata are conveniently presented using the *precondition-effect* style. In this style, typed state variables with initial values specify the set of states and the start states. Transitions are grouped by action name, and are specified using a `pre:` block with preconditions (guards) on the states in which the action is enabled and an `eff:` block which specifies how the pre-state is modified. The effect is executed *atomically* to yield the post-state.

Example 2.1.1 *Figure 2.1 presents an I/O automaton,* UpSeq*, that prints nondecreasing sequences of integers. The automaton is expressed in the precondition-effect notation. The signature of* UpSeq *consists of output actions of the type* print(x)*, where* x *is an integer. The state of* UpSeq *consists of a single integer variable,* last*, initialized to an arbitrary value. The transitions of* UpSeq *specify that action* print(x)*, with a given* x*, is enabled in every state in which* $x \geq last$*, as enforced by the* `pre:` *statement; once* print(x) *occurs, the automaton moves into a state in which* $last = x$*, as specified by the* `eff:` *statement.*

```
AUTOMATON UpSeq
Signature:   Output print(x), x ∈ Integer
State:       last ∈ Integer, initially arbitrary
Transitions: OUTPUT print(x)
             pre: x ≥ last
             eff: last ← x
Tasks: {print(x) : x ∈ Integer}
```

Figure 2.1: Automaton UpSeq printing a nondecreasing sequence of integers.

The following is a sample infinite execution of UpSeq*, where square brackets represent states of* UpSeq*, that is, values of* last*:*

$$[3], \texttt{print(5)}, [5], \texttt{print(11)}, [11], \texttt{print(11)}, [11], \texttt{print(14)}, [14], \ldots$$

The trace of this execution is "`print(5), print(11), print(11), print(14), ...`*" In general, the set of traces of* `UpSeq` *is the set of all possible sequences printing nondecreasing integers, both finite and infinite. In this set, only the infinite sequences are fair.*

When reasoning about an automaton, we are interested in only its externally-observable behavior as reflected in its traces. There are two types of trace properties: *safety* and *liveness*. Safety properties usually specify that some particular bad thing never happens. In this thesis we specify safety properties using centralized, global, I/O automata that generate the allowed sets of traces; for such automata we do not specify task partitions. If every trace of the automaton modeling the system is also a trace of the specification automaton, then the system always does what is allowed by its specification. In this case, we say that the system automaton *satisfies*, or *implements*, the specification automaton. Liveness properties usually specify that some good thing eventually happens. An algorithm automaton satisfies a liveness property if the property holds in all of its *fair* traces.

2.2 Proof Techniques

Invariants

The most fundamental type of property to be proved about an automaton is an *invariant assertion*, or just *invariant*, for short. An invariant assertion of an automaton `A` is defined as any property that is true in every single reachable state of `A`.

Invariants are typically proved by induction on the number of steps in an execution leading to the state in question. While proving an inductive step, we consider only *critical actions*, which affect the state variables appearing in the invariant.

Hierarchical Modeling and Verification

One of the important proof strategies is based on a hierarchy of automata. This hierarchy represents a series of descriptions of a system or algorithm, at different levels of abstraction. The process of moving through the series of abstractions, from the highest level to the lowest level, is known as *successive refinement*. The top level may be nothing more than a problem specification written in the form of an automaton. The next level is typically a very abstract representation of the system: it may be centralized rather than distributed, or have actions with large granularity, or have simple but inefficient data structures. Lower levels in the hierarchy look more and more like the actual system or algorithm that will be used in practice: they may be more distributed, have actions with small granularity, and contain optimizations. Because of all this extra detail, lower levels in the hierarchy are usually harder to understand than the higher levels. The best way to prove properties of the lower-level automata is by relating these automata to automata at higher levels in the hierarchy, rather than by carrying out direct proofs from scratch.

By way of an example, regard automaton `UpSeq` of Example 2.1.1 to be a (safety) specification for the sequences of nondecreasing integers. In order for some automaton to satisfy this specification, any possible trace of this automaton has to be a trace of `UpSeq`. In the following example we present such an automaton.

Example 2.2.1 *Figure 2.2 contains an automaton,* `FibSeq`*, that prints, as its sole infinite trace, the suffix of the Fibonacci sequence that begins with "*$1, 2, \ldots$*". The Fibonacci sequence is an infinite sequence that begins with* 0 *and* 1*, and in which every further element is equal to the sum of the two preceding elements.*

The signature of the `FibSeq` *automaton is the same as that of* `UpSeq`*. The state of* `FibSeq` *consists of two integer variables,* `n` *and* `m`*, initialized to* `0` *and* `1`*, respectively. The transitions of* `FibSeq` *specify that action* `print(x)` *is enabled in every state in which* `x` *is equal to the sum of* `n` *and* `m`*, and that, once* `print(x)` *occurs, the automaton moves into a state in which* `n` *has the value that* `m` *has in the pre-state, and* `m` *has the value of* `x`*. Thus,* `FibSeq` *uses* `n` *and* `m` *to store the last two elements printed and to compute from them the next element to be printed.*

```
AUTOMATON FibSeq

Signature:   Output print(x), x ∈ Integer

State:       n ∈ Integer, initially 0
             m ∈ Integer, initially 1

Transitions: OUTPUT print(x)
             pre: x = n + m
             eff: n ← m
                  m ← x

Tasks: {print(x) : x ∈ Integer}
```

Figure 2.2: Automaton `FibSeq` printing the Fibonacci sequence.

The traces generated by `FibSeq` *are the infinite sequence of Fibonacci numbers,* "`print(1)`, `print(2)`, `print(3)`, `print(5)`, `print(8)`, `print(13)`, ... " *and all of its prefixes. Only the infinite sequence is fair.*

Every trace of `FibSeq` is clearly a trace of `UpSeq`. Therefore, automaton `FibSeq` satisfies, or implements, automaton `UpSeq`. But how can we prove this formally?

Simulation Relations and Refinements

A common technique for establishing that the set of traces of one automaton is included in the set of traces of another is to exhibit a so-called *simulation* relation (also known as an *abstraction relation*) that relates the states of the two automata and to prove that this relation satisfies certain conditions [1, 54, 71, 68], as defined below:

Definition 2.2.1 *Let* $\mathtt{A}$ *and* $\mathtt{S}$ *be two automata with the same external signature. A relation* $\mathtt{R} \subseteq \mathtt{states(A)} \times \mathtt{states(S)}$ *is a* simulation from $\mathtt{A}$ to $\mathtt{S}$ *if it satisfies the following two conditions:*

1. *If* $\mathtt{t}$ *is any initial state of* $\mathtt{A}$*, then there is an initial state* $\mathtt{s}$ *of* $\mathtt{S}$ *such that* $\mathtt{s} \in \mathtt{R(t)}$*, where we use notation* $\mathtt{R(t)}$ *as an abbreviation for* $\{\mathtt{s} : (\mathtt{t}, \mathtt{s}) \in \mathtt{R}\}$*.*
2. *If* $\mathtt{t}$ *and* $\mathtt{s} \in \mathtt{R(t)}$ *are reachable states of* $\mathtt{A}$ *and* $\mathtt{S}$ *respectively, and if* $(\mathtt{t}, \pi, \mathtt{t}')$ *is a step of* $\mathtt{A}$*, then there exists an execution fragment of* $\mathtt{S}$ *from* $\mathtt{s}$ *to some* $\mathtt{s}' \in \mathtt{R(t')}$*, having the same trace as* $(\mathtt{t}, \pi, \mathtt{t}')$*. The latter conditions means that the externally observable behavior of the execution fragment must be the same as that of the step.*

The two conditions above guarantee that whatever steps $\mathtt{A}$ executes, there is always a way for $\mathtt{S}$ to produce the same trace. The following theorem (from [71], Ch. 8) expresses this property formally:

Theorem 2.2.1 *If* $\mathtt{A}$ *and* $\mathtt{S}$ *are two automata with the same external signature and if* $\mathtt{R}$ *is a simulation from* $\mathtt{A}$ *to* $\mathtt{S}$ *then* $\mathtt{traces(A)} \subseteq \mathtt{traces(S)}$*.*

Any finite trace inclusion can be shown by using simulation relations, possibly after adding a special kind of variables, called "prophecy variables" [1, 85].

In some cases, a simulation relation is actually a function from states of the implementation automaton to the states of the specification automaton. In this case it is called a *simulation mapping* (also known as *abstraction function* or *refinement*). If $\mathtt{R}$ is a simulation function and $\mathtt{t}$ is a state of the implementation automaton, we use $\mathtt{R(t)}$ to denote the corresponding state of the specification automaton.

Example 2.2.2 *We illustrate the simulation technique by presenting a simulation function* $\mathtt{R}$ *from* $\mathtt{FibSeq}$ *to* $\mathtt{UpSeq}$*.* $\mathtt{R}$ *maps a state* $\mathtt{t}$ *of* $\mathtt{FibSeq}$ *to the state* $\mathtt{s}$ *of* $\mathtt{UpSeq}$ *with* $\mathtt{s.last} = \mathtt{t.m}$*, where* $\mathtt{s.last}$ *denotes an instance of variable* $\mathtt{last}$ *in state* $\mathtt{s}$*, and* $\mathtt{t.m}$ *denotes* $\mathtt{m}$ *in state* $\mathtt{t}$*. We now argue that* $\mathtt{R}$ *satisfies Definition 2.2.1:*

1. *In the initial state* $\mathtt{t_0}$ *of* $\mathtt{FibSeq}$*,* $\mathtt{t_0.m} = \mathtt{1}$*; therefore* $\mathtt{R(t_0).last} = \mathtt{1}$*, which is a valid initial state of* $\mathtt{UpSeq}$*.*
2. *Consider a step* $(\mathtt{t}, \mathtt{print(x)}, \mathtt{t}')$ *of* $\mathtt{FibSeq}$*. We claim that* $(\mathtt{R(t)}, \mathtt{print(x)}, \mathtt{R(t')})$ *is a legal step of* $\mathtt{UpSeq}$*.*
   (a) *We show that, in state* $\mathtt{R(t)}$ *of* $\mathtt{UpSeq}$*,* $\mathtt{print(x)}$ *is enabled, that is, that its precondition,* $\mathtt{x} \geq \mathtt{R(t).last}$*, is satisfied. The fact that* $(\mathtt{t}, \mathtt{print(x)}, \mathtt{t}')$ *is a step of* $\mathtt{FibSeq}$ *implies that the precondition,* $\mathtt{x} = \mathtt{t.n} + \mathtt{t.m}$*, holds in state* $\mathtt{t}$*. Since* $\mathtt{R(t).last}$ *is equal to* $\mathtt{t.m}$ *by definition of* $\mathtt{R}$*,* $\mathtt{x} = \mathtt{t.n} + \mathtt{R(t).last}$*. Therefore,* $\mathtt{x} \geq \mathtt{R(t).last}$*, since* $\mathtt{t.n} \geq 0$*, as stated in the following invariant:*

Invariant 2.2.1 *In every reachable state* $\mathtt{t}$ *of* FibSeq, $\mathtt{t.n} \geq 0$ *and* $\mathtt{t.m} \geq 0$.
Proof: *The proposition is true in the initial state* $\mathtt{t_0}$*, since* $\mathtt{t_0.n} = 0$ *and* $\mathtt{t_0.m} = 1$*. The proposition is true in state* $\mathtt{t'}$*, after a step* $(\mathtt{t}, \mathtt{print(x)}, \mathtt{t'})$ *of* FibSeq*, assuming it is true in state* t*, since* $\mathtt{t'.n} = \mathtt{t.m} \geq 0$ *and* $\mathtt{t'.m} = \mathtt{t.n} + \mathtt{t.m} \geq 0$. ■

(b) After print(x) *occurs in state* R(t)*, the value of* last *in the resulting post-state* s′ *is* x *(see Figure 2.1). In state* t′*, the value of* m *is also* x *(see Figure 2.2). Hence, by definition of* R*,* s′ = R(t′)*.*

Therefore, R *is a simulation mapping from* FibSeq *to* UpSeq*, and, as implied by Theorem 2.2.1,* FibSeq *satisfies* UpSeq*.*

History and Prophecy Variables

Sometimes, however, even when the traces of one automaton, A, are the traces of another, S, it is not possible to give a refinement mapping from A to S. This may happen due to the following two generic reasons:

- The states of S may contain more information than the states of A.
- S may make some premature choices, which A makes later.

The situation when A has been optimized not to retain certain information that S maintains can be resolved by augmenting the state of A with additional components, called *history variables* (because they keep track of additional information about the history of execution), subject to the following constraints ([68, 73]):

1. Every initial state has at least one value for the history variables.
2. No existing step is disabled by the addition of predicates involving history variables.
3. A value assigned to an existing state component must not depend on the value of a history variable.

These constraints guarantee that the history variables simply record additional state information and do not otherwise affect the behavior exhibited by the automaton. If the automaton $\mathtt{A_{HV}}$ augmented with history variables can be shown to implement S by presenting a refinement mapping, it follows that the original automaton A without the history variables also implements S, because they have the same traces.

The situation when S is making a premature choice, which A makes later, can be resolved by augmenting A with a different sort of auxiliary variable, *prophecy variable*, which can look into the future just as history variable looks into the past ([68, 73]). A prophecy variable guesses in advance some non-deterministic choice that A is going to make later. The guess gives enough information to construct a refinement mapping to S (which is making the premature choice). For an added variable to be a prophecy variable, it must satisfy the following conditions:

1. Every state has at least one value for the prophecy variable.

2. No existing step is disabled *in the backward direction* by the new preconditions involving a prophecy variable. More precisely, for each step (t, π, t') there must be a state (t, p) and a p such that there is a step $((t, p), \pi, (t', p'))$.

3. A value assigned to an existing state component must not depend on the value of the prophecy variable.

4. If t is an initial state of `A` and (t, p) is a state of the `A` augmented with the prophecy variable, then it must be its initial state.

If these conditions are satisfied, the automaton augmented with the prophecy variable will have the same (finite) traces as the automaton without it. Therefore, if we can exhibit a refinement mapping from $\mathtt{A}_{\mathtt{PV}}$ to `S`, we know that the `A` implements `S`.

Chapter 3

Incremental Modeling and Verification

In this chapter we present the novel inheritance-based formalism that we have developed for incremental modeling and verification of systems; the presentation is based on the material published in [61, 62]. We rely on this formalism in Part 2 of this dissertation, where we present our design of the Virtually Synchronous GCS. The next section overviews the formalism at a level sufficient for skipping the rest of this chapter and moving on to Part 2.

The rest of the chapter contains thorough description of the formalism; it relies on the material and examples presented in Chapter 2. The organization of this chapter is as follows:

In Section 3.2, we formally define and study a construct that corresponds to the specialization form of inheritance. Then, in Section 3.3, we present a general theorem that enables incremental verification of systems that are modeled and specified incrementally using the specialization construct. This theorem provides the foundation for incremental construction of simulation proofs, and is the key contribution described in this part of the thesis. In Section 3.4, we extend the theory of incremental modeling and proof construction to the subclassing for extension form of inheritance: we give a formal definition of the signature extension construct and show how it can be used in conjunction with the specialization construct to achieve subclassing for extension; we then extend the proof-reuse theory presented in Section 3.3 to this situation.

Section 3.5 compares our results with related work; it shows that, while many other works (e.g., [2, 16, 28, 33, 49, 56, 70, 69, 79, 88, 87, 86]) have dealt formally with inheritance, the distinguishing contribution of our approach is the provision of a formal framework for applying inheritance to both system modeling and system verification. Section 3.6 concludes this chapter.

This chapter employs a simple running example to illustrate the use of the presented formalism. Part 2 of this thesis is an illustration of how this formalism can be employed in a real large-scale modeling and verification project.

3.1 Overview

In our inheritance-based formalism, a *child* automaton is specified as a modification of the parent automaton's code. When presenting a child we first specify a *signature extension* which consists of new actions, labeled new, and modified actions. A modified action is labeled with the name of the action which it modifies as follows: modifies `parent.action(parameters)`). We next specify the *state extension* consisting of new state variables added by the child. Finally, we describe the *transition restriction* which consists of new preconditions and effects added by the child to both new and modified actions. For modified actions, the preconditions and effects of the parent are appended to those added by the child. New effects added by the child are performed before the effects of the parent, all of them in a single atomic step. The child's effects are not allowed to modify state variables of the parent. This ensures that the set of traces of the child, when projected onto the parent's signature, is a subset of the parent's set of traces.

Inheritance allows us to reuse code and avoid redundancies. It also allows us to reuse proofs: Assume that an algorithm automaton A can simulate a specification automaton S, and let A' and S' be child automata of A and S, respectively. Then the Proof Extension theorem below (Theorem 3.4.4) asserts that in order to prove that A' can simulate S' it is sufficient to show that the restrictions added by A' are consistent with the restrictions S' places on S, and that the new functionality of A' can simulate new functionality of S'.

3.2 Specialization

We now present the *specialization* construct for creating a child automaton by specializing the parent automaton. This construct captures the notion of subtyping [22]. In the next section, we present the main technical contribution of this paper: a theorem that allows one to construct a simulation proof from a specialization of an algorithm to a specialization of its specification by extending the original simulation proof from the algorithm to its specification.

The specialization construct defined below operates on a parent automaton, and accepts three additional parameters: a *state extension* – the new state components, an *initial state extension* – the initial values of the new state components, and a *transition restriction* specifying how the child specializes the parent's transitions.

Definition 3.2.1 (Specialization) *Let* A *be an automaton;* N *be any set of states, called a* state extension*;* N_0 *be a non-empty subset of* N*, called an* initial state extension*; and* $TR \subseteq (\texttt{states}(A) \times N) \times \texttt{sig}(A) \times N$ *be a relation, called a* transition restriction.

Then $\texttt{specialize}(A)(N, N_0, TR)$ *defines the following automaton* A'*:*

- $\texttt{sig}(A') = \texttt{sig}(A)$;
- $\texttt{states}(A') = \texttt{states}(A) \times N$; *(we denote compound states using angle brackets:* $\langle \cdot, \cdot \rangle$*)*
- $\texttt{start}(A') = \texttt{start}(A) \times N_0$;

- $\texttt{trans(A}') = \{(\langle \texttt{t}_\texttt{p}, \texttt{t}_\texttt{n}\rangle, \pi, \langle \texttt{t}'_\texttt{p}, \texttt{t}'_\texttt{n}\rangle) : (\texttt{t}_\texttt{p}, \pi, \texttt{t}'_\texttt{p}) \in \texttt{trans(A)} \wedge (\langle \texttt{t}_\texttt{p}, \texttt{t}_\texttt{n}\rangle, \pi, \texttt{t}'_\texttt{n}) \in \texttt{TR}\}$, *where* $\langle \texttt{t}_\texttt{p}, \texttt{t}_\texttt{n}\rangle$ *denotes a state in* $\texttt{states(A}')$*.*

Notation 3.2.2 *If* $\texttt{A}' = \texttt{specialize(A)(N, N}_\texttt{0}\texttt{, TR)}$ *we use the following notation: Given* $\texttt{t} \in \texttt{states(A}')$*, we write* $\texttt{t}|_\texttt{p}$ *to denote its parent component and* $\texttt{t}|_\texttt{n}$ *to denote its new component. If* α *is an execution fragment of* $\texttt{A}'$*, then* $\alpha|_\texttt{p}$ *and* $\alpha|_\texttt{n}$ *denote sequences obtained by replacing each state* $\texttt{t}$ *in* α *with* $\texttt{t}|_\texttt{p}$ *and* $\texttt{t}|_\texttt{n}$*, respectively.*

In the precondition-effect notation, a transition restriction (TR) can be specified for each action π by (a) additional preconditions that a child places on π, and (b) additional effects that specify how the *new* state components are modified as a result of a child taking a step involving π. Note that these additional effects can rely on but cannot modify the parent's state components. The additional preconditions work in conjunction with the preconditions placed on π by the parent automaton, and the additional effects are executed before the parent's effects; thus, when the additional effects read parent state components, they observe their pre-state values. The transition restriction expressed in this style is the union of the following two sets:

- All triples of the form $(\texttt{t}, \pi, \texttt{t}|_\texttt{n})$ for which π is not mentioned in the code for $\texttt{A}'$, that is, for which $\texttt{A}'$ does not restrict transitions involving π. Note that the post-state $\texttt{t}|_\texttt{n}$ is the same as the new state component of the pre-state $\texttt{t}$.
- All triples $(\texttt{t}, \pi, \texttt{t}'_\texttt{n})$ for which state $\texttt{t}$ satisfies the new preconditions on π placed by $\texttt{A}'$, and state $\texttt{t}'_\texttt{n}$ is the result of applying π's new effects to $\texttt{t}$.

Example 3.2.1 *Figure 3.1 below illustrates the use of the specialization construct. It presents precondition-effect code for automaton* AccSeq*, which specializes automaton* UpSeq *of Figure 2.1 on page 28 to print only* accelerating *sequences, that is, sequences in which the differences between consecutive elements are nondecreasing (in addition to the sequence itself being nondecreasing).*

```
AUTOMATON AccSeq  SPECIALIZES UpSeq

State Extension:   diff ∈ Integer, initially arbitrary

Transitions Restriction:
                   OUTPUT print(x)
                   new pre: x - last ≥ diff
                   new eff: diff ← x - last
```

Figure 3.1: Automaton AccSeq printing accelerating sequences of integers.

AccSeq *extends the state of* UpSeq *with a new integer variable* diff *having an arbitrary initial value. This variable is used for storing the difference between the last pair of elements printed. The new precondition placed on* print(x) *states that* $\texttt{x} - \texttt{last}$ *has to be greater than or equal to* diff*; it works in conjunction with the precondition,* $\texttt{x} \geq \texttt{last}$*, of* print(x) *in* UpSeq*. The new effect updates* diff *to be the current difference,* $\texttt{x} - \texttt{last}$*; it occurs before the effect that updates* last *in* UpSeq*.*

As a result of the new precondition and effect, transitions of $\mathtt{UpSeq}$ *are restricted to only those in which* $\mathtt{diff}$ *is non-decreasing. Thus, the sample trace of* $\mathtt{UpSeq}$ *given in Example 2.1.1 is* not *a trace of* $\mathtt{AccSeq}$ *because* $(\mathtt{11} - \mathtt{11}) \not\geq (\mathtt{11} - \mathtt{5})$*, while that in Example 2.2.1 is.*

Our specialization construct is defined so that any behavior of a child is allowed by its parent. Theorem 3.2.1 below states this property formally: it says that (1) every execution α of a specialization $\mathtt{A}'$ of an automaton $\mathtt{A}$ is also an execution of $\mathtt{A}$ when the state extension of $\mathtt{A}'$ is projected out from α; and (2) every trace of $\mathtt{A}'$ is a trace of $\mathtt{A}$.

Theorem 3.2.1 *If* $\mathtt{A}'$ *is a specialization of automaton* $\mathtt{A}$*, then:*

1. $\alpha \in \mathtt{execs}(\mathtt{A}') \Rightarrow \alpha|_{\mathtt{p}} \in \mathtt{execs}(\mathtt{A})$.
2. $\beta \in \mathtt{traces}(\mathtt{A}') \Rightarrow \beta \in \mathtt{traces}(\mathtt{A})$.

Proof 3.2.1:

1. Let α be an execution of $\mathtt{A}'$, which, by definition of execution, means that α begins in some initial state $\mathtt{t_0}$ and that every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α is a transition of $\mathtt{A}'$. By Definition 3.2.1, $\mathtt{t_0}|_{\mathtt{p}}$ is an initial state of $\mathtt{A}$ and, for every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α, the triple $(\mathtt{t_i}|_{\mathtt{p}}, \pi, \mathtt{t_{i+1}}|_{\mathtt{p}})$ is a transition of $\mathtt{A}$. From this it follows that the sequence obtained by replacing each state $\mathtt{t}$ in α with $\mathtt{t}|_{\mathtt{p}}$ is an execution of $\mathtt{A}$. Since this sequence is $\alpha|_{\mathtt{p}}$, we conclude that $\alpha|_{\mathtt{p}}$ is an execution of $\mathtt{A}$.
2. Follows from Part 1 and the fact that $\mathtt{sig}(\mathtt{A}') = \mathtt{sig}(\mathtt{A})$.

■

As a consequence of part 2 of Theorem 3.2.1, we have the following corollary:

Corollary 3.2.1 *If automaton* $\mathtt{A}$ *satisfies automaton* $\mathtt{S}$ *in terms of trace inclusion, then a specialization* $\mathtt{A}'$ *of automaton* $\mathtt{A}$ *also satisfies* $\mathtt{S}$ *in terms of trace inclusion.*

Moreover, given a simulation relation $\mathtt{R_p}$ from $\mathtt{A}$ to $\mathtt{S}$, the same relation is a simulation from $\mathtt{A}'$ to $\mathtt{S}$, except for the obvious projection of the states of $\mathtt{A}'$ onto the states of $\mathtt{A}$.

Corollary 3.2.2 *If relation* $\mathtt{R_p}$ *is a simulation from* $\mathtt{A}$ *to* $\mathtt{S}$*, and* $\mathtt{A}'$ *is a specialization of* $\mathtt{A}$*, then relation* $\mathtt{R}'_{\mathtt{p}} = \{(\mathtt{t}, \mathtt{s}) \;:\; \mathtt{t} \in \mathtt{states}(\mathtt{A}') \;\wedge\; (\mathtt{t}|_{\mathtt{p}}, \mathtt{s}) \in \mathtt{R_p}\}$ *is a simulation from* $\mathtt{A}'$ *to* $\mathtt{S}$.

Many similar inheritance constructs, such as, for example, [70, 69, 33, 13] and superposition of [76], were defined and proven to satisfy properties similar to those of Theorem 3.2.1 and Corollary 3.2.1. However, these properties are *not enough* to address the situation illustrated in Figure 1.4, where we are interested in reusing and extending a proof that automaton $\mathtt{A}$ satisfies automaton $\mathtt{S}$ in order to prove that a specialization $\mathtt{A}'$ of $\mathtt{A}$ satisfies a specialization $\mathtt{S}'$ of $\mathtt{S}$. Indeed, from Theorem 3.2.1 and Corollary 3.2.1, we know only that $\mathtt{traces}(\mathtt{S}') \subseteq$

$\mathtt{traces(S)}$ and that $\mathtt{traces(A')} \subseteq \mathtt{traces(A)} \subseteq \mathtt{traces(S)}$; the solid arrows in Figure 1.4 correspond to these trace inclusions. But, we do not know whether $\mathtt{traces(A')} \subseteq \mathtt{traces(S')}$; this is what we would like to be able to show without having to repeat the reasoning used in showing that $\mathtt{traces(A)} \subseteq \mathtt{traces(S)}$. In the next section, we address this question by developing a general theorem that facilitates reuse of simulation proofs at the parent level for the construction of simulation proofs at the child level. The theorem pinpoints exactly which parts of the child-level proof follow from the parent-level proof (these are the parts reused), and which do not, and therefore still need to be done in order to complete the proof.

3.3 Incremental Proofs

We now present the main technical contribution of this paper — a general theorem that lays the foundation for incremental proof construction. Consider the situation illustrated in Figure 1.4, where $\mathtt{A'}$ and $\mathtt{S'}$ are specializations of automata $\mathtt{A}$ and $\mathtt{S}$ respectively. Given a simulation relation $\mathtt{R_p}$ from $\mathtt{A}$ to $\mathtt{S}$, Theorem 3.3.1 below states conditions for reusing and extending $\mathtt{R_p}$ to a simulation relation $\mathtt{R_c}$ from $\mathtt{A'}$ to $\mathtt{S'}$. Relation $\mathtt{R_c}$ has to relate every initial state of $\mathtt{A'}$ to some initial state extension of $\mathtt{S'}$, and it has to satisfy a step condition similar to the one in Definition 2.2.1, but only involving the transition restriction relation of $\mathtt{S'}$.

Theorem 3.3.1 *Let automaton* $\mathtt{A'}$ *be a specialization of automaton* $\mathtt{A}$*. Let automaton* $\mathtt{S'}$ *be a specialization of automaton* $\mathtt{S}$*, such that* $\mathtt{S'} = \mathtt{specialize(S)(N, N_0, TR)}$*. Assume that* $\mathtt{A}$ *and* $\mathtt{S}$ *have the same external signatures and that* $\mathtt{A}$ *implements* $\mathtt{S}$ *via a simulation relation* $\mathtt{R_p}$*.*

A relation $\mathtt{R_c} \subseteq \mathtt{states(A')} \times \mathtt{states(S')}$*, defined in terms of relation* $\mathtt{R_p}$ *and a new relation* $\mathtt{R_n} \subseteq \mathtt{states(A')} \times \mathtt{N}$ *as* $\{(\mathtt{t}, \mathtt{s}) : (\mathtt{t}|_\mathtt{p}, \mathtt{s}|_\mathtt{p}) \in \mathtt{R_p} \wedge (\mathtt{t}, \mathtt{s}|_\mathtt{n}) \in \mathtt{R_n}\}$*, is a simulation from* $\mathtt{A'}$ *to* $\mathtt{S'}$ *if* $\mathtt{R_c}$ *satisfies the following two conditions:*

1. *For every* $\mathtt{t} \in \mathtt{start(A')}$*, there exists a state* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$ *such that* $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$*.*
2. *If* $\mathtt{t}$ *is a reachable state of* $\mathtt{A'}$*,* $\mathtt{s}$ *is a reachable state of* $\mathtt{S'}$ *such that* $\mathtt{s}|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t}|_\mathtt{p})$ *and* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$*, and* $(\mathtt{t}, \pi, \mathtt{t'})$ *is a step of* $\mathtt{A'}$*, then there exists a finite sequence* α *of alternating states and actions of* $\mathtt{S'}$*, beginning from* $\mathtt{s}$ *and ending at some state* $\mathtt{s'}$*, and satisfying the following conditions:*[1]
   - *(a)* $\alpha|_\mathtt{p}$ *is an execution fragment of* $\mathtt{S}$*.*
   - *(b) For every step* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}})$ *in* α*,* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}|_\mathtt{n}) \in \mathtt{TR}$*.*
   - *(c)* $\mathtt{s'}|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t'}|_\mathtt{p})$*.*
   - *(d)* $\mathtt{s'}|_\mathtt{n} \in \mathtt{R_n(t')}$*.*
   - *(e)* α *has the same trace as* $(\mathtt{t}, \pi, \mathtt{t'})$*.*

[1] *Note that if we do not explicitly qualify a sequence as "an execution sequence" we mean that it is "a mathematical sequence", as for example sequence* α *here and also later in Notation 3.4.3 and Theorem 3.4.4.*

The theorem follows from Corollary 3.2.2 and Lemma 3.3.2 below. Recall that Corollary 3.2.2 defines a simulation relation $\mathtt{R}'_\mathtt{p}$ from $\mathtt{A}'$ to $\mathtt{S}$ in terms of the simulation relation $\mathtt{R}_\mathtt{p}$ from $\mathtt{A}$ to $\mathtt{S}$ (see Figure 3.2). The lemma considers how to construct a simulation relation

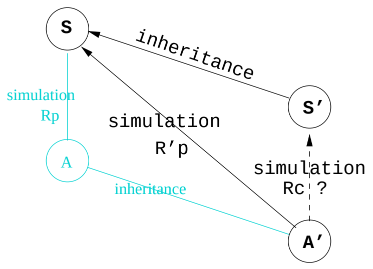


Figure 3.2: Intermediate step: Reusing $\mathtt{R}'_\mathtt{p}$ for building $\mathtt{R}_\mathtt{c}$.

$\mathtt{R}_\mathtt{c}$ from $\mathtt{A}'$ to $\mathtt{S}'$ from the simulation relation $\mathtt{R}'_\mathtt{p}$. This is a special case of Theorem 3.3.1, when $\mathtt{A}'$ is the same as $\mathtt{A}$. The statement of this lemma is almost identical to that of Theorem 3.3.1; the only difference is that, in Theorem 3.3.1, state $\mathtt{t}$ of $\mathtt{states}(\mathtt{A}')$ is projected onto its parent's state in order to be used in the simulation relation $\mathtt{R}_\mathtt{p}$. The lemma is stated in terms of $\mathtt{A}'$ and $\mathtt{R}'_\mathtt{p}$ in order to match the notation in Theorem 3.3.1.

Lemma 3.3.2 *Let* $\mathtt{S}$ *and* $\mathtt{A}'$ *be automata with the same external signatures, and let relation* $\mathtt{R}'_\mathtt{p}$ *be a simulation from* $\mathtt{A}'$ *to* $\mathtt{S}$*. Let* $\mathtt{S}' = \mathtt{specialize}(\mathtt{S})(\mathtt{N}, \mathtt{N_0}, \mathtt{TR})$*. A relation* $\mathtt{R}_\mathtt{c} \subseteq \mathtt{states}(\mathtt{A}') \times \mathtt{states}(\mathtt{S}')$*, defined in terms of relation* $\mathtt{R}'_\mathtt{p}$ *and a new relation* $\mathtt{R}_\mathtt{n} \subseteq \mathtt{states}(\mathtt{A}') \times \mathtt{N}$ *as* $\{(\mathtt{t}, \mathtt{s}) \ : \ (\mathtt{t}, \mathtt{s}|_\mathtt{p}) \in \mathtt{R}'_\mathtt{p} \wedge (\mathtt{t}, \mathtt{s}|_\mathtt{n}) \in \mathtt{R}_\mathtt{n}\}$*, is a simulation from* $\mathtt{A}'$ *to* $\mathtt{S}'$ *if* $\mathtt{R}_\mathtt{c}$ *satisfies the following two conditions:*

1. *For every* $\mathtt{t} \in \mathtt{start}(\mathtt{A}')$*, there exists a state* $\mathtt{s}|_\mathtt{n} \in \mathtt{R}_\mathtt{n}(\mathtt{t})$ *such that* $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$*.*
2. *If* $\mathtt{t}$ *is a reachable state of* $\mathtt{A}'$*,* $\mathtt{s}$ *is a reachable state of* $\mathtt{S}'$ *such that* $\mathtt{s}|_\mathtt{p} \in \mathtt{R}'_\mathtt{p}(\mathtt{t})$ *and* $\mathtt{s}|_\mathtt{n} \in \mathtt{R}_\mathtt{n}(\mathtt{t})$*, and* $(\mathtt{t}, \pi, \mathtt{t}')$ *is a step of* $\mathtt{A}'$*, then there exists a finite sequence* α *of alternating states and actions of* $\mathtt{S}'$*, beginning from* $\mathtt{s}$ *and ending at some state* $\mathtt{s}'$*, and satisfying the following conditions:*
   (a) $\alpha|_\mathtt{p}$ *is an execution fragment of* $\mathtt{S}$*.*
   (b) *For every step* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}})$ *in* α*,* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}|_\mathtt{n}) \in \mathtt{TR}$*.*
   (c) $\mathtt{s}'|_\mathtt{p} \in \mathtt{R}'_\mathtt{p}(\mathtt{t}')$*.*
   (d) $\mathtt{s}'|_\mathtt{n} \in \mathtt{R}_\mathtt{n}(\mathtt{t}')$*.*
   (e) α *has the same trace as* $(\mathtt{t}, \pi, \mathtt{t}')$*.*

Proof 3.3.2: We show that $\mathtt{R}_\mathtt{c}$ satisfies the two conditions of Definition 2.2.1:

1. Consider an initial state $\mathtt{t}$ of $\mathtt{A}'$. By the fact that $\mathtt{R}'_\mathtt{p}$ is a simulation, there must exist a state $\mathtt{s}|_\mathtt{p} \in \mathtt{R}'_\mathtt{p}(\mathtt{t})$ such that $\mathtt{s}|_\mathtt{p} \in \mathtt{start}(\mathtt{S})$. By condition 1 of the lemma, there must exist a state $\mathtt{s}|_\mathtt{n} \in \mathtt{R}_\mathtt{n}(\mathtt{t})$ such that $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$. Consider state $\mathtt{s} = \langle \mathtt{s}|_\mathtt{p},\ \mathtt{s}|_\mathtt{n} \rangle$. State $\mathtt{s}$ is in $\mathtt{R}_\mathtt{c}(\mathtt{t})$ by definition. Also, by Definition 3.2.1, $\mathtt{start}(\mathtt{S}') = \mathtt{start}(\mathtt{S}) \times \mathtt{N_0}$; therefore, $\mathtt{s} = \langle \mathtt{s}|_\mathtt{p},\ \mathtt{s}|_\mathtt{n} \rangle \in \mathtt{start}(\mathtt{S}) \times \mathtt{N_0} = \mathtt{start}(\mathtt{S}')$.

2. First, notice that the definitions of state $\mathtt{s}$ and relation $\mathtt{R_c}$ imply that $\mathtt{s} \in \mathtt{R_c}(\mathtt{t})$; also, notice that conditions 2c and 2d imply that $\mathtt{s}' \in \mathtt{R_c}(\mathtt{t}')$.

   Next, we show that α is an execution fragment of $\mathtt{S}'$ with the right trace. Indeed, every step of α is consistent with $\mathtt{trans}(\mathtt{S})$ (by 2a) and is consistent with $\mathtt{TR}$ (by 2b). Therefore, by definition of $\mathtt{trans}(\mathtt{S}')$ (Definition 3.2.1), every step of α is consistent with $\mathtt{trans}(\mathtt{S}')$. In other words, α is an execution fragment of $\mathtt{S}'$ that starts with state in $\mathtt{R_c}(\mathtt{t})$, ends with state in $\mathtt{R_c}(\mathtt{t}')$, and has the same trace as $(\mathtt{t}, \pi, \mathtt{t}')$ (by 2e).

■

We are now ready to prove Theorem 3.3.1:

Proof 3.3.1: Theorem 3.3.1 follows from Lemma 3.3.2 applied to automata $\mathtt{A}'$, $\mathtt{S}$, and $\mathtt{S}'$, with a simulation relation $\mathtt{R}'_\mathtt{p}$ from $\mathtt{A}'$ to $\mathtt{S}$ being $\{(\mathtt{t}, \mathtt{s}) : \mathtt{t} \in \mathtt{states}(\mathtt{A}') \wedge (\mathtt{t}|_\mathtt{p}, \mathtt{s}) \in \mathtt{R_p}\}$, as proved in Corollary 3.2.2. Each of the conditions in this theorem implies the corresponding condition in the lemma. ■

In practice, Theorem 3.3.1 (or Lemma 3.3.2) would be exploited as follows: The simulation proof between the parent automata already provides a corresponding execution fragment of the parent specification for every step of the parent algorithm. It is typically the case that the same execution fragment, padded with new state variables, corresponds to the same step at the child algorithm. Thus, conditions 2a, 2c, and 2e of Lemma 3.3.2 hold for this fragment. The only conditions that have to be verified are 2b, and 2d, that is, that every step of this execution fragment is consistent with the transition restriction TR placed on S by S′ and that the values of the new state variables of S′ in the final state of this execution are related to the post-state of the child algorithm. The verification of these two conditions may depend on some of the invariant assertions that were uncovered during the parent proof.

To exemplify how Theorem 3.3.1 and Lemma 3.3.2 would be exploited in practice, we use Lemma 3.3.2 to prove that `FibSeq` satisfies `AccSeq`, a specialization of `UpSeq`. Automata `UpSeq`, `FibSeq`, and `AccSeq` are simple enough to keep the example tractable, but they are arguably too simple to demonstrate the full utility of incremental proof construction. Part II of this dissertation serves as a large-scale example of the use of this framework in the design of a complex group communication service; Chapter 11 will comment on the specific contributions of the modeling and verification framework on making the design project tractable.

Example 3.3.1 *Recall that in Example 2.2.2 we presented a simulation mapping* R *from the states of* `FibSeq` *to the states of* `UpSeq`*. To construct a simulation mapping* R′ *from* `FibSeq` *to* `AccSeq`*, we extend* R *with the following mapping* $\mathtt{R_n}$ *that maps each state* t *of* `FibSeq` *to the state extension* s *of* `AccSeq` *such that*

$$\mathtt{s.diff} = \begin{cases} \mathtt{t.m} - \mathtt{t.n} & \textit{if}\ \mathtt{t.n} \neq 0 \\ 0 & \textit{otherwise} \end{cases}$$

In order to prove that R′ *is a simulation mapping we have to prove that it satisfies each of the conditions of Lemma 3.3.2.*

Condition 1 is satisfied because, if $\texttt{t}$ *is the initial state of* FibSeq*,* $\texttt{R}_\texttt{n}(\texttt{t}).\texttt{diff} = 0$ *is a valid initial value for the state extension of* AccSeq*.*

For Condition 2, the action correspondence is the same as in the simulation of UpSeq *by* FibSeq*: a step of* AccSeq *involving* print(x) *is simulated whenever* FibSeq *takes a step involving* print(x)*. Conditions 2a, 2c, and 2e are implied by the fact that* R *is a simulation relation from* FibSeq *to* UpSeq*; these were proven in Example 2.2.2. Thus, we only need to prove conditions 2b and 2d. Condition 2b requires the new precondition,* $\texttt{x} - \texttt{last} \geq \texttt{diff}$*, to be satisfied in state* $\texttt{R}'(\texttt{t})$*, provided the parent's precondition,* $\texttt{x} = \texttt{n} + \texttt{m}$*, holds in state* t*. Condition 2d requires the* $\texttt{R}_\texttt{n}$ *mapping to be preserved in the post-transition states of* FibSeq *and* AccSeq*; namely, the value of the new state variable* diff *in the post-transition state of* AccSeq *has to be the same as that of* $\texttt{R}_\texttt{n}(\texttt{t}').\texttt{diff}$*. Proving that these two conditions are satisfied involves reasoning only about how* AccSeq *specializes* UpSeq*.*

We now prove that conditions 2b and 2d hold. Consider a step $(\texttt{t}, \texttt{print(x)}, \texttt{t}')$ *of* FibSeq*; it implies that* $\texttt{x} = \texttt{t.n} + \texttt{t.m}$*, and that* $\texttt{t}'.\texttt{n} = \texttt{t.m}$ *and* $\texttt{t}'.\texttt{m} = \texttt{t.n} + \texttt{t.m}$*.*

- *Condition 2b: We have to show that the corresponding* print(x) *step of* AccSeq *is enabled in state* $\texttt{R}'(\texttt{t})$*, that is, that* $\texttt{x} - \texttt{R}'(\texttt{t}).\texttt{last} \geq \texttt{R}'(\texttt{t}).\texttt{diff}$*. By using the simulation mapping, we derive:* $\texttt{x} - \texttt{R}'(\texttt{t}).\texttt{last} = \texttt{x} - \texttt{R}(\texttt{t}).\texttt{last} = \texttt{x} - \texttt{t.m} = \texttt{t.n} + \texttt{t.m} - \texttt{t.m} = \texttt{t.n}$*. If* $\texttt{t.n} = 0$ *(as in the initial state of* FibSeq*), then, by definition of* $\texttt{R}'$ *and* $\texttt{R}_\texttt{n}$*,* $\texttt{R}'(\texttt{t}).\texttt{diff} = \texttt{R}_\texttt{n}(\texttt{t}).\texttt{diff} = 0$*, and we are done. Otherwise, if* $\texttt{t.n} \neq 0$*, then* $\texttt{R}'(\texttt{t}).\texttt{diff} = \texttt{R}_\texttt{n}(\texttt{t}).\texttt{diff} = \texttt{t.m} - \texttt{t.n}$*, and it remains to show that* $\texttt{t.n} \geq \texttt{t.m} - \texttt{t.n}$*. Invariant 3.3.2 below establishes this fact by relying on the following auxiliary invariant:*

  Invariant 3.3.1 *In every reachable state* t *of* FibSeq*,* $\texttt{t.m} \geq \texttt{t.n}$*.*
  Proof: *The proposition is true in the initial state* $\texttt{t}_0$*, since* $\texttt{t}_0.\texttt{n} = 0$ *and* $\texttt{t}_0.\texttt{m} = 1$*. The proposition is true in state* $\texttt{t}'$*, after a step* $(\texttt{t}, \texttt{print(x)}, \texttt{t}')$ *of* FibSeq*, since* $\texttt{t.n} \geq 0$ *(Invariant 2.2.1), and hence* $\texttt{t}'.\texttt{m} = \texttt{t.n} + \texttt{t.m} \geq \texttt{t.m} = \texttt{t}'.\texttt{n}$*.* ■

  Invariant 3.3.2 *In every reachable state* t *of* FibSeq*,* $\texttt{t.n} \geq \texttt{t.m} - \texttt{t.n}$*, if* $\texttt{t.n} \neq 0$*.*
  Proof: *The proposition is vacuously true in the initial state* $\texttt{t}_0$*, since* $\texttt{t}_0.\texttt{n} = 0$*. The proposition is true in state* $\texttt{t}'$*, after a step* $(\texttt{t}, \texttt{print(x)}, \texttt{t}')$ *of* FibSeq*, since* $\texttt{t.m} \geq \texttt{t.n}$ *(Invariant 3.3.1), and therefore* $\texttt{t}'.\texttt{n} = \texttt{t.m} \geq \texttt{t.n} = \texttt{t}'.\texttt{m} - \texttt{t.m} = \texttt{t}'.\texttt{m} - \texttt{t}'.\texttt{n}$*.* ■

- *Condition 2d: According to the code, the post-transition value of* diff *is* $\texttt{x} - \texttt{R}'(\texttt{t}).\texttt{last} = \texttt{t.n} = \texttt{t}'.\texttt{m} - \texttt{t.m} = \texttt{t}'.\texttt{m} - \texttt{t}'.\texttt{n}$*. If* $\texttt{t}'.\texttt{n} \neq 0$*, then* $\texttt{t}'.\texttt{m} - \texttt{t}'.\texttt{n} = \texttt{R}_\texttt{n}(\texttt{t}').\texttt{diff}$*, and we are done. Otherwise, if* $\texttt{t}'.\texttt{n} = 0$*,* $\texttt{R}_\texttt{n}(\texttt{t}').\texttt{diff} = 0$ *by definition, and the post-transition value of* diff *is also* 0*, since* $0 = \texttt{t}'.\texttt{n} = \texttt{t.m} \geq \texttt{t.n} \geq 0$ *(Invariants 2.2.1 and 3.3.1).*

Notice that, in verifying conditions 2b and 2d in Example 3.3.1, we relied on Invariant 2.2.1, which was stated and proven during the simulation proof from FibSeq to UpSeq. In general, knowing the invariant assertions that have been uncovered during the parent's proof can be helpful in extending that proof to the children.

3.4 Subclassing for Extension

In this section, we extend the theory of incremental modeling and proof construction to a new modification construct, called *specialized extension*; the construct is formulated in Definition 3.4.2 and the extended proof-reuse theorem appears as Theorem 3.4.4. This construct corresponds to the *subclassing for extension* form of inheritance [22], which is similar to specialization in that a child cannot override its parent's behavior, but it is more powerful than specialization in that a child can introduce new types of behavior through new actions, nonexistent in the parent.

We define a specialized extension of an automaton by first extending the parent automaton with new actions using a new construct, called *signature extension*, and then applying specialization of Section 3.2. The new actions introduced by signature extension are enabled in every state and do not modify the state; the subsequent specialization operation gives meaning to these new actions by restricting transitions involving the new actions, and, possibly, those involving parent's actions as well. The resulting automaton can interact with its environment through both the parent's actions and the new ones. Because new actions (even after being specialized) do not affect the parent's state, any trace of the child is indistinguishable from a trace of the parent when new actions are projected out from the trace.[2]

The signature extension construct, formulated in Definition 3.4.1, creates a new automaton by adding new actions to an existing automaton. The new automaton has an extended signature, but the same states and start states as the original automaton; the new state-transition relation is the same as the one in the original automaton, except that it includes additional transitions that relate every state to itself via new actions (i.e., new actions are enabled in every state, but do not modify the state); such transitions are called "stuttering" steps in [65].

In addition, the signature extension construct allows the new automaton to rename some or all of the original automaton's actions. The renaming is specified by a *signature-mapping* function that maps actions in the new signature to their counterparts in the parent signature. The function is allowed to be many-to-one, which means that the same action of the parent may be renamed into several actions of the child; this is useful because it allows a child to add new parameters to its parent's actions, and because instances of the same parent's action can be specialized differently under different names. The signature-mapping is onto, that is, every parent action has at least one corresponding action at the child. The function is defined only for actions inherited from the parent (renamed or not); it is undefined for new actions introduced by the signature extension. If π is such a new action and $\mathtt{f}$ is a signature-mapping, we write $\mathtt{f}(\pi) = \perp$ to denote the fact that π is not in the domain of definition of $\mathtt{f}$; $\perp$ is a assumed to be different from any action name.

Definition 3.4.1 (Signature Extension) *Let* $\mathtt{A}$ *be an automaton, and* $\mathtt{X}$ *be some signature.*

[2]Notice that this is stronger than behavioral subtyping of Liskov and Wing [69, 70], in which a trace of a child is required to be indistinguishable from a trace of its parent only when the trace does not contain actions introduced by the child (see Section 3.5).

Let $\mathtt{f}$ *be a partial function, called a* signature-mapping, *from* $\mathtt{X}$ *to* $\mathtt{sig(A)}$ *such that* $\mathtt{f}$ *is onto and preserves the classification of actions as "input", "output", and "internal"; the latter means that, if* $\mathtt{f}(\pi)$ *is defined, it is of the same classification as* π.[3]

Then, $\mathtt{extend(A)(X,f)}$ *is defined to be the following automaton* $\mathtt{A'}$*:*

- $\mathtt{sig(A')} = \mathtt{X}$,
- $\mathtt{states(A')} = \mathtt{states(A)}$,
- $\mathtt{start(A')} = \mathtt{start(A)}$, *and*
- $\mathtt{trans(A')} = \{(\mathtt{t},\pi,\mathtt{t'}) \in \mathtt{states(A')} \times \mathtt{sig(A')} \times \mathtt{states(A')}\ :$
  $((\mathtt{f}(\pi) = \bot) \wedge (\mathtt{t} = \mathtt{t'})) \vee ((\mathtt{f}(\pi) \in \mathtt{sig(A)}) \wedge ((\mathtt{t},\mathtt{f}(\pi),\mathtt{t'}) \in \mathtt{trans(A)}))\}$.

We say that $\mathtt{A'}$ is the *signature extension* of $\mathtt{A}$ with signature-mapping $\mathtt{f}$ if $\mathtt{A'}$ is such that $\mathtt{A'}$ $= \mathtt{extend(A)(sig(A'),f)}$ for some signature-mapping $\mathtt{f}$ from $\mathtt{sig(A')}$ to $\mathtt{sig(A)}$.

Having defined the signature extension construct, we now combine it with specialization to yield specialized extensions of automata.

Definition 3.4.2 (Specialized Extension) *Automaton* $\mathtt{A'}$ *is called a* specialized extension *of an automaton* $\mathtt{A}$ *if* $\mathtt{A'}$ *is a specialization of a signature extension of* $\mathtt{A}$.

In precondition-effect notation, we express a specialized extension $\mathtt{A'}$ of an automaton $\mathtt{A}$ by writing "$\mathtt{A'}$ MODIFIES $\mathtt{A}$" and then specifying the signature extension and the specialization parts of $\mathtt{A'}$. The signature extension part contains the new actions labeled with a keyword new, and the renamed actions labeled with their original names in $\mathtt{sig(A)}$, according to the signature-mapping; for example, if the signature mapping maps π of $\mathtt{A'}$ to σ of $\mathtt{A}$, we write "π modifies σ". We omit specifying the actions of $\mathtt{sig(A')}$ that are inherited from $\mathtt{A}$ without renaming. The specialization part contains the state extension and the transition restriction specifications, as described in Section 3.2 on page 37.

We now exemplify how the signature extension construct can be used in conjunction with the specialization construct to create specialized extensions.

Example 3.4.1 *Figure 3.3 presents automaton* FibSeq+ *that modifies automaton* FibSeq *to print each element of the Fibonacci sequence together with its sequence number. The signature-mapping specified by the Signature Extension clause maps actions* $\mathtt{print(i,x)}$ *where* $\mathtt{i} \in \mathtt{Integer}$ *to actions* $\mathtt{print(x)}$ *of* FibSeq*. Thus, for example, actions* $\mathtt{print(8,43)}$ *and* $\mathtt{print(23,43)}$ *of* FibSeq+ *are among those actions mapped to the* $\mathtt{print(43)}$ *action of* FibSeq*. Then, the specialization construct adds a new state variable,* last_i*, that keeps track of the sequence number of the last Fibonacci element printed; it also adds a new precondition and a new effect to the* $\mathtt{print(i,x)}$ *action to maintain* i *and* last_i *properly.*

[3]Signature-mapping is similar to *strong correspondence* of [94].

```
AUTOMATON FibSeq+  MODIFIES FibSeq
Signature Extension: Output print(i, x), i ∈  Integer modifies FibSeq.print(x)
New State:   last_i ∈ Integer, initially 0
Transition Restriction:
             OUTPUT print(i, x)
             new pre: i = last_i + 1
             new eff: last_i ← i
```

Figure 3.3: Automaton `FibSeq+` specifying enumerated Fibonacci sequences.

Notice that any execution α of `FibSeq+` is an execution of `FibSeq` when the newly added state variable, `last_i`, is projected out from every state in α and when every action in α is renamed according to the specified signature-mapping. Theorem 3.4.2 below formalizes this property in general. It follows from Theorem 3.2.1, which is a similar execution-inclusion property of specialization. This is because, modulo the signature-mapping, a signature extension of an automaton and the automaton itself have exactly the same executions and traces; we prove this result in Lemma 3.4.1 below.

Notation 3.4.3 *Let* $\mathtt{A}'$ *be a signature extension of* $\mathtt{A}$ *with a signature-mapping* $\mathtt{f}$*.*

If α *is a sequence of alternating states and actions of* $\mathtt{A}'$*, then* $\mathtt{f}(\alpha)$ *denotes the sequence obtained by replacing each action* π *in* α *with* $\mathtt{f}(\pi)$*, and then collapsing every triple of the form* $(\mathtt{t}, \perp, \mathtt{t})$ *to* $\mathtt{t}$*. Triples of the form* $(\mathtt{t}, \perp, \mathtt{t}')$ *where* $\mathtt{t}' \neq \mathtt{t}$ *are not collapsed; such triples are possible because* α *is not necessarily an execution sequence of* $\mathtt{A}'$*.*

Likewise, if β *is a sequence of external actions of* $\mathtt{A}'$*, then* $\mathtt{f}(\beta)$ *denotes a sequence obtained by replacing each action* π *in* β *with* $\mathtt{f}(\pi)$*, and then removing all the occurrences of* $\perp$*.*

Lemma 3.4.1 *Let automaton* $\mathtt{A}'$ *be a signature extension of* $\mathtt{A}$ *with a signature-mapping* $\mathtt{f}$*.*

Let α *be a sequence of alternating states and actions of* $\mathtt{A}'$ *and let* β *be a sequence of external actions of* $\mathtt{A}'$*. Then:*

1. $\alpha \in \mathtt{execs}(\mathtt{A}') \Leftrightarrow \mathtt{f}(\alpha) \in \mathtt{execs}(\mathtt{A})$.
2. $\beta \in \mathtt{traces}(\mathtt{A}') \Leftrightarrow \mathtt{f}(\beta) \in \mathtt{traces}(\mathtt{A})$.

Proof 3.4.1: The proof follows from Definition 3.4.1 and Notation 3.4.3.

1. $\Rightarrow$: Let α be an execution of $\mathtt{A}'$. By definition of execution, α begins in some initial state $\mathtt{t_0}$, and every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α is a transition of $\mathtt{A}'$. From this and Definition 3.4.1, $\mathtt{t_0}$ is an initial state of $\mathtt{A}$, and, for every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α, either $(\mathtt{t_i}, \mathtt{f}(\pi), \mathtt{t_{i+1}})$ is a step of $\mathtt{A}$ when $\mathtt{f}(\pi) \in \mathtt{sig}(\mathtt{A})$, or $\mathtt{t_i} = \mathtt{t_{i+1}}$ when $\mathtt{f}(\pi) = \perp$.

Therefore, by definition of execution, the sequence obtained by replacing every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α with either $(\mathtt{t_i}, \mathtt{f}(\pi), \mathtt{t_{i+1}})$ when $\mathtt{f}(\pi) \in \mathtt{sig(A)}$, or $\mathtt{t_i}$ when $\mathtt{f}(\pi) = \perp$ is an execution of A. Since this sequence is $\mathtt{f}(\alpha)$, we conclude that $\mathtt{f}(\alpha) \in \mathtt{execs(A)}$.

$\Leftarrow$: Let α be a sequence of alternating states and actions of $\mathtt{A}'$ such that $\mathtt{f}(\alpha) \in \mathtt{execs(A)}$. This means that α begins with some initial state $\mathtt{t_0}$ of A, and that, for every triple $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ of elements in α, either $(\mathtt{t_i}, \mathtt{f}(\pi), \mathtt{t_{i+1}})$ is a step of A when $\mathtt{f}(\pi) \in \mathtt{sig(A)}$, or $\mathtt{t_i} = \mathtt{t_{i+1}}$ when $\mathtt{f}(\pi) = \perp$. From this assumption and Definition 3.4.1, it follows that $\mathtt{t_0}$ is an initial state of $\mathtt{A}'$ and that every triple $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ of elements in α is a transition of $\mathtt{A}'$. Thus, $\alpha \in \mathtt{execs(A')}$.

2. Follows from part 1 and the fact that f preserves the classification of actions as "input", "output", and "internal".

■

Theorem 3.4.2 *If $\mathtt{A}'$ is a specialized extension of A with a signature-mapping f, then*

1. $\alpha \in \mathtt{execs(A')} \Rightarrow \mathtt{f}(\alpha|_{\mathtt{p}}) \in \mathtt{execs(A)}$.
2. $\beta \in \mathtt{traces(A')} \Rightarrow \mathtt{f}(\beta) \in \mathtt{traces(A)}$.

Proof 3.4.2: Follows immediately from Theorem 3.2.1 and Lemma 3.4.1 ■

Since signature extension does not modify the original automata beyond simple renaming of actions, we would expect it to have minimal effect on the proof-reuse theorems (Theorem 3.3.1 and Lemma 3.3.2) of Section 3.3 when those theorems are used in verifying specialized extensions of automata. We prove this intuition correct in Theorem 3.4.4 below; this theorem is an adaptation of Theorem 3.3.1 for the case when child automata are specialized extensions of their parents. The theorem follows from Theorem 3.3.1 and the following lemma, which establishes that a simulation relation between two automata is preserved when these automata are signature-extended:

Lemma 3.4.3 *Let $\mathtt{A}'$ be the signature extension of A with a signature-mapping f. Let $\mathtt{S}'$ be the signature extension of S with a signature-mapping g. Assume that A has the same external signature as S and that there is a simulation relation R from A to S. Assume further that $\mathtt{A}'$ has the same external signature as $\mathtt{S}'$, and that, for every external action $\pi \in \mathtt{sig(A')}$, $\mathtt{g}(\pi) = \mathtt{f}(\pi)$. Then, R is a simulation relation from $\mathtt{A}'$ to $\mathtt{S}'$.*

Proof 3.4.3: Follows straightforwardly from Definitions 2.2.1 and 3.4.1. ■

The only difference between the statements of Theorem 3.4.4 below and Theorem 3.3.1 is that here, whenever child's actions are used in the context of the parent automaton (as in Condition 2a), they are translated via the signature-mapping to the corresponding actions of the parent.

Theorem 3.4.4 *Let automaton* $\mathtt{A'}$ *be a specialized extension of* $\mathtt{A}$ *with a signature-mapping* $\mathtt{f}$*. Let automaton* $\mathtt{S'}$ *be a specialized extension of* $\mathtt{S}$ *with a signature-mapping* $\mathtt{g}$*, such that* $\mathtt{S'} = \mathtt{specialize(extend(S)(G,g))(N,N_0,TR)}$*. Assume that* $\mathtt{A}$ *and* $\mathtt{S}$ *have the same external signatures and that* $\mathtt{A}$ *implements* $\mathtt{S}$ *via a simulation relation* $\mathtt{R_p}$*. Assume further that* $\mathtt{A'}$ *and* $\mathtt{S'}$ *have the same external signatures, and that, for every external action* $\pi \in \mathtt{A'}$*,* $\mathtt{g}(\pi) = \mathtt{f}(\pi)$*.*

A relation $\mathtt{R_c} \subseteq \mathtt{states(A')} \times \mathtt{states(S')}$*, defined in terms of relation* $\mathtt{R_p}$ *and a new relation* $\mathtt{R_n} \subseteq \mathtt{states(A')} \times \mathtt{N}$ *as* $\{(\mathtt{t},\mathtt{s}) : (\mathtt{t}|_\mathtt{p}, \mathtt{s}|_\mathtt{p}) \in \mathtt{R_p} \wedge (\mathtt{t}, \mathtt{s}|_\mathtt{n}) \in \mathtt{R_n}\}$*, is a simulation from* $\mathtt{A'}$ *to* $\mathtt{S'}$ *if* $\mathtt{R_c}$ *satisfies the following two conditions:*

1. *For every* $\mathtt{t} \in \mathtt{start(A')}$*, there exists a state* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$ *such that* $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$*.*
2. *If* $\mathtt{t}$ *is a reachable state of* $\mathtt{A'}$*,* $\mathtt{s}$ *is a reachable state of* $\mathtt{S'}$ *such that* $\mathtt{s}|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t}|_\mathtt{p})$ *and* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$*, and* $(\mathtt{t}, \pi, \mathtt{t'})$ *is a step of* $\mathtt{A'}$*, then there exists a finite sequence* α *of alternating states and actions of* $\mathtt{S'}$*, beginning from* $\mathtt{s}$ *and ending at some state* $\mathtt{s'}$*, and satisfying the following conditions:*
   - *(a)* $\mathtt{g}(\alpha|_\mathtt{p})$ *is an execution fragment of* $\mathtt{S}$*.*
   - *(b) For every step* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}})$ *in* α*,* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}|_\mathtt{n}) \in \mathtt{TR}$*.*
   - *(c)* $\mathtt{s'}|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t'}|_\mathtt{p})$*.*
   - *(d)* $\mathtt{s'}|_\mathtt{n} \in \mathtt{R_n(t')}$*.*
   - *(e)* α *has the same trace as* $(\mathtt{t}, \pi, \mathtt{t'})$*.*

Proof 3.4.4: Follows straightforwardly as a corollary from Theorem 3.3.1 and Lemma 3.4.3. ■

Theorem 3.4.4 can be used in practice in the same way as Theorem 3.3.1 and Lemma 3.3.2 (see the discussion after the proof of Theorem 3.3.1 on page 41): Transitions involving new actions introduced by signature extension are defined entirely by the specialization code and, therefore, involve reasoning about this code alone. Transitions involving parent's actions, which are possibly renamed by the child, depend on the code of both the parent and the child. Even when actions are renamed, the task of proving that the simulation relation holds for such transitions typically allows one to rely on the simulation proof of the parent automata to deduce conditions 2a, 2c, and 2e, and requires verification of conditions 1, 2b, and 2d only.

3.5 Related work

The works that most closely relate to ours are those of Soundarajan and Fridella [87, 86] and Stata and Guttag [88]. Unlike our formalisms, both of these works are restricted to the context of sequential programming and do not encompass reactive components.

Like us, Soundarajan and Fridella [87, 86] have recognized that incremental reasoning is important in exploiting the full potential of inheritance. They present a specification notation and a verification procedure geared towards such incremental reasoning. However, they consider a more general type of inheritance — one that allows a child to override behavior of the parent. As a result, the proof-reuse result they obtain is much weaker and less structured than ours. In particular, reasoning reuse applies only when the simulation function (abstraction function, in their case) between child automata is identical to that between parent automata, and only to those actions that are inherited from the parent without any modification. In contrast, our framework applies to all types of actions, including those which are modified by the child.

Stata and Guttag [88] have also recognized the need for proof-reuse in a manner similar to that suggested here. They suggest a framework for defining programming guidelines and supplement this framework with informal rules that may be used to facilitate reasoning about correctness of a subclass given the correctness of the superclass is known. However, they only addressed informal reasoning and did not provide the mathematical foundation for formal proofs.

Numerous other research projects, for example [2, 16, 28, 33, 49, 56, 70, 69, 79, 13, 76, 48], have dealt formally with inheritance and its semantics. In particular, many projects, such as [70, 69, 33, 13, 76], focus on defining inheritance constructs in ways that either automatically imply or simplify the task of proving that a child behaves indistinguishably from its parent, in other words, that the child satisfies its parent's specification. However, no other work that we are aware of allows for reuse of a parent-level simulation proof when showing that a child satisfies (simulates) *its own specification*.

We note also that, while our definition of subclassing for extension is similar to behavioral subtyping of Liskov and Wing [70, 69], it is not identical: Behavioral subtyping requires only that a child behave indistinguishably from its parent when the child is used in the context of the parent, that is, when the execution of the child contains only the parent's actions, and none of the actions introduced by the child. Subclassing for extension enforces a stronger property: any trace (execution) of the child, even one that has actions introduced by the child, is indistinguishable from a parent's trace when all such new actions (and new state variables) are projected out.

3.6 Discussion

In this part of the thesis, we have presented an inheritance-based formalism for modeling and verifying systems incrementally.

The formalism defines two inheritance constructs that can be used to model a modified version of an abstract model of a system by specifying how the modification is different from the original. Using these constructs, one can model a complex system incrementally, by starting from a basic model and then, at each step, adding support for a new property of the system.

For simplicity, we described the formalism in terms of two levels of abstraction: "specification" and "algorithm"; but in general, the formalism is complementary to the technique of successive refinement. It can be used for modeling systems at any relevant level of abstraction, from the lowest level corresponding to software code, to the highest one corresponding to the most abstract system specification.

A distinguishing feature of our formalism is its support for incremental verification, which compliments incremental modeling. The formalism provides fundamental theorems (3.3.1 and 3.4.4) that state formally how a simulation proof of one abstract model of a system satisfying another can be reused and extended to a simulation proof for the modified versions of these models. This allows one, not only to model and specify a complex system incrementally, but also to verify incrementally that the model satisfies its specification.

The formalism, and in particular its incremental verification component, was motivated by and refined during the design and modeling of the complex middleware system that is presented in Part II of this dissertation. The ability to model and verify the system incrementally was critical in making the project tractable and in making it clear which part of the algorithm implemented which property. As we explain in Chapter 11, standard compositional techniques would have not been sufficient.

The formalism described here has been presented using the I/O automaton model — the same model that we used to model the complex middleware system. The I/O automaton model has been used extensively for modeling and reasoning about complex distributed systems and has been developed into a programming and modeling language, called *IOA* [44, 45]. As one of our future projects, we plan to facilitate the incorporation of our inheritance-based approach into the IOA toolset, thereby enriching its modeling and reasoning facilities.

The I/O automaton model has been a convenient model in which to express our formalism. The essence of the approach, however, is general enough to be applicable to other state-transition formal models, such as TLA [65], UNITY [76], or various forms of process algebra [53, 75], or, in other words, to any formal model that supports simulation proofs. One interesting direction for future research is to enrich the standard formal modeling languages with a version of our formalism.

The formalism presented in this chapter allows modeling of systems using two standard and important types of inheritance: specialization and subclassing for extension. In our future work, we are planning to expand the formalism, including its incremental verification aspect, to support other types of inheritance.

Of particular importance is a construct that would allow modifications that override some of the system's behavior. In general, one would expect little, if any, proof reuse possible for such a construct, since modifications done to a system may invalidate whatever reasoning has been done about it. Nevertheless, useful approaches to circumventing this impasse could rely on limiting the types of the modifications allowed by the construct and on requiring the modifications to preserve certain invariants.

The formalism presented here is an important step toward scalable and cost-effective formal methods and toward practical software design methodologies that, in addition to facilitating reuse of code, also facilitate reuse of reasoning. In general, any extensions to the formalism that we make in the future will be motivated and guided by our work on designing and modeling complex distributed systems. This approach will ensure that, like the formalism presented in this thesis, these extensions will have important, practical implications.

Part II

Group Communication Service

This part of the dissertation contains our design of the Virtually Synchronous Group Communication service targeted for wide-area networks. Chapters 4 – 8 are based on the material published in [59, 60].

In Chapter 4 we present the client-server architecture of our GCS and formally specify the assumptions we make on the membership service and the underlying communication substrate. Chapter 5 contains precise specifications of the safety and liveness properties satisfied by our GCS.

The algorithm implementing Virtual Synchrony is then given in Chapter 6; it is accompanied by informal correctness arguments. A formal correctness proof that the algorithm of Chapter 6, when it operates in the environment specified in Chapter 4, satisfies the specifications of Chapter 5 is given in the subsequent two chapters: safety – in Chapter 7, and liveness – in Chapter 8.

In Chapter 9 we consider performance characteristics of our algorithm; we formalize and prove our claim that our Virtual Synchrony algorithm involves just a single message exchange round among members of the new view. This chapter builds upon the liveness proof of Chapter 8: The liveness proof establishes that, after the environment starts to behave well, our GCS system *eventually* does what it is supposed to do; the performance analysis quantifies what "eventually" means in terms of various parameters such as message latency and timing of network and membership events.

In Chapter 10 we illustrate the utility of our GCS system by describing a simple application that can be effectively built using GCS. The application implements a variant of a data service that allows a dynamic group of clients to access and modify a replicated data object. The application is prototypical of some collaborative computing applications, such as for example a shared white-board application.

Chapter 11 concludes the dissertation.

Chapter 4

Client-Server Architecture and Environment Specification

Our service is designed to operate in an asynchronous message-passing environment. Processes and communication links may fail and may later recover, possibly causing network partitions and merges. For simplicity, we assume that processes recover with their running state intact; this is a plausible assumption as processes can keep their running state on stable storage. We do not explicitly model process crashes and recoveries because under this assumption a crashed process is indistinguishable from a slow one. In Section 6.4, we argue that our algorithm also provides meaningful semantics when group communication processes lose their entire state upon a crash and recover with their state reset to an initial value.

Our Group Communication service is implemented by a collection of GCS *end-points*, which are the GCS processes that run at the application clients' locations. GCS end-points handle clients' multicast requests and inform their clients of view changes.

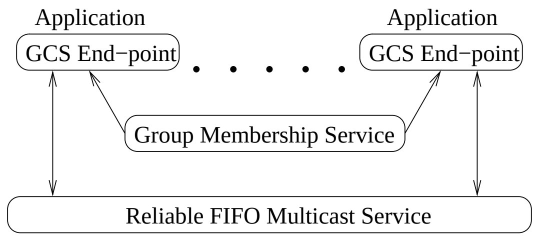


Figure 4.1: Client-server architecture: GCS end-points usie an external membership service. Arrows represent interaction between GCS end-points and underlying services.

The GCS architecture is depicted in Figure 4.1. All GCS end-points run the same algorithm. The algorithm relies on the underlying membership and multicast services to handle respectively formation of views and transmission of messages. The algorithm's task is to synchronize output of the two underlying services to implement the Virtual Synchrony semantics.

Sections 4.1 and 4.2 below give precise specifications of the interface and semantics that the underlying membership and multicast services have to provide in order to be suitable for our algorithm. Services that satisfy these (or very similar) requirements have been previously used for GCSs, and efficient implementations of these services for WANs exist.

4.1 The membership service specification

This section presents a formal specification of the membership services that are appropriate for our GCS design. For simplicity, here and in the rest of the thesis, we assume that there is a single process group; multiple groups can be supported by treating each independently. We also omit the part of the interface that handles processes' requests to join and leave groups.

Figure 4.2 contains an I/O automaton, called MBRSHP, that defines the interface and the safety properties of the membership service. The service interface is given by the automaton's signature;[1] Informally, it consists of the following two output actions:

$\mathtt{start\_change}_\mathtt{p}(\mathtt{cid}, \mathtt{set})$ notifies process p that the membership service is attempting to form a view with the members of set; cid is a local start-change identifier.

$\mathtt{view}_\mathtt{p}(\mathtt{v})$ notifies process p that the membership service has succeeded in forming view v. A view v is a triple consisting of an identifier v.id, a set of members v.set, and a function v.startId that maps members of v to start-change identifiers. Two views are the same if they consist of identical triples.

Automaton MBRSHP maintains two state variables, mbrshp_view[p] and start_change[p], for each client p. These variables contain respectively the last view and the last start_change message issued to client p; the variables are updated in the effects of the transitions. The safety properties satisfied by the MBRSHP automaton include two basic properties, which are provided by virtually all group membership services (for example, [21, 35, 8, 43, 12, 58, 82, 6]), as well as some new properties concerning the start_change notifications.

The two basic properties are *Self Inclusion* and *Local Monotonicity.* Self Inclusion requires every view issued to a client p to include p as a member; this property is enforced with a precondition $\mathtt{p} \in \mathtt{v.set}$ on the $\mathtt{view}_\mathtt{p}(\mathtt{v})$ action. Local Monotonicity requires that view identifiers delivered to p be monotonically increasing; this property is enforced with a precondition $\mathtt{v.id} > \mathtt{mbrshp\_view}[\mathtt{p}]$ on the $\mathtt{view}_\mathtt{p}(\mathtt{v})$ action. Local Monotonicity has two important consequences: the same view is not delivered more than once to the same client, and clients that receive the same two views receive them in the same order [27].

[1]When specifying a distributed system as a centralized automaton, we subscript each external action of the specification automaton with the location (or process) in the distributed system at which the action occurs.

```
AUTOMATON MBRSHP

Type:
    Proc: Set of end-points.
    StartChangeId: Total-order; cid_0 is smallest.
    ViewId: Partial-order; vid_0 is smallest.
    View: ViewId × SetOf(Proc) × (Proc → StartChangeId).
Def:  v_p = ⟨vid_0, {p}, {(p →cid_0)}⟩.

Signature:
 Output: start_change_p(cid, set), Proc p, StartChangeId cid, SetOf(Proc) set
         view_p(v), Proc p, View v

State:
 For all Proc p: View  mbrshp_view[p], initially v_p
 For all Proc p: (StartChangeId × SetOf(Proc)) start_change[p], initially ⟨cid_0, {}⟩

Transitions:
 OUTPUT  start_change_p(cid, set)
 pre: cid > mbrshp_view[p].startId(p)
      cid ≥ start_change[p].id
      p ∈ set
 eff: start_change[p] ← ⟨cid, set⟩

 OUTPUT  view_p(v)
 pre: p ∈ v.set ∧ v.id > mbrshp_view[p].id
      v.set ⊆ start_change[p].set
      v.startId(p) = start_change[p].id
      v.startId(p) > mbrshp_view[p].startId(p)
 eff: mbrshp_view[p] ← v
```

Figure 4.2: Membership service interface and safety specification.

In addition, the MBRSHP automaton specifies that the membership service must issue at least one `start_change` notification to client `p` before issuing a new view `v` to `p`. Also, the start-change identifier `v.startId(p)` contained in the new view `v` must be the same as the identifier of the latest preceding `start_change` issued to `p`. These two requirements are enforced by the last two preconditions on $\texttt{view}_\texttt{p}(\texttt{v})$. In particular, the former one is achieved by requiring that a bigger start-change identifier than the one associated with `p` in the last view has been issued to `p`.

The MBRSHP specification allows the membership service to react to connectivity changes happening during view formation. Whenever the service wants to add new members to the membership, it has to issue a new `start_change` notification to the clients: the second precondition on $\texttt{view}_\texttt{p}(\texttt{v})$ actions requires the membership `v.set` to be a subset of the tentative membership `set` included in the last `start_change` notification. In order to remove members from a forming view, the service does not need to issue a new `start_change` notification.

The first `start_change` notification issued to `p` after a view marks the beginning of a new view formation period. It includes a new local identifier `cid`, different from the ones that were previously sent to `p`: the first precondition on $\texttt{start\_change}_\texttt{p}(\texttt{cid}, \texttt{set})$ requires `cid` to be strictly greater than `mbrshp_view[p].startId(p)`. Subsequent `start_change` notifications sent during an on-going view formation may either reuse the last start-change identifier or issue a new one, as specified by the second precondition on `start_change` actions. We ensure uniqueness of local start-change identifiers by generating them in increasing order.

Notice that the MBRSHP automaton does not specify any relationship between views issued to different clients.

Example 4.1.1 *Figure 4.3 presents a sample execution that shows the* MBRSHP *service delivering different sequences of views to two different clients,* $\mathtt{a}$ *and* $\mathtt{b}$*. Arrows represent time passage at each client; gray dots represent events. First, both clients receive the same view* $\mathtt{v} = \langle 2, \{\mathtt{a}, \mathtt{b}\}, [\mathtt{a}:1, \mathtt{b}:1]\rangle$*; we illustrate this with a circle around the view events at both clients. Then, client* $\mathtt{b}$ *receives a view* $\mathtt{v}_{\mathtt{mid}} = \langle 3, \{\mathtt{b}\}, [\mathtt{b}:2]\rangle$ *by itself. Then, both clients receive another common view* $\mathtt{v}' = \langle 4, \{\mathtt{a}, \mathtt{b}\}, [\mathtt{a}:2, \mathtt{b}:3]\rangle$*. Notice how the start-change identifiers included in the views correspond to the last start-change identifiers issued to the clients.*

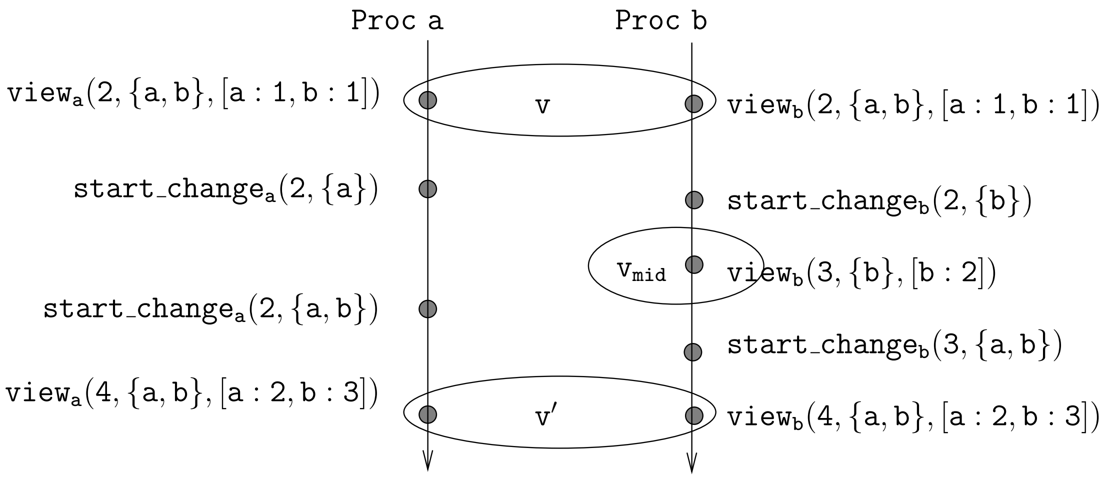

Proc a
Proc b
view_a(2, {a, b}, [a : 1, b : 1])
v
view_b(2, {a, b}, [a : 1, b : 1])
start_change_a(2, {a})
start_change_b(2, {b})
v_mid
view_b(3, {b}, [b : 2])
start_change_a(2, {a, b})
start_change_b(3, {a, b})
view_a(4, {a, b}, [a : 2, b : 3])
v′
view_b(4, {a, b}, [a : 2, b : 3])

Figure 4.3: A sample execution of MBRSHP.

We do *not* specify liveness properties for membership services. Instead, when we specify the liveness properties of our GCS in Section 5.2, we *condition* them on the behavior of the membership service. For example, we state that if the same view is delivered to all the members and the members do not receive any subsequent membership events, then they eventually deliver this view to their application clients. Existing membership services do satisfy meaningful liveness properties. For example, [58] guarantees that, when the network stabilizes, all members receive the "correct" view and no other views thereafter. By combining our GCS liveness properties with such membership liveness properties, we can restate the liveness properties of our GCS conditionally on the network behavior.

The MBRSHP specification allows for simple and efficient distributed implementations that also satisfy meaningful liveness properties. The membership service of [58] is an example of such an implementation; our design was implemented by Tarashchanskiy [92] using this membership service. In this service, a small number of servers support a large number of clients, communicating with them asynchronously via FIFO ordered channels (TCP sockets). In case a server fails, clients can migrate to another server. Other existing membership algorithms (for example, [35, 8]) could also be extended easily to provide the interface and semantics specified here.

4.2 The reliable FIFO multicast service specification

The group communication end-points communicate with each other using an underlying multicast service that provides reliable FIFO communication between every pair of connected processes. Many existing group communication systems (for example, [47, 12, 35, 6]) implement Virtual Synchrony over similar communication substrates. In our implementation [92], we use the service of [10].

Figure 4.4 presents an I/O automaton, CO_RFIFO, that specifies a multicast service appropriate for our GCS design. Portions of the code that define liveness properties are colored gray.

```
AUTOMATON CO_RFIFO

Signature:
Input:                                                Output:   deliver_{p,q}(m), Proc p, Proc q, Msg m
  send_p(set,m), Proc p, SetOf(Proc) set, Msg m      Internal: lose(p,q), Proc p, Proc q
  reliable_p(set), Proc p, SetOf(Proc) set                     skip_task(p,q), Proc p, Proc q
  live_p(set), Proc p, SetOf(Proc) set

State:
 For all Proc p, Proc q: SequenceOf(Msg) channel[p][q], initially empty
 For all Proc p: SetOf(Proc) reliable_set[p], initially {p}
 For all Proc p: SetOf(Proc) live_set[p], initially {p}

Transitions:
 INPUT  send_p(set, m)                              INTERNAL  lose(p, q)
 eff: (∀ q ∈ set) append m to channel[p][q]         pre: q ∉ reliable_set[p]
                                                    eff: dequeue last message from channel[p][q]
 OUTPUT  deliver_{p,q}(m)
 pre: m = first(channel[p][q])                      INPUT live_p(set)
 eff: dequeue m from channel[p][q]                  eff: live_set[p] ← set

 INPUT  reliable_p(set)                             INTERNAL skip_task(p, q)
 eff: reliable_set[p] ←  set                        pre: q ∉ live_set[p]

Tasks:
 For each Proc p, Proq q:  C_{p,q} = ({deliver_{p,q}(m) | m ∈ Msg} ∪ {skip_task(p,q)} ∪ {lose(p,q)})
```

Figure 4.4: Reliable FIFO multicast service specification. Liveness-related code is colored gray.

Automaton CO_RFIFO maintains a FIFO queue channel[p][q] for every pair of end-points. An input action $\texttt{send}_\texttt{p}(\texttt{set},\texttt{m})$ models a multicast of message m from end-point p to the end-points listed in the set by appending m to the channel[p][q] queues for every end-point q in set. The $\texttt{deliver}_{\texttt{p,q}}(\texttt{m})$ action removes the first message from channel[p][q] and delivers it to q.

In addition, the CO_RFIFO specification allows an end-point p to use the $\texttt{reliable}_\texttt{p}(\texttt{set})$ action to require that the multicast service maintain a reliable (gap-free) FIFO connection to the end-points listed in set. Whenever this action occurs, set is stored in a special variable reliable_set[p]. For every process q not in reliable_set[p], the multicast service may lose an arbitrary suffix of the messages sent from p to q, as modeled by an internal action lose(p, q).

In order for the multicast service to be considered live, messages sent to live and connected processes must eventually reach their destinations. The CO_RFIFO specification enforces this property in the gray-colored portion of its code.

Recall from Chapter 2 that an infinite fair execution of an automaton must contain either infinitely many events from each task $\mathtt{C}$ or infinitely many occurrences of states in which no action in $\mathtt{C}$ is enabled. Automaton CO_RFIFO defines the set $\mathtt{C}_{p,q} = (\{\mathtt{deliver}_{p,q} \mid \mathtt{m} \in \mathtt{Msg}\} \cup \mathtt{skip\_task}(\mathtt{p},\mathtt{q}) \cup \mathtt{lose}(\mathtt{p},\mathtt{q}))$ to be a task for each pair of end-points p and q. This definition implies that $\mathtt{deliver}_{p,q}$ actions must occur in an infinite fair execution of CO_RFIFO, provided the following three conditions hold: there are messages sent from p to q – hence, $\mathtt{deliver}_{p,q}$ is enabled; the client at p is interested in maintaining reliable connection to q – hence, $\mathtt{lose}(\mathtt{p},\mathtt{q})$ is disabled; and q is believed to be connected to p – hence, a special action $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ is disabled, as explained below.

Action $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ is defined only to provide an alternative to $\mathtt{deliver}_{p,q}$ actions so that $\mathtt{deliver}_{p,q}$ actions are not required to happen when q is believed to be disconnected from p. $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ is an internal action that has no effect on the state of CO_RFIFO and is enabled when q is believed to be disconnected from p. Such belief is modeled using special $\mathtt{live}_p(\mathtt{set})$ input actions. The set argument is assumed to represent a set of processes that are alive and connected to p; when such an input happens, set is stored in a state variable $\mathtt{live\_set}[\mathtt{p}]$. The precondition on the $\mathtt{skip\_task}(\mathtt{p},\mathtt{q})$ action is $\mathtt{q} \notin \mathtt{live\_set}[\mathtt{p}]$.

An important implication of how tasks are defined in CO_RFIFO is that, if q remains in both $\mathtt{live\_set}[\mathtt{p}]$ and $\mathtt{reliable\_set}[\mathtt{p}]$ from some point on in a fair execution of CO_RFIFO, then all the messages that p sends to q from that point on are eventually delivered to q.

Chapter 5

Specifications of the Group Communication Service

The two sections of this chapter contain specifications of the safety and liveness properties satisfied by our group communication service (GCS). The specifications capture a core set of properties that is commonly provided by group communication systems and that have been shown useful for facilitating implementations of many distributed applications and other, stronger, GCS properties (see [27]). In Chapter 10, we will illustrate the utility of GCS by describing a simple application that can be effectively built using GCS. The application implements a variant of a data service that allows a dynamic group of clients to access and modify a replicated data object.

5.1 Safety properties

We present the safety specification of our group communication service incrementally, as four automata: In Section 5.1.1 we specify a simple group communication service that synchronizes delivery of views and application messages to require *Within-View Delivery* of messages. In Section 5.1.2 we extend the specification of Section 5.1.1 to also require *Virtually-Synchronous Delivery*, the key property of Virtual Synchrony (see Section 1.1). In Section 5.1.3 we specify the *Transitional Set* property, which complements Virtually-Synchronous Delivery. Finally, in Section 5.1.4, we specify the *Self Delivery* property, which requires the GCS to deliver to each client the client's own messages.

The incremental development of the safety specification is matched later when we develop the algorithm and its correctness proof in Chapter 6 and Chapter 7.

5.1.1 Within-View reliable FIFO multicast

In this section we specify a GCS that captures the following properties:

- Views delivered to the application satisfy the Self Inclusion and Local Monotonicity properties of the MBRSHP service, see Section 4.1.
- Messages are delivered in the same view in which they were sent. This property is useful for many applications (see [43, 27, 91]) and appears in several systems and specifications (for example, [21, 93, 8, 77, 39, 52, 31]). A weaker property that requires each message to be delivered in the same view at every process that delivers it, but not necessarily the view in which it was sent, is typically implemented on top of an implementation of Within-View Delivery (see [27]).
- Messages are delivered in gap-free FIFO order (within views). This is a basic property upon which one can build services with stronger ordering guarantees, such as causal order or total order. The totally ordered multicast algorithm of [25] is implemented atop a service with a similar specification.

```
AUTOMATON WV_RFIFO : SPEC

Signature:
 Input:  send_p(m), Proc p, AppMsg m
 Output: deliver_p(q, m), Proc p, Proc q, AppMsg m
         view_p(v), Proc p, View v

State:
 For all Proc p, View v: SequenceOf(AppMsg) msgs[p][v], initially empty
 For all Proc p, Proc q: Int last_dlvrd[p][q], initially 0
 For all Proc p: View current_view[p], initially v_p

Transitions:
 INPUT  send_p(m)
 eff: append m to msgs[p][current_view[p]]

 OUTPUT  deliver_p(q, m)
 pre: m = msgs[q][current_view[p]] [last_dlvrd[q][p]+1]
 eff: last_dlvrd[q][p] ← last_dlvrd[q][p]+1

 OUTPUT  view_p(v)
 pre: p ∈ v.set ∧ v.id > current_view[p].id
 eff: (∀ q) last_dlvrd[q][p] ← 0
      current_view[p] ← v
```

Figure 5.1: WV_RFIFO service specification.

Figure 5.1 presents automaton WV_RFIFO : SPEC that models this specification. The automaton uses centralized queues msgs[p][v] of application messages for each sender p and view v. It also maintains a variable current_view[p] that contains the last view delivered to each process p, and a variable last_dlvrd[q][p], for every pair of processes q and p, containing the index in the msgs[q][current_view[p]] queue of the last message from q delivered to p in p's current view.

Action $\mathtt{view}_\mathtt{p}(\mathtt{v})$ models the delivery of view v to process p; the precondition on this action enforces Self Inclusion and Local Monotonicity. Action $\mathtt{send}_\mathtt{p}(\mathtt{m})$ models the multicast of message m from process p to the members of p's current view by appending m to msgs[p][current_view[p]]. Action $\mathtt{deliver}_\mathtt{p}(\mathtt{q},\mathtt{m})$ models the delivery to process p of message m sent by process q. The gap-free FIFO ordered delivery of messages within-views is enforced by its precondition, which allows delivery of only the message indexed by last_dlvrd[q][p] + 1 in the msgs[q][current_view[p]] queue.

5.1.2 Virtually-Synchronous delivery

In this section we use the inheritance-based methodology to modify the WV_RFIFO : SPEC automaton to also enforce the *Virtually-Synchronous Delivery* property. The modified automaton, VSRFIFO : SPEC is defined by the code contained in both Figures 5.1 and 5.2.

```
AUTOMATON VS_RFIFO : SPEC    MODIFIES  WV_RFIFO : SPEC

Signature Extension:
 Output:   view_p(v) modifies wv_rfifo.view_p(v)
 Internal: set_cut(v, v′, c), View v, View v′, (Proc → Int)⊥ c new

State Extension:
 For all View v, v′: (Proc→Int)⊥ cut[v][v′], initially ⊥

Transition Restriction:
 OUTPUT  view_p(v)                                          INTERNAL  set_cut(v, v′, c)
 pre: cut[current_view[p]] [v] ≠ ⊥                          pre: cut[v][v′] = ⊥
      (∀ q) last_dlvrd[q][p]=cut[current_view[p]] [v](q)    eff: cut[v][v′] ← c
```

Figure 5.2: VS_RFIFO service specification.

Figure 5.2 contains the code that enforces the *Virtually-Synchronous Delivery* property. Recall from Section 1.1 that this property requires processes to move together from view v to view v' to deliver same set of messages while in view v. Since the parent specification, WV_RFIFO : SPEC, imposes gap-free FIFO delivery of messages, a message set can be represented by a set of indices, each pointing to the last message from each member of v; such representation of a set is called a *cut*.

The WV_RFIFO : SPEC automaton fixes a cut for processes that wish to move from some view v to some view v': A new internal action $\texttt{set\_cut}(v, v', c)$ sets a new variable $\texttt{cut}[v][v']$ to a cut mapping c. For a given pair of views, v and v', the cut is chosen only once, *nondeterministically*. Delivery of a view v to process p is allowed only if a cut for moving from p's current view into v has been set and if p has delivered all the messages identified in this cut. These conditions are enforced by the two new preconditions of the $\texttt{view}_p(v)$ action (see Figure 5.2). Since VSRFIFO : SPEC is a modification of WV_RFIFO : SPEC the new preconditions work in conjunction with the preconditions in $\texttt{view}_p(v)$ of WV_RFIFO : SPEC.

The VSRFIFO : SPEC automaton, being a safety specification, does not require liveness properties to hold, for instance, that processes actually deliver messages specified by the cuts, and hence, are able to satisfy conditions for delivering new views. Such liveness specifications are stated in Section 5.2.

5.1.3 Transitional Set

While Virtually-Synchronous Delivery is a useful property, a process that moves from view v to view v' cannot tell locally which of the processes in $v.\texttt{set} \cap v'.\texttt{set}$ move to view v' directly from view v, and which move to v' from some other view. In order for the application to be able to exploit the Virtually-Synchronous Delivery property, application processes need to be informed which other processes move together with them from their current view into their new view. The set of processes that transition together from one view into the next is called a *transitional set* [27]:

Definition 5.1.1 *A transitional set from view* v *to view* v′*, is a subset of* v.set ∩ v′.set *that includes: (a) all processes that receive view* v′ *while in view* v*; and (b) no process that receive view* v′ *while in a view other than* v*.*

The notion of a transitional set was first introduced as part of a special transitional view in the EVS [77] model. In our formulation (as in [27]), transitional sets are delivered to the application along with views, as an additional parameter T.

Example 5.1.1 *Assume that Alice and Bob are using a Virtually Synchronous GCS that eventually reports the views produced by the* MBRSHP *service to Alice and Bob. Consider the scenario described in Example 4.1.1: both Alice and Bob receive views* v *and* v′ *with the membership {Alice, Bob}. Just from these views, Alice does not know whether Bob receives view* v′ *while in view* v*, or while in some other view,* $\mathtt{v_{mid}}$ *with the membership {Bob}. If the former holds, then Alice does not need to synchronize with Bob because Virtually-Synchronous Delivery guarantees that they have received the same messages while in view* v*; otherwise, she does. The transitional set given to Alice together with view* v′ *provides this information.*

```
AUTOMATON TRANS_SET : SPEC

Signature:
Output: view_p(v,T), Proc p, View v, SetOf(Proc) T
Internal: set_prev_view_p(v), Proc p, View v

State:
For all Proc p:  View current_view[p], initially v_p
For all Proc p, View v: View_⊥ prev_view[p][v], initially ⊥

Transitions:
 OUTPUT  view_p(v, T)                              INTERNAL  set_prev_view_p(v)
 pre: prev_view[p][v] = current_view[p]            pre: p ∈ v.set
      (∀ q ∈ v.set ∩ current_view[p].set)               prev_view[p][v] = ⊥
            prev_view[q][v] ≠ ⊥                   eff: prev_view[p][v] ← current_view[p]
      T = {q ∈ v.set ∩ current_view[p].set |
            prev_view[q][v] = current_view[p]}
 eff: current_view[p] ← v
```

Figure 5.3: Transitional set specification.

Figure 5.3 presents an automaton TS : SPEC that specifies delivery of transitional sets (Definition 5.1.1). The automaton has two types of actions: output actions $\mathtt{view_p}(\mathtt{v}, \mathtt{T})$, which deliver view v and transitional set T to process p; and internal actions $\mathtt{set\_prev\_view_p}(\mathtt{v})$, which declare that q intends to deliver view v while in its current view. The intentions are recorded in the variable prev_view[p][v], and the current views are recorded in the variable current_view[p].

Before process p can deliver a view v, each member q in the intersection of these views must execute $\mathtt{set\_prev\_view_q}(\mathtt{v})$, as enforced by the second precondition. The transitional set T delivered by p with v is then computed to consist of those processes q in the intersection current_view[p].set ∩ v.set for which prev_view[q][v] is the same as current_view[p]; this is specified by the third precondition on $\mathtt{view}_{\mathtt{p}}(\mathtt{v}, \mathtt{T})$.

5.1.4 Self Delivery

We now specify the *Self Delivery* property, which requires that each client receives all the messages it sent in a given view before receiving a new view. We specify this property as a simple modification of the WV_RFIFO : SPEC automaton presented in Section 5.1.1; the modified automaton is defined by the code contained in both Figures 5.1 and 5.4.

```
AUTOMATON WV_RFIFO+SELF : SPEC     MODIFIES  WV_RFIFO : SPEC

Signature Extension:
 Output: view_p(v) modifies wv_rfifo.view_p(v)

Transition Restriction:
 OUTPUT  view_p(v)
 pre: last_dlvrd[p][p] = LastIndexOf(msgs[p][current_view[p]])
```

Figure 5.4: WV_RFIFO+SELF service specification.

In order to enforce Self Delivery, a new precondition on the $\mathtt{view}_{\mathtt{p}}(\mathtt{v})$ action requires the last_dlvrd[p][p] index to point to the last message sent by client p in its current view. Since the parent automaton, WV_RFIFO : SPEC, guarantees within-view gap-free FIFO delivery, this precondition implies that all of p's messages have in fact been delivered back to p.

In order for a GCS to be live and satisfy Within-View Delivery, Self Delivery, and Virtually-Synchronous Delivery, the GCS must *block* its application from sending new messages during view formation periods; this is proved in [43]. Therefore, we introduce a block/block_ok synchronization when we extend our algorithm to support the Self Delivery property in Section 6.3.

Our formulation of Self Delivery as a safety property, when combined with the liveness property of Section 5.2, implies the formulations in [27] and [77] of Self Delivery as a liveness property. These formulations require a GCS to *eventually* deliver to each process its own messages.

5.2 Liveness property

In a fault-prone asynchronous model, it is not feasible to require that a group communication service be live in every execution. The only way to specify useful liveness properties without strengthening the communication model is to make these properties *conditional* on the underlying network behavior (as specified, for example, in [39, 29, 27]). Since our GCS uses an external membership service, we condition the GCS liveness on the behavior of the membership service.

We define the liveness property for a restricted set of executions in which a component stabilizes from some point on forever thereafter.

Property 5.2.1 (View stability)
Let GCS *be a group communication service whose interface with its clients consists of* `send`*,* `deliver`*, and* `view` *events as defined in the automaton signature in Figure 5.1. Furthermore, assume that the* GCS *uses a membership service* MBRSHP *described in Chapter 4.*

A view $\mathtt{v}$ *eventually becomes* stable *in a given timed execution* $\alpha = \mathtt{s}_0, \pi_1, \mathtt{s}_1, \pi_2, \ldots$ *of the* GCS *service, provided the* MBRSHP.$\mathtt{view}_\mathtt{p}(\mathtt{v})$ *event occurs in* α *for every* $\mathtt{p} \in \mathtt{v.set}$ *and is followed by neither* MBRSHP.$\mathtt{view}_\mathtt{p}$ *nor* MBRSHP.$\mathtt{start\_change}_\mathtt{p}$ *events.*

Given an execution that satisfies Property 5.2.1, the liveness property requires each end-point in the stable view to eventually deliver this last view and all the messages sent in this view to its client. Formally:

Property 5.2.2 (Liveness)
Let GCS *be a group communication service whose interface with its clients consists of* `send`*,* `deliver`*, and* `view` *events as defined in the automaton signature in Figure 5.1. Furthermore, assume that the* GCS *uses a membership service* MBRSHP *described in Chapter 4.*

Let α *be a fair execution of* GCS *in which view* $\mathtt{v}$ *eventually becomes stable (Property 5.2.1).*

Then at each $\mathtt{p} \in \mathtt{v.set}$*,* GCS.$\mathtt{view}_\mathtt{p}(\mathtt{v})$ *eventually occurs. Moreover, for every* GCS.$\mathtt{send}_\mathtt{p}(\mathtt{m})$ *that occurs after* GCS.$\mathtt{view}_\mathtt{p}(\mathtt{v})$*, and for every* $\mathtt{q} \in \mathtt{v.set}$*,* GCS.$\mathtt{deliver}_\mathtt{q}(\mathtt{p}, \mathtt{m})$ *also occurs.*

It is important to note that although our liveness property requires the GCS to be live only in *certain* executions, any implementation that satisfies this property has to attempt to be live in *every* execution because it cannot test the external condition of the membership becoming stable. Also note that, even though membership stability is formally required to last forever, in practice it only has to hold "long enough" for the GCS to reconfigure, as explained in [36, 46]. However, we cannot explicitly introduce the bound on this time period in a fully asynchronous model, since it depends on external conditions such as message latency, process scheduling, and processing time.

Chapter 6

The Virtually Synchronous Group Multicast Algorithm

In this chapter we present an algorithm for a group communication service, GCS, that satisfies the specifications in Chapter 5. The group communication service is implemented by a collection of GCS end-points, each running the same algorithm. Figure 6.1 (a) shows the interaction of a GCS end-point with its environment: a membership service MBRSHP and a reliable FIFO multicast service CO_RFIFO; these services are assumed to satisfy the specifications of Chapter 4. The end-point interacts with its application client by accepting the client's send-requests and by delivering application messages and views to the client. The end-point uses the CO_RFIFO service to send messages to other GCS end-points and to receive messages sent by other GCS end-points. When necessary, the end-point uses the `reliable` action to inform CO_RFIFO of the set of end-points to which CO_RFIFO must maintain reliable (gap-free) FIFO connections. The GCS end-point also receives `start_change` and `view` notifications from the membership service.

The algorithm running at each end-point is constructed incrementally using the inheritance-based methodology presented in Chapter 3. We proceed in three steps, at each step adding support for a new property (see Figure 6.1 (b)):

- First, in Section 6.1, we present an algorithm WV_RFIFO_p for an end-point of the within-view reliable FIFO multicast service specified in Section 5.1.1, and argue that this service satisfies safety specification WV_RFIFO : SPEC and liveness Property 5.2.2.
- Then, in Section 6.2, we add support for the Virtually-Synchronous Delivery and Transitional Set properties specified in Sections 5.1.2 and 5.1.3. We present a child VS_RFIFO+TS_p of WV_RFIFO_p, and argue that the service built from VS_RFIFO+TS_p end-points satisfies safety specifications VSRFIFO : SPEC and TS : SPEC, and liveness Property 5.2.2.
- Finally, in Section 6.3, we add support for the Self Delivery property specified in Section 5.1.4. The resulting automaton VS_RFIFO+TS+SD_p models a complete GCS end-point. Due to the use of inheritance, the service built from these end-points automatically satisfies safety specifications WV_RFIFO : SPEC, VSRFIFO : SPEC, and TS : SPEC.

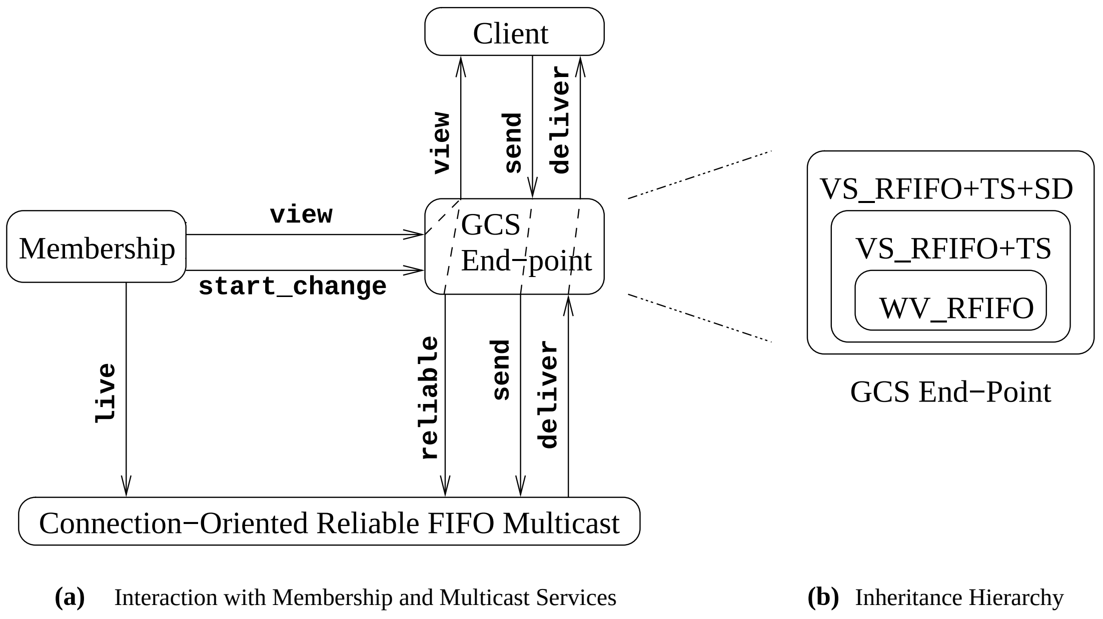


Figure 6.1: A GCS end-point and its environment.

We argue that it also satisfies safety specification SELF : SPEC and liveness Property 5.2.2.

In the presented automata, each locally controlled action is defined to be a task by itself, which means that, if it becomes and stays enabled, it eventually gets executed.

When composing automata into a service, actions of the type $\texttt{mbrshp.start\_change}_\texttt{p}(\texttt{id}, \texttt{set})$ are linked with $\texttt{co\_rfifo.live}_\texttt{p}(\texttt{set})$, and actions of the type $\texttt{mbrshp.view}_\texttt{p}(\texttt{v})$ are linked with $\texttt{co\_rfifo.live}_\texttt{p}(\texttt{v.set})$; the "link" operation can be formally expressed using the signature extension construct. When MBRSHP and CO_RFIFO actions are linked this way, the live_set[p] variable of CO_RFIFO matches the MBRSHP's perception of which end-points are alive and connected to p. (We assume that every permanently disconnected end-point is eventually excluded by either a start_change or a view notification.) In the composed system, all output actions except the application interface are reclassified as internal.

For simplicity of the code, the presented automata do not include certain practical optimizations such as, for example, garbage collection; we point out some of the important ones in Section 6.4.

6.1 Within-view reliable FIFO multicast algorithm

In this section we present the $\text{WV\_RFIFO}_\texttt{p}$ algorithm running at an end-point p of a basic group communication service, WV_RFIFO. The end-point algorithm is quite simple: It relies on the MBRSHP service to form and deliver views involving end-point p; the end-point forwards these views to its client. The algorithm also relies on the CO_RFIFO service to provide reliable gap-free FIFO multicast communication. When the end-point receives a message-send request from its client, it uses CO_RFIFO to send the message to other end-points in the client's current view. The end-point delivers to its client the messages received from other end-points via CO_RFIFO, provided the client's current view matches the views in which the messages were sent. The algorithm keeps track of the views in which messages are sent using the following technique: each time the end-point delivers a view v to its client, it sends a special view_msg message to the end-points in v.set, informing them that the end-point's future messages will be sent in view v. Reliable delivery of messages is ensured by having CO_RFIFO maintain a reliable connection to every member of the end-point's view.

```
AUTOMATON WV_RFIFO$_p$

Type:
  ViewMsg = View
  FwdMsg  = Proc × View × AppMsg × Int

Signature:
  Input:  send$_p$(m), AppMsg m
          co_rfifo.deliver$_{q,p}$(m), Proc q,
                (AppMsg + ViewMsg + FwdMsg) m
          mbrshp.view$_p$(v), View v

  Output: deliver$_p$(q, m), Proc q, AppMsg m
          co_rfifo.send$_p$(set, m), SetOf(Proc) set,
                (AppMsg + ViewMsg + FwdMsg) m
          co_rfifo.reliable$_p$(set), SetOf(Proc) set
          view$_p$(v), View v

State:
  // Variables for handling application messages
  For all Proc q, View v: SequenceOf(AppMsg$_\perp$)
      msgs[q][v], initially empty
  Int last_sent, initially 0
  For all Proc q: Int last_rcvd[q],  initially 0
  For all Proc q: Int last_dlvrd[q], initially 0

  // Variables for handling views and view messages
  View current_view, initially v$_p$
  View mbrshp_view,  initially v$_p$
  For all Proc q: View view_msg[q], initially v$_q$

  SetOf(Proc) reliable_set, initially v$_p$.set

Transitions:
  INPUT  mbrshp.view$_p$(v)
  eff: mbrshp_view ← v

  OUTPUT  view$_p$(v)
  pre: v = mbrshp_view ≠ current_view
  eff: current_view ← v
       last_sent   ← 0
       (∀ q) last_dlvrd[q] ← 0

  OUTPUT  co_rfifo.reliable$_p$(set)
  pre: current_view.set ⊆ set
       reliable_set ≠ set
  eff: reliable_set ← set

  OUTPUT  co_rfifo.send$_p$(set, ⟨'view_msg', v⟩)
  pre: view_msg[p] ≠ current_view
       current_view.set ⊆ reliable_set
       set = current_view.set - {p}
       v = current_view
  eff: view_msg[p] ← current_view

  INPUT  co_rfifo.deliver$_{q,p}$(⟨'view_msg', v⟩)
  eff: view_msg[q]  ← v
       last_rcvd[q] ← 0

  INPUT  send$_p$(m)
  eff: append m to msgs[p][current_view]

  OUTPUT  deliver$_p$(q, m)
  pre: m = msgs[q][current_view][last_dlvrd[q]+1]
  eff: last_dlvrd[q] ← last_dlvrd[q] + 1

  OUTPUT  co_rfifo.send$_p$(set, ⟨'app_msg', m⟩)
  pre: view_msg[p] = current_view
       set = current_view.set - {p}
       m = msgs[p][current_view][last_sent + 1]
  eff: last_sent ← last_sent + 1

  INPUT  co_rfifo.deliver$_{q,p}$(⟨'app_msg', m⟩)
  eff: msgs[q][view_msg[q]] [last_rcvd[q]+1] ← m
       last_rcvd[q] ← last_rcvd[q] + 1

  OUTPUT  co_rfifo.send$_p$(set,⟨'fwd_msg',r,v,m,i⟩)
  pre: (p ∉ set)   ∧   (m = msgs[r][v][i])

  INPUT  co_rfifo.deliver$_{q,p}$(⟨'fwd_msg',r,v,m,i⟩)
  eff:  msgs[r][v][i] ← m
```

Figure 6.2: Within-view reliable FIFO multicast end-point automaton.

Figure 6.2 models the WV_RFIFO$_p$ algorithm as an automaton. The signature defines the interface through which end-point p interacts with its client and with the MBRSHP and

CO_RFIFO services.

When a view $\mathtt{v}$ is received from MBRSHP via action $\mathtt{mbrshp.view_p(v)}$, end-point $\mathtt{p}$ saves it in a variable mbrshp_view and then delivers $\mathtt{v}$ to its client by executing action $\mathtt{view_p(v)}$. Variable current_view contains the last view delivered to the client. The precondition, $\mathtt{v} = \mathtt{mbrshp\_view} \neq \mathtt{current\_view}$, on the $\mathtt{view_p(v)}$ action ensures that $\mathtt{v}$ is indeed the last view received from MBRSHP and that it has not already been delivered to the client. After end-point $\mathtt{p}$ delivers view $\mathtt{v}$ to its client, it sends a view_msg containing $\mathtt{v}$ to the rest of the members of current_view.set by using action $\mathtt{co\_rfifo.send_p(set}, \langle\text{'}\textbf{view\_msg}\text{'}, \mathtt{v}\rangle)$ with $\mathtt{set} = \mathtt{current\_view.set} - \{\mathtt{p}\}$ and $\mathtt{v} = \mathtt{current\_view}$. Variable $\mathtt{view\_msg[p]}$ contains the last view sent as a view_msg. The first precondition, $\mathtt{view\_msg[p]} \neq \mathtt{current\_view}$, on $\mathtt{co\_rfifo.send_p(set}, \langle\text{'}\textbf{view\_msg}\text{'}, \mathtt{v}\rangle)$ ensures that each view_msg is sent only once, and the second precondition, $\mathtt{current\_view.set} \subseteq \mathtt{reliable\_set}$, ensures that, prior to sending the view_msg, end-point $\mathtt{p}$ has requested CO_RFIFO to maintain reliable connection to every member of the client's view by executing action $\mathtt{co\_rfifo.reliable_p(set)}$, which sets variable reliable_set to the value of set. When end-point $\mathtt{p}$ receives a view_msg from some end-point $\mathtt{q}$ via the $\mathtt{co\_rfifo.deliver_{q,p}}(\langle\text{'}\textbf{view\_msg}\text{'}, \mathtt{v}\rangle)$ action, it stores $\mathtt{v}$ in a variable $\mathtt{view\_msg[q]}$.

End-point $\mathtt{p}$ maintains a queue $\mathtt{msgs[q][v]}$ per each end-point $\mathtt{q}$ and view $\mathtt{v}$; these queues are used for storing application messages received from other end-points via $\mathtt{co\_rfifo.deliver_{q,p}}$ and from the end-point's own client via $\mathtt{send_p}$. When action $\mathtt{send_p(m)}$ occurs, message $\mathtt{m}$ is appended to $\mathtt{msgs[p][current\_view]}$. The end-point maintains the following indices that enforce message handling in the order of their appearances in the msgs queues:

- last_sent points to the last application message $\mathtt{m}$ on $\mathtt{msgs[p][current\_view]}$ that was sent using $\mathtt{co\_rfifo.send_p(set}, \langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}\rangle)$;
- $\mathtt{last\_rcvd[q]}$, for each end-point $\mathtt{q}$, points to the last message $\mathtt{m}$ on $\mathtt{msgs[q][view\_msg[q]]}$ that was delivered to $\mathtt{p}$ by $\mathtt{co\_rfifo.deliver_{q,p}}(\langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}\rangle)$;
- $\mathtt{last\_dlvrd[q]}$, for each end-point $\mathtt{q}$, points to the last message $\mathtt{m}$ on $\mathtt{msgs[q][current\_view]}$ that was delivered to $\mathtt{p}$'s client using $\mathtt{deliver_p(q, m)}$.

The first precondition of $\mathtt{co\_rfifo.send_p(set}, \langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}\rangle)$ ensures that a view_msg containing current_view has been already sent to everybody in $\mathtt{set} = \mathtt{current\_view} - \{\mathtt{p}\}$. The preconditions on sending view_msg s, imply that CO_RFIFO already maintains a reliable connection to everyone in set, when $\mathtt{co\_rfifo.send_p(set}, \langle\text{'}\textbf{app\_msg}\text{'}, \mathtt{m}\rangle)$ occurs.

Automaton $\text{WV\_RFIFO}_\mathtt{p}$ also implements auxiliary functionality that allows end-point $\mathtt{p}$ to forward an application message received from some end-point to some other end-points. Specifically, using $\mathtt{co\_rfifo.send_p(set}, \langle\text{'}\textbf{fwd\_msg}\text{'}, \mathtt{r, v, m, i}\rangle)$, end-point $\mathtt{p}$ can forward to some set of end-points the $\mathtt{i}$ th message, $\mathtt{m}$, sent by the client at $\mathtt{r}$ in view $\mathtt{v}$. In turn, when end-point $\mathtt{p}$ receives $\mathtt{co\_rfifo.deliver_{q,p}}(\langle\text{'}\textbf{fwd\_msg}\text{'}, \mathtt{r, v, m, i}\rangle)$, it stores the forwarded message $\mathtt{m}$ in the $\mathtt{i}$ th location of the $\mathtt{msgs[r][v]}$ queue. The code of $\text{WV\_RFIFO}_\mathtt{p}$ does not specify a particular strategy for forwarding messages; the strategy can be chosen nondeterministically. Such a strategy can be specified by more refined versions of the algorithm and/or by modifications of $\text{WV\_RFIFO}_\mathtt{p}$, as we do in the $\text{VS\_RFIFO+TS}_\mathtt{p}$ modification of the $\text{WV\_RFIFO}_\mathtt{p}$ automaton in Section 6.2 below.

Leaving a certain level of non-determinism at the parent automaton, with the intention of resolving it later at the child automaton, is a technique similar to the use of *abstract methods* or *pure virtual methods* in object-oriented methodology. We use the same technique in the $\texttt{co\_rfifo.reliable}_p(\texttt{set})$ action when we require `set` to be a nondeterministic superset of `current_view.set`. The VS_RFIFO+TS$_p$ modification of WV_RFIFO$_p$ places additional preconditions on this action, thereby specifying precise values for the `set` argument.

The WV_RFIFO automaton resulting from the composition of all the end-point automata and the MBRSHP and CO_RFIFO automata models the WV_RFIFO service. The automaton satisfies the safety properties specified by WV_RFIFO : SPEC: it preserves the Local Monotonicity and Self Inclusion properties of view deliveries guaranteed by the MBRSHP service; and it also extends the gap-free FIFO-ordered message delivery of CO_RFIFO with the Within-View Delivery property. The Within-View Delivery is achieved by delivering messages to the clients only if the views in which the messages were sent match the clients' current views.

Chapter 7.1 contains a simulation from WV_RFIFO to WV_RFIFO : SPEC: Actions of automaton WV_RFIFO : SPEC involving $\texttt{view}_p(\texttt{v})$, $\texttt{send}_p(\texttt{m})$, and $\texttt{deliver}_p(\texttt{q},\texttt{m})$ are simulated when WV_RFIFO takes the corresponding $\texttt{view}_p(\texttt{v})$, $\texttt{send}_p(\texttt{m})$, and $\texttt{deliver}_p(\texttt{q},\texttt{m})$ actions. Steps of WV_RFIFO involving other actions correspond to empty steps of WV_RFIFO : SPEC. We define the following function `R` that maps every reachable state `s` of WV_RFIFO to a reachable state of WV_RFIFO : SPEC, where `s[p].var` denotes an instance of a variable `var` of end-point `p` in a state `s`:

```
R(s ∈ ReachableStates(WV_RFIFO)) = t ∈ ReachableStates(WV_RFIFO : SPEC), where
  For each Proc p, View v:       t.msgs[p][v]   =   s[p].msgs[p][v]
  For each Proc p, Proc q: t.last_dlvrd[p][q]   =   s[q].last_dlvrd[p]
  For each Proc p:          t.current_view[p]   =   s[p].current_view
```

Lemma 7.1.1 states that `R` is a refinement mapping from WV_RFIFO to WV_RFIFO : SPEC; the proof relies on a number of invariant assertions, stated and proved in Chapter 7.1 as well.

The WV_RFIFO automaton also satisfies liveness Property 5.2.2. Consider a fair execution in which each end-point `p` in `v.set` receives the same view `v` from the membership and no view events afterwards. Starting from the time the $\texttt{mbrshp.view}_p(\texttt{v})$ action occurs, the $\texttt{view}_p(\texttt{v})$ action stays enabled; therefore it eventually happens due to the fairness of the execution. After view `v` is delivered to the clients, all messages sent in view `v` are also eventually delivered to the clients. This is due to the liveness property of CO_RFIFO, which guarantees that messages sent between live and connected end-points (as perceived by the membership service) are eventually delivered to their destinations. We prove these claims formally for the complete GCS algorithm in Chapter 8.

6.2 Adding support for Virtually Synchronous Delivery and Transitional Sets

The WV_RFIFO service of the previous section guarantees that each member `p` of a view `v` receives *some* prefix of the FIFO ordered stream of messages sent by every member `q` in `v`.

In this section, we modify the WV_RFIFO$_p$ algorithm to yield an end-point VS_RFIFO+TS$_p$ of a service, VS_RFIFO+TS, that, in addition to the semantics provided by WV_RFIFO, guarantees that those members that transition from `v` in to *the same* view `v′`, receive not just *some* but *the same* prefix of the message stream sent by each member `q` in `v`. This is the Virtually-Synchronous Delivery property, the key property of Virtual Synchrony semantics (see Section 5.1.2). Overall, the VS_RFIFO+TS service satisfies the VSRFIFO : SPEC and TS : SPEC safety specifications, as well as liveness Property 5.2.2; we prove these claims respectively in Sections 7.2 and 7.3 and in Chapter 8.

Algorithm Overview

In a nutshell, here is how VS_RFIFO+TS$_p$ computes transitional sets and enforces Virtually-Synchronous Delivery: When end-point `p` is notified via $\texttt{start\_change}_p(\texttt{cid}, \texttt{set})$ of the MBRSHP's attempt to form a new view, `p` sends via CO_RFIFO a synchronization message tagged with `cid` to every end-point in `set`. The synchronization message includes `p`'s current view `v` and a mapping `cut`, such that `cut(q)` is the index of the last message from each `q` in `v.set` that `p` commits to deliver in view `v`.

End-point `p` may receive subsequent $\texttt{start\_change}_p(\texttt{cid}, \texttt{set})$ notifications from MBRSHP. When such a notification includes a new `cid`, `p` sends a new synchronization message, with a freshly made cut, to the proposed `set`; otherwise, when the `cid` is the same as the last one, `p` simply forwards the last synchronization message to the joining end-points, that is, to the end-points of the current `set` that were not listed in the previously proposed membership.

Once `p` receives via $\texttt{view}_p(\texttt{v}')$ a new view `v′` from MBRSHP and a synchronization message tagged with `v′.startId(q)` from each end-point `q` in $\texttt{v.set} \cap \texttt{v}'\texttt{.set}$, `p` computes a transitional set from `v` to `v′` and decides on which messages it needs to deliver to its client in view `v` before delivering view `v′`. A transitional set `T` from `v` to `v′` is computed to include every client `q` in $\texttt{v.set} \cap \texttt{v}'\texttt{.set}$ whose synchronization message tagged with `v′.startId(q)` contains `p`'s current view `v`. For each client `r` in `v.set`, end-point `p` decides to deliver all the messages of `r` that appear in the cut of the synchronization message of any member `q` of `T`. Section 6.2 describes two message-forwarding strategies that ensure `p`'s ability to actually deliver all the messages it decides to deliver. After `p` delivers all these messages to its client, it then delivers to its client the new view `v′` along with the transitional set `T`.

Virtually-Synchronous Delivery follows from the fact that all end-points transitioning from view `v` to `v′` consider the same synchronization messages, compute the same set `T`, and hence use the same data to decide which messages to deliver in view `v` before delivering view `v′`. Set `T` satisfies Definition 5.1.1 of a transitional set from `v` to `v′` because (a) every end-point that computes `T` is itself included in `T`, and (b) no end-point `q` in `T` is allowed to deliver `v′` while in some view other than `v` because `v′.startId(q)` is linked through `q`'s synchronization message to `v`.

Illustration

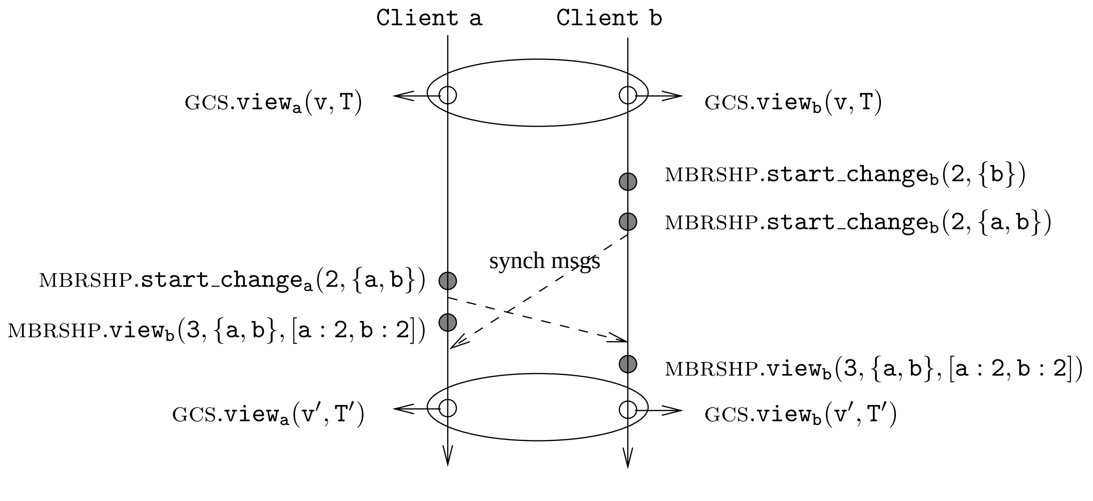
The following example demonstrates how our algorithm does not waste resources on forming and synchronizing views that are known to be obsolete.

Example 6.2.1 *Figure 6.3 presents a sample execution of* GCS *involving two clients,* a *and* b*. The two vertical arrows represent time passage at each client; small empty circles represent client-level events, and gray circles –* MBRSHP*-level events.*

Initially, both clients receive the same view $v = \langle 2, \{a, b\}, [a : 1, b : 1]\rangle\rangle$ *from their* GCS *end-points,* GCS_a *and* GCS_b*; the ellipse encircling these view events highlights the fact that the delivered views are the same.*

Then, at some point, the MBRSHP *service notifies* GCS_b *that it is starting to form a view without* a*. While doing so,* MBRSHP *detects that* a *is connected to* b *afterall, so it changes the membership of the forming view to* $\{a, b\}$*. As a result, end-point* GCS_b *forwards to* GCS_a *its latest synchronization message (denoted by a dashed arrow).*

End-point GCS_a *is also notified by* MBRSHP *of the* MBRSHP*'s attempt to form a new view with* b*; this causes* GCS_a *to send a synchronization message to* GCS_b*.*



Client a
Client b
GCS.view_a(v, T)
GCS.view_b(v, T)
MBRSHP.start_change_b(2, {b})
MBRSHP.start_change_b(2, {a, b})
synch msgs
MBRSHP.start_change_a(2, {a, b})
MBRSHP.view_b(3, {a, b}, [a : 2, b : 2])
MBRSHP.view_b(3, {a, b}, [a : 2, b : 2])
GCS.view_a(v', T')
GCS.view_b(v', T')

Figure 6.3: Handling membership changes while synchronization protocol is running.

When the MBRSHP *service completes its view formation protocol, it delivers the new view* $v' = \langle 3, \{a, b\}, [a : 2, b : 2]\rangle$ *to each* GCS *end-point. When, in addition to these views, the* GCS *end-points receive each-others' synchronization messages, the end-points compute the transitional set for view* v' *to be* $T' = \{a, b\}$*; they also decide which application messages they need to deliver to their clients, deliver these messages, and then deliver to their clients view* v' *and transition set* T'*.*

From the transitional set T'*, clients* a *and* b *know that, due to Virtual Synchrony, they received the same messages while in* v*, and therefore their states are synchronized. Hence, they can avoid state-transfer.*

In addition to demonstrating the benefits of not wasting resources on forming and synchronizing views that are known to be obsolete, Example 6.2.1 also demonstrates that the application too benefits from not seeing obsolete views, as it has to do fewer state transfers (or other similar view processing activity). In contrast to our algorithm, algorithms that do not allow new members to be added to the membership of an already forming view (such as, [8, 47, 12, 82]) lack these advantages. The following example illustrates this.

Example 6.2.2 *When executed in the scenario of Example 6.2.1, algorithms that do not allow new members to be added to the membership of an already forming view would deliver an obsolete view* $\mathtt{v_{mid}}$ *with membership* $\{\mathtt{b}\}$ *to client* b*, and then re-start the view formation and the synchronization protocols anew in order to deliver to* a *and* b *a view with membership* $\{\mathtt{a},\mathtt{b}\}$*. As part of the synchronization protocol,* a *and* b *would first exchange messages to agree upon a common identifier before actually exchanging synchronization messages. At the end, however, the synchronization protocol would* not *synchronize end-points* a *and* b *because they would be transitioning into the new view from different views,* a *from* v *and* b *from* $\mathtt{v_{mid}}$*. Hence, after the clients get the final view from* GCS*, they would still need to synchronize by running a state transfer protocol.*

Algorithm Automaton

Figures 6.2, 6.4 and 6.5, together, contain the code of the VS_RFIFO+TS$_\mathtt{p}$ automaton that models end-point p of the VS_RFIFO+TS service. Figures 6.4 and 6.5 specify how the WV_RFIFO$_\mathtt{p}$ automaton of Figure 6.2 is modified to support Virtually-Synchronous Delivery and Transitional Sets. Figure 6.4 contains the Signature Extension that defines the signatures of new and modified actions; Figure 6.5 contains the State Extension and Transition Restriction defining respectively new state variables and new precondition/effect code. We now describe automaton VS_RFIFO+TS$_\mathtt{p}$ in detail.

```
AUTOMATON VS_RFIFO+TS_p  MODIFIES  WV_RFIFO_p

Type:  SyncMsg  = StartChangeId × View × (Proc → Int)

Signature Extension:
 Input:  mbrshp.start_change_p(id, set), StartChangeId id, SetOf(Proc) set new
         co_rfifo.deliver_q,p(m), Proc q, SyncMsg m  new

 Output: deliver_p(q, m) modifies wv_rfifo.deliver_p(q, m)
         view_p(v, T), SetOf(Proc) T  modifies wv_rfifo.view_p(v)
         co_rfifo.reliable_p(set), SetOf(Proc) set modifies wv_rfifo.co_rfifo.reliable_p(set)
         co_rfifo.send_p(set, m), SetOf(Proc) set, SyncMsg m new
         co_rfifo.send_p(set, m) modifies wv_rfifo.co_rfifo.send_p(set, m), FwdMsg m

 Internal: set_cut_p()  new
```

Figure 6.4: Virtually Synchronous reliable FIFO multicast: Signature Extension.

Upon receiving $\mathtt{mbrshp.start\_change_p(cid,set)}$, VS_RFIFO+TS$_\mathtt{p}$ stores the `cid` and `set` parameters in the `id` and `set` fields of a variable `start_change`. When $\mathtt{start\_change} \neq \bot$, it indicates that VS_RFIFO+TS$_\mathtt{p}$ is engaged in a synchronization protocol, during which it exchanges synchronization messages tagged with `start_change.id` with the end-points in `start_change.set`; after VS_RFIFO+TS$_\mathtt{p}$ sends a view to its client, it sets `start_change` to $\bot$.

Variable sync_set indicates the set of end-points to which a synchronization message tagged with the latest start_change.id has already been sent. When end-point p receives $\texttt{start\_change}_\texttt{p}(\texttt{cid}, \texttt{set})$ with a new cid, sync_set is reset to $\emptyset$ to indicate that a new synchronization message needs to be sent to every end-point in set. However, if the cid is the same as the last one, sync_set is set to sync_set $\cap$ set. This way, the end-point will send its last synchronization message only to the joining end-points (i.e., those in set $-$ sync_set), and not to those to which the message was already sent. Notice that the disconnected end-points (i.e., those that are not in set) are removed from sync_set.

After $\text{VS\_RFIFO+TS}_\text{p}$ receives a $\texttt{start\_change}_\texttt{p}(\texttt{cid}, \texttt{set})$ input from MBRSHP, it executes an internal action, $\texttt{set\_cut}_\texttt{p}()$. This action commits p to deliver to its client all the messages it has so far received from the members of its current view. For each member q of current_view.set, cut(q) is set to the length of the longest continuous prefix of messages in msgs[q][current_view].[1] Action $\texttt{set\_cut}_\texttt{p}()$ results in p's current view being stored in sync_msg[p][start_change.id].view, the committed cut being stored in sync_msg[p][start_change.id].cut, and sync_set being set to {p}.

$\text{VS\_RFIFO+TS}_\text{p}$ specifies precise preconditions on the $\texttt{co\_rfifo.reliable}_\texttt{p}(\texttt{set})$ actions. When $\text{VS\_RFIFO+TS}_\text{p}$ is not engaged in a synchronization protocol (i.e., when start_change $= \perp$), CO_RFIFO is asked to maintain reliable connection just to the end-points in p's current view, current_view.set. When $\text{VS\_RFIFO+TS}_\text{p}$ is engaged in a synchronization protocol, it requires CO_RFIFO to maintain reliable connection to the members of a forming view, start_change.set, as well as to those in current_view.set. Thus, CO_RFIFO avoids loss of messages sent to the disconnected end-points in case these end-points are later added to the forming view.

After setting the cut and telling CO_RFIFO to maintain reliable connection to everyone in current_view.set $\cup$ start_change.set, $\text{VS\_RFIFO+TS}_\text{p}$ uses $\texttt{co\_rfifo.send}_\texttt{p}$ to send the synchronization message sync_msg[p][start_change.id] tagged with start_change.id to the end-points in start_change.set $-$ sync_set, that is, to all those end-points in the proposed membership to which this synchronization message has not already been sent. Afterwards, sync_set is adjusted to start_change.set.

Whenever end-point p receives synchronization messages from other end-points,
via $\texttt{co\_rfifo.deliver}_\texttt{q,p}(\langle\text{`}\textbf{sync\_msg}\text{'}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)$, it saves $\langle \texttt{v}, \texttt{cut}\rangle$ in sync_msg[q][cid].

$\text{VS\_RFIFO+TS}_\text{p}$ restricts delivery of application messages while it is engaged in a synchronization protocol (i.e., when start_change $\neq \perp$ and sync_msg[p][start_change.id] $\neq \perp$): Prior to receiving a new view from MBRSHP, only the messages identified in the cut of its own latest synchronization message, sync_msg[p][start_change.id].cut, can be delivered to the client. After $\texttt{mbrshp.view}_\texttt{p}(\texttt{v})$ occurs, $\text{VS\_RFIFO+TS}_\text{p}$ is allowed to deliver messages identified in the cut sync_msg[q][v.startId(q)].cut received from q, provided q is a member of the transitional set from current_view to v. An end-point q $\in$ current_view.set $\cap$ v.set is considered to be in the transitional set from current_view to v if sync_msg[q][v.startId(q)].view is the same as p's current_view.

[1]The longest continuous prefix can be different from the length of msgs[q][current_view] because forwarded messages may arrive out of order and introduce gaps in the msgs queues.

```
AUTOMATON VS_RFIFO+TS_p  MODIFIES  WV_RFIFO_p

State Extension:
 (StartChangeId × SetOf(Proc))_⊥ start_change, initially ⊥
 For all Proc q, StartChangeId id:  (View v, (Proc → Int) cut)_⊥ sync_msg[q][id], initially ⊥
 SetOf(Proc) sync_set, initially empty
 SetOf((Proc × Proc × View × Int)) forwarded_set, initially empty

Transition Restriction:
 INPUT  mbrshp.start_change_p(cid, set)
 eff: if start_change ≠ ⊥ ∧ start_change.id = cid
         then  sync_set ← sync_set ∩ set
         else  sync_set ← ∅
      start_change ← ⟨cid, set⟩

 OUTPUT  co_rfifo.reliable_p(set)
 pre: start_change = ⊥ ⇒ set = current_view.set
      start_change ≠ ⊥ ⇒ set = current_view.set ∪ start_change.set

 INTERNAL  set_cut_p()
 pre: start_change ≠ ⊥ ∧ sync_msg[p][start_change.id] = ⊥
 eff: Let cut = {⟨q, LongestPrefixOf(msgs[q][current_view])⟩ | q ∈ current_view.set}
      sync_msg[p][start_change.id] ← ⟨current_view, cut⟩
      sync_set ← {p}

 OUTPUT  co_rfifo.send_p(set, ⟨'sync_msg', cid, v, cut⟩)
 pre: start_change ≠ ⊥  ∧  sync_msg[p][start_change.id] ≠ ⊥
      set = (start_change.set - sync_set) ≠ ∅
      set ⊆ reliable_set
      cid = start_change.id ∧ ⟨v, cut⟩ = sync_msg[p][cid]
 eff: sync_set ← start_change.set

 INPUT  co_rfifo.deliver_q,p(⟨'sync_msg', cid, v, cut⟩)
 eff: sync_msg[q][cid] ← ⟨v, cut⟩

 OUTPUT  deliver_p(q, m)
 pre: if (start_change ≠ ⊥ ∧ sync_msg[p][start_change.id] ≠ ⊥) then
        if start_change.id ≠ mbrshp_view.startId(p) then
           last_dlvrd[q]+1 ≤ sync_msg[p][start_change.id].cut(q)
        else  let S = {r ∈ mbrshp_view.set ∩ current_view.set |
                          sync_msg[r][mbrshp_view.startId(r)].view = current_view}
              last_dlvrd[q]+1 ≤ max_{r ∈ S} sync_msg[r][mbrshp_view.startId(r)].cut(q)

 OUTPUT  view_p(v, T)
 pre: v.startId(p) = start_change.id                 // to prevent delivery of obsolete views
      v.set - sync_set = ∅                           // all sync msgs are sent
      last_sent ≥ sync_msg[p][v.startId(p)].cut(p)   // sent out your own msgs
      (∀ q ∈ v.set ∩ current_view.set) sync_msg[q][v.startId(q)] ≠ ⊥
      T = {q ∈ v.set ∩ current_view.set | sync_msg[q][v.startId(q)].view = current_view}
      (∀ q ∈ current_view.set) last_dlvrd[q] = max_{r ∈ T} sync_msg[r][v.startId(r)].cut(q)
 eff: start_change ← ⊥
      sync_set ← ∅

 OUTPUT  co_rfifo.send_p(set,⟨'fwd_msg',r,v,m,i⟩)
 pre: (∀ q ∈ set) (⟨q, r, v, i⟩ ∉ forwarded_set)   ∧   ForwardStrategyPredicate(set, r, v, i)
 eff: (∀ q ∈ set) add ⟨q, r, v, i⟩ to forwarded_set
```

Figure 6.5: Virtually Synchronous reliable FIFO multicast: State Extension & Transition Restriction.

VS_RFIFO+TS_p delivers a view `v` received from MBRSHP and a transitional set `T` to its client when `p` has received a synchronization message `sync_msg`[q][`v.startId`(q)] from every `q` in `current_view.set` ∩ `v.set`, has computed `T`, and has delivered all the application messages identified in the cuts of the members of `T`, as specified by the last three preconditions on $\texttt{view}_p(\texttt{v},\texttt{T})$. The first two preconditions ensure respectively that no new $\texttt{mbrshp.start\_change}_p$ notification was issued after $\texttt{mbrshp.view}_p(\texttt{v})$ and that `p` has sent its synchronization message to everybody in `v.set`. The third precondition specifies that `p` has sent to others all of its own messages indicated in its own cut. All these preconditions work in conjunction with those in $\texttt{wv\_rfifo.view}_p(\texttt{v})$.

Recall from Section 6.1 that WV_RFIFO_p allows for nondeterministic forwarding of other end-points' application mesages. VS_RFIFO+TS_p resolves this nondeterminism by placing two additional preconditions on $\texttt{co\_rfifo.send}_p(\texttt{set}, \langle\text{'}\textbf{fwd\_msg}\text{'}, \texttt{r}, \texttt{v}, \texttt{m}, \texttt{i}\rangle)$: The first checks a variable `forwarded_set` to make sure that message `m` was not previously forwarded to anyone in `set`. The second tests that a certain `ForwardingStrategyPredicate`(`set`, `r`, `v`, `i`) holds. This predicate is designed to ensure that all end-points in the transitional set `T` are able to deliver all the messages that each has committed to deliver in its synchronization message, in particular those sent by disconnected clients. End-points test `ForwardingStrategyPredicate` to decide whether they need to forward any messages to others.

Forwarding Strategy Predicate

We now provide two examples of `ForwardingStrategyPredicate`s. With the first, multiple copies of the same message may be forwarded by different end-points. The second strategy reduces the number of forwarded copies of a message. Many other possible strategies exist. For example, a strategy can employ randomization to decide whether an end-point should forward a message in a certain time slice, and suppress forwarding of messages that have already been forwarded by others.

A simple strategy: With our first strategy, end-point `p` forwards message `m` only if `p` has committed to deliver `m`. In addition, if `m` was originally sent in view `v`, `p` forwards `m` to an end-point `q` only if `p` does not know of any view of `q` later than `v`, and if the latest `sync_msg` from `q` sent in view `v` indicates that `q` has not received message `m`.

```
ForwardingStrategyPredicate(set, r, v, i) ≡
 (∃ cid) (sync_msg[p][cid].view = v  ∧  i ≤ sync_msg[p][cid].cut(r))
 ∧ set = { q |  view_msg[q] ≤ v  ∧  (∃ cid′) (sync_msg[q][cid′].view = v
                ∧ (∄ cid′′ > cid′) sync_msg[q][cid′′].view = v
                ∧ sync_msg[q][cid′].cut(r) < i) }
```

If some end-point `q` is missing a certain message `m`, `m` will be forwarded to `q` by some end-point `p` that has committed to deliver `m`, when `p` learns from `q`'s synchronization message that `q` misses `m`.

Reducing the number of forwarded copies of a message: The second strategy relies on the computed transitional set `T` from view `v` to `v′` to decide which message should be forwarded by which member of the transitional set. Assume that a member u of T misses a message m that was originally sent in v by a non-member r of T, but that was committed to delivery by some other members of T. Among these members, ForwardingStrategyPredicate selects the one with the minimal process-identifier to forward m to u; variations of this predicate may use a different deterministic rule for selecting a member, for example, accounting for network topology or communication costs. The selected end-point, p, forwards the message to u only if view v′ is the latest view known to p, as specified by the first conjunct below. Otherwise, v′ is an obsolete view, so there is no need to help u transition in to v′. The described strategy does not forward to u ∈ T messages from the members of T because u is guaranteed to receive these messages directly from their original senders (unless v′ becomes obsolete because of further view changes occur).

```
ForwardingStrategyPredicate(set, r, v, i) ≡
  Let v′ = mbrshp_view ∧                          // latest view known to { p}
  sync_msg[p][v′.startId(p)] ≠ ⊥ ∧                // already sent own sync_msg
  Let v = sync_msg[p][v′.startId(p)].view ∧
  (∀ q ∈  v.set ∩ v′.set) sync_msg[q][v′.startId(q)] ≠ ⊥  ∧   // received right sync_msgs
  Let T = {q ∈ v.set ∩ v′.set | sync_msg[q][v′.startId(q)].view = v} ∧
  r ∉ T   ∧                            // only forward messages from end-point not in T
  set = {u ∈ T  | sync_msg[u][v′.startId(u)].cut(r) < i }  ∧
  p = min{u ∈ T  | sync_msg[u][v′.startId(u)].cut(r) ≥ i }
```

If all end-points receive the same view from MBRSHP, only one copy of m will be forwarded to each u. In rare cases, however, when MBRSHP delivers different views to different end-points, more than one end-point may forward the same message m to the same end-point u.

Each end-point waits to receive a new view from MBRSHP and all the right synchronization messages before it forwards messages to others. Thus, compared to the first strategy, this strategy reduces the communication traffic at the cost of slower recovery of lost messages.

Correctness Argument Overview

The VS_RFIFO+TS automaton, resulting from the composition of all end-point automata and the MBRSHP and CO_RFIFO automata, satisfies the VSRFIFO : SPEC and TS : SPEC safety specifications, as well as Liveness Property 5.2.2, as we formally prove in Chapters 7.2, 7.3, and 8, respectively. Below we give highlights of these proofs.

VSRFIFO : SPEC is a modification of WV_RFIFO : SPEC. The proof that VS_RFIFO+TS satisfies VSRFIFO : SPEC reuses the proof that WV_RFIFO satisfies WV_RFIFO : SPEC and involves reasoning about only how VSRFIFO : SPEC modifies WV_RFIFO : SPEC. The proof extends refinement mapping R between WV_RFIFO and WV_RFIFO : SPEC with a mapping R_n. R_n maps the cut used by the end-points of VS_RFIFO+TS to move from a view v to a view v′ to the cut[v][v′] variable of VSRFIFO : SPEC. The proof depends on Invariant 7.2.1 and Corollary 7.2.1, which state that all end-points that move from a view v to a view v′ use the same synchronization messages, compute the same transitional set T, and therefore, use the same cut.

The proof in Chapter 7.3 shows that VS_RFIFO+TS satisfies TS : SPEC. The proof augments VS_RFIFO+TS_p with a *prophecy variable* that guesses, at the time end-point p receives a $\texttt{start\_change}_p(\texttt{cid}, \texttt{set})$ notification from MBRSHP, possible future views that may contain cid in their startId(p) mappings. For each of these views v', VS_RFIFO+TS simulates a $\texttt{set\_prev\_view}_p(v')$ action of TS : SPEC, thereby fixing the previous view of v' to be p's current view v.

In a fair execution of VS_RFIFO+TS in which the same last view v' is delivered to all its members and no start_change events subsequently occur, the three preconditions on the $\texttt{view}_p(v', T_p)$ delivery are eventually satisfied for every $p \in v'.\texttt{set}$:

1. Condition $v'.\texttt{startId}(p) = \texttt{start\_change.id}$ remains true since the execution has no subsequent start_change events at p.
2. End-point p eventually receives synchronization messages tagged with the "right" cid from every member of $\texttt{v.set} \cap v'.\texttt{set}$ because they keep taking steps towards reliably sending these synchronization messages to p (by low-level fairness of the code) and because CO_RFIFO eventually delivers these messages to p (by the liveness assumption on CO_RFIFO).
3. End-point p eventually receives and delivers all the messages committed to in the cuts of the members of the transitional set T_p because for each such message there is at least one end-point in T_p that has the message in its msgs buffer and that will reliably forward it to p (according to the ForwardingStrategyPredicate) if necessary. Also, p never delivers any messages beyond those committed to in the cuts of the members of T_p because of the precondition on application message delivery.

6.3 Adding support for Self Delivery

As a final step in constructing the automaton that models an end-point of our group communication service, GCS_p, we add support for Self Delivery to the VS_RFIFO+TS_p automaton presented above. Self Delivery requires each end-point to deliver to its client all the messages the client sends in a view, before moving on to the next view.

In order to implement Self Delivery, Virtually-Synchronous Delivery, and Within-View Delivery together in a live manner, each end-point must *block* its client from sending new messages while a view change is taking place (as proven in [43]). Therefore, we add to VS_RFIFO+TS_p an output action block and an input action block_ok. We assume that the client at end-point p has the matching actions and that it eventually responds to every block request with a block_ok response and subsequently refrains from sending messages until a view is delivered to it. In Section 7.4, we formalize this requirement as an abstract client automaton.

The GCS_p automaton appears in Figure 6.6. After receiving the first start_change notification in a given view, end-point p issues a block request to its client and awaits receiving a block_ok response before executing $\texttt{set\_cut}_p()$. As a result of $\texttt{set\_cut}_p()$, p commits to deliver all the messages its client has sent in the current view. Therefore, p has to deliver

```
AUTOMATON GCS_p = VS_RFIFO+TS+SD_p   MODIFIES  VS_RFIFO+TS_p

Signature Extension:
Input:  block_ok_p() new                        Output: block_p() new
Internal: set_cut_p() modifies set_cut_p()              view_p(v,T) modifies vs_rfifo+ts.view_p(v,T)

State Extension:
block_status ∈ {unblocked, requested, blocked}, initially unblocked

Transition Restriction:
  INTERNAL  set_cut_p()                                   OUTPUT  block_p()
  pre: block_status = blocked                             pre: start_change ≠ ⊥
                                                               block_status = unblocked
                                                          eff: block_status ← requested

  OUTPUT  view_p(v,T)                                     INPUT  block_ok_p()
  eff: block_status ← unblocked                           eff: block_status ← blocked
```

Figure 6.6: $\mathtt{GCS_p}$ end-point automaton.

all these messages before moving on to a new view, and Self Delivery is satisfied. Due to the use of inheritance, the GCS automaton preserves all the safety properties satisfied by its parent. Since end-point p has its own messages on the msgs[p][p] queue and can deliver them to its client, liveness is also preserved. Thus, GCS satisfies all the properties we have specified in Chapter 5.

6.4 Optimizations and Extensions

Having formally presented the basic algorithm for an end-point of our Virtually-Synchronous GCS, we now discuss several optimizations and extensions that can be added to the algorithm to make its implementation more practical. Specifically, we discuss ways to reduce the size and number of synchronization messages, as well as to avoid the use of non-volatile storage. We also discuss garbage collection.

The first optimization that reduces the size of synchronization messages relies on the following observation: An end-point p does not need to send its current view and its cut to end-points that are not in current_view.set because p cannot be included in their transitional sets. However, these end-points still need to hear from p if p is in their current views. Therefore, end-point p could send a smaller synchronization message to the end-points in start_change.set − current_view.set, containing its start_change.id only (but neither a view nor a cut). This message would be interpreted as saying "I am not in your transitional set", and the recipients of this message would know not to include p in their transitional sets for views v′ with v′.startId(p) = p's start_change.id. When using this optimization, p also does not need to include its current view in the synchronization messages sent to current_view.set − start_change.set, since the view information can be deduced from p's view_msg.

An additional optimization can be used if we strengthen the membership specification to require a mbrshp.start_change with a new identifier to be sent every time MBRSHP changes its mind about the membership of a forming view. In this case, the latest mbrshp.start_change has the same membership as the delivered mbrshp.view. Therefore, the synchronization messages can be shortened to not include information about application messages delivered from end-points in start_change.set ∩ current_view.set: for an end-point p, end-points that have p in their transitional sets will deliver all the application messages that p sent before its synchronization message.

Other optimizations can reduce the total number of messages sent during synchronization protocol by all end-points. A simple way to do this is to transform the algorithm into a leader-based one, as [93, 82]. A more scalable approach was suggested by Guo et al. [47]. Their algorithm uses a two-level hierarchy for message dissemination in order to implement Virtual Synchrony: end-points send synchronization messages to their designated leaders, which in turn exchange only the cumulative information among themselves. Also, the number of messages exchanged to synchronize multiple groups can be reduced, as suggested in [18, 81], by aggregating information pertaining to multiple groups into a single message.

Another optimization addresses the use of stable storage. Recall that in Chapter 4 we assumed that end-points keep their running states on stable storage, and therefore, recover with their state intact. However, our group multicast service does provide meaningful semantics even when GCS end-points maintain their running state on volatile storage. When an end-point p recovers after a crash, it can start executing with its state reset to an initial value with current_view being the singleton view v_p. It needs to contact the MBRSHP service to be re-admitted to its groups. The client would refrain from sending any messages in its recovered view until it receives a new view from its end-point. This view would satisfy Local Monotonicity and Self Inclusion because these are the properties guaranteed by the MBRSHP service. The specification of Virtually-Synchronous Delivery should be changed so that recovery is interpreted as delivering a singleton view. The remaining safety properties are also preserved because they involve message delivery within a single view.

In a practical implementation of our service, some sort of garbage collection mechanism is required in order to keep the buffer sizes finite. The implementation of [92] discards messages from older views when moving to a new view, and also when learning that they were already delivered to every client in the view. This implementation also discards older synchronization messages: an end-point holds on to only the latest synchronization message it has received from each end-point. This optimization does not violate liveness since discarded synchronization messages necessarily pertain to obsolete views.

Chapter 7

Correctness Proof: Safety Properties

We now formally prove using invariant assertions and simulations that our algorithms satisfies the safety properties of Section 5.1. Proofs done with invariant assertions and simulations are verifiable (even by a computer) because they involve reasoning only about single steps of the algorithm. A review of the standard proof techniques used in this chapter appears in Chapter 2; our incremental verification technique is presented in Chapter 3.

The safety proof is *modular*: we exploit the inheritance-based structure of our specifications and algorithms to reuse proofs. In Section 7.1 we prove correctness of the within-view reliable FIFO multicast service by showing a refinement mapping from WV_RFIFO to WV_RFIFO : SPEC. In Section 7.2 we extend this refinement mapping to map the new state added in VS_RFIFO+TS to that in VSRFIFO : SPEC. In Section 7.3 we prove that VS_RFIFO+TS also simulates TS : SPEC. Finally, in Section 7.4 we extend the refinement above to map the new state of GCS to that of SELF : SPEC. The proof-extension theorem of Chapter 3 implies that the GCS automaton satisfies WV_RFIFO : SPEC, VSRFIFO : SPEC, TS : SPEC, and SELF : SPEC.

7.1 Within-view reliable FIFO multicast

Intuitively, in order to simulate WV_RFIFO : SPEC with WV_RFIFO, we need to show that WV_RFIFO satisfies Self Inclusion and Local Monotonicity for delivered views, and we need to show that the i'th message delivered by q from p in view v is the i'th message sent in view v by the client at p. In order to prove this, we need to show that the algorithm correctly associates messages with the views in which they were sent and with their indices in the sequences of messages sent in these views. We split the proof into three parts: Section 7.1.1 states key invariants, but defers the proof of one of them to Section 7.1.3; Section 7.1.2 contains the simulation proof.

7.1.1 Key Invariants

The following invariant captures the Self Inclusion property.

Invariant 7.1.1 (Self-Inclusion) *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p`*,* `p` $\in$ `s[p].mbrshp_view.set` *and* `p` $\in$ `s[p].current_view.set`*.*

Proof 7.1.1: Immediate from the MBRSHP specification. ■

The Local Monotonicity property follows directly from the precondition, `v.id` $>$ `mbrshp_view`, of the MBRSHP.$\texttt{view}_\texttt{p}$(`v`) actions.

The following invariant relates application messages at different end-points' queues to the corresponding messages on the original senders' queues.

Invariant 7.1.2 (Message Consistency) *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p` *and* `Proc q`*, if* `s[q].msgs[p][v][i] = m`*, then* `s[p].msgs[p][v][i] = m`*.*

This proposition is vacuously true in the initial state because all message queues are empty. For the inductive step, we have to consider actions `co_rfifo`.$\texttt{deliver}_{\texttt{q,p}}$($\langle$'**app_msg**', `m`$\rangle$) and `co_rfifo`.$\texttt{deliver}_{\texttt{q,p}}$($\langle$'**fwd_msg**', `r`, `v`, `m`, `i`$\rangle$), and have to argue that the message `m` they deliver is placed in the right place in `q`'s `msgs` buffer. The proof of this invariant appears in Section 7.1.3, after the simulation proof.

7.1.2 Simulation

Lemma 7.1.1 *The following function* `R` *is a* refinement mapping *from automaton* WV_RFIFO *to automaton* WV_RFIFO : SPEC *with respect to their reachable states.*

```
R(s ∈ ReachableStates(WV_RFIFO)) = t ∈ ReachableStates(WV_RFIFO : SPEC), where
  For each Proc p, View v:       t.msgs[p][v]  =  s[p].msgs[p][v]
  For each Proc p, Proc q: t.last_dlvrd[p][q]  =  s[q].last_dlvrd[p]
  For each Proc p:          t.current_view[p]  =  s[p].current_view
```

Proof 7.1.1:

Action Correspondence: Automaton WV_RFIFO : SPEC has three types of actions. Actions of the types $\texttt{view}_\texttt{p}$(`v`), $\texttt{send}_\texttt{p}$(`m`), and $\texttt{deliver}_\texttt{p}$(`q, m`), are simulated when WV_RFIFO takes the corresponding $\texttt{view}_\texttt{p}$(`v`), $\texttt{send}_\texttt{p}$(`m`), and $\texttt{deliver}_\texttt{p}$(`q, m`) actions. Steps of WV_RFIFO involving other actions correspond to empty steps of WV_RFIFO : SPEC.

Simulation Proof: In the most part the simulation proof is straightforward. Here, we present only the interesting steps:

The fact that the corresponding step of WV_RFIFO : SPEC is enabled when WV_RFIFO takes a step involving $\mathtt{view_p(v)}$ relies on $\mathtt{p} \in \mathtt{mbrshp\_view.set}$ (Invariant 7.1.1).

For the steps involving the $\mathtt{deliver_p(q, m)}$ action, in order to deduce that the corresponding step of WV_RFIFO : SPEC is enabled, we need to know that the message located at index `s[p].last_dlvrd[q] + 1` on the `s[p].msgs[q][s[p].current_view]` queue is the same message that end-point `q` has on its corresponding queue at the same index. This property is implied by Invariant 7.1.2.

Steps that involve receiving original and forwarded application messages from the network simulate empty steps of WV_RFIFO : SPEC. Among these steps the only critical ones are those that deliver a message from `p` to `p` because they may affect `s[p].msgs[p][p]` queue. Since end-points do not send messages to themselves, such steps may not happen. Indeed, CO_RFIFO.$\mathtt{send_p}(\mathtt{set}, \langle \text{'}\mathbf{app\_msg}\text{'}, \mathtt{m}\rangle)$ has a precondition `set = s[p].current_view.set` $- \{\mathtt{p}\}$, and $\mathtt{co\_rfifo.send_p}(\mathtt{set}, \langle \text{'}\mathbf{fwd\_msg}\text{'}, \mathtt{r}, \mathtt{v}, \mathtt{m}, \mathtt{i}\rangle)$ has a precondition $\mathtt{p} \notin \mathtt{set}$. ■

From Lemma 7.1.1 and Theorem 2.2.1 we conclude the following:

Theorem 7.1.2 WV_RFIFO *implements* WV_RFIFO : SPEC *in the sense of trace inclusion.*

7.1.3 Auxiliary Invariants

We now state and prove a number of auxiliary invariants necessary for the proof of the key message consistency invariant (Invariant 7.1.2).

In any view, before an end-point sends a `view_msg` to others (and hence before it sends any application message to others) it tells CO_RFIFO to maintain reliable connection to every member of its current view. The following invariant captures this property.

Invariant 7.1.3 (Connection Reliability) *In every reachable state* `s` *of* WV_RFIFO, *for all* `Proc p`, *if* `s[p].current_view` $=$ `s[p].view_msg[p]`, *then* `s[p].current_view.set` $\subseteq$ `s[p].reliable_set`.

Proof 7.1.3: By induction on the length of the execution sequence; follows directly from the code. ■

After an end-point delivers a new view to its client, it sends a `view_msg` to other members of the view. The stream of `view_msg`s that an end-point sends to others is monotonic because the delivered views satisfy Local Monotonicity. The following invariant captures this property. It states that the subsequence of messages in transit from end-point `p` to end-point `q` consisting solely of the `view_msg` s is monotonically increasing. It also relates the current view of an end-point `p` to the view contained in the `p`'s latest `view_msg` to `q`.

Invariant 7.1.4 (Monotonicity of View Messages) *Let* `s` *be a reachable state of* WV_RFIFO. *Consider the subsequence of messages in* `s.channel[p][q]` *of the* `ViewMsg` *type. Examine the sequence of views included in these view messages, and construct a new* *sequence* seq *of views by pre-pending this view sequence with the element* `s[q].view_msg[p]`*. For all* `Proc p, Proc q`*, the following propositions are true:*

1. *The sequence* seq *is (strictly) monotonically increasing.*
2. *If* `s[p].current_view` $\neq$ `s[p].view_msg[p]`*, then* `s[p].current_view` *is strictly greater then the last (largest) element of* seq.
3. *If* `s[p].current_view` $=$ `s[p].view_msg[p]`*, and if* `q` $\in$ `s[p].current_view.set`*, then* `s[p].current_view` *is equal to the last (largest) element of* seq.

Proof 7.1.4: All three propositions are true in the initial state. We now consider steps involving the critical actions:
$\underline{\text{CO\_RFIFO.}\texttt{lose}(\texttt{p},\texttt{q})}$: The first two propositions remain true because this action throws away only the last message from the CO_RFIFO `s.channel[p][q]`.
The third proposition is vacuously true because `q` can not be in `s[p].current_view.set`. If it were, the CO_RFIFO.`lose(p,q)` action would not be enabled because Invariant 7.1.3 would imply that `s[p].current_view.set` is a subset of `s[p].reliable_set`, which would then imply that `q` $\in$ `s.reliable_set[p]` (because `s[p].reliable_set` $=$ `s.reliable_set[p]`, as can be shown by straightforward induction).
$\underline{\texttt{view}_\texttt{p}(\texttt{v})}$: The first proposition is unaffected. The second proposition follows from the inductive hypothesis and the precondition `v.id` $>$ `s[p].current_view.id`. The third proposition is vacuously true because `s[p].current_view` $\neq$ `s[p].view_msg[p]` as follows from the precondition `v.id` $>$ `s[p].current_view.id` and the fact that, in every reachable state `s`, `s[p].current_view` $\geq$ `s[p].view_msg[p]` (can be proved by straightforward induction).
$\underline{\text{CO\_RFIFO.}\texttt{send}_\texttt{p}(\texttt{set}, \langle\text{'}\textbf{view\_msg}\text{'}, \texttt{v}\rangle)}$: The first proposition is true in the post-state because of the inductive hypothesis of the second proposition. The second proposition is vacuously true in the post-state. The third proposition is true in the post-state because of the effect of this action.
$\underline{\text{CO\_RFIFO.}\texttt{deliver}_{\texttt{p,q}}(\langle\text{'}\textbf{view\_msg}\text{'}, \texttt{v}\rangle)}$: It is straightforward to see that all three propositions remain true in the post-state.

■

History Tags

In order to reason about original application messages traveling on CO_RFIFO channels we need a way to reference, for each of these messages, the view in which it was originally sent and its index in the FIFO-ordered sequence of messages sent in that view. To this end, we augment each original application message $\langle$'**app_msg**', `m`$\rangle$ with two *history tags*, `Hv` and `Hi`, that are set to `current_view` and `last_sent + 1` respectively when CO_RFIFO.$\texttt{send}_\texttt{p}$(`set`, $\langle$'**app_msg**', `m`$\rangle$) occurs. (See Chapter 2 for details on history variables).

```
OUTPUT co_rfifo.send_p(set, ⟨'app_msg', m, Hv, Hi⟩)
pre: ...
     Hv = current_view
     Hi = last_sent + 1
 eff: ...
```

With the history tags, the interface between WV_RFIFO and CO_RFIFO for handling original application messages becomes CO_RFIFO.$\texttt{send}_\texttt{p}(\texttt{set}, \langle$'**app_msg**'$, \texttt{m}, \texttt{Hv}, \texttt{Hi}\rangle)$ and CO_RFIFO.$\texttt{deliver}_{\texttt{p,q}}(\langle$'**app_msg**'$, \texttt{m}, \texttt{Hv}, \texttt{Hi}\rangle)$.

The goal of the next three invariants is to show that, when end-point `q` receives an application message `m` tagged with a history view `Hv` and a history index `Hi`, the current value of `q`'s `view_msg[p]` equals `Hv` and that of `last_rcvd[p] + 1` equals `Hi`.

Invariant 7.1.5 (History View Consistency) *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p, Proc q`*, the following holds: For all messages* $\langle$'**app_msg**'$, \texttt{m}, \texttt{Hv}, \texttt{Hi}\rangle$ *on the* CO_RFIFO `s.channel[p][q]`*, view* `Hv` *equals either the view of the closest preceding view message on* `s.channel[p][q]` *if there is such, or* `s[q].view_msg[p]` *otherwise.*

Proof 7.1.5: Induction. A step involving CO_RFIFO.$\texttt{send}_\texttt{p}(\texttt{set}, \langle$'**app_msg**'$, \texttt{m}, \texttt{Hv}, \texttt{Hi}\rangle)$ follows directly from Invariant 7.1.4 Part 3. The proposition is not affected by steps involving CO_RFIFO.$\texttt{lose}(\texttt{p}, \texttt{q})$ because those may only remove the last messages from the CO_RFIFO `s.channel[p][q]`. The other steps are straightforward. ■

The following invariant states that the value of `s[p].last_sent` equals to the number of application messages that `p` sent in its current view and that are either still in transit on the CO_RFIFO `s.channel[p][q]` or are already received by `q`.

Invariant 7.1.6 *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p` *and for all* `Proc q`*, such that* $\texttt{q} \in \texttt{s[p].current\_view.set} - \{\texttt{p}\}$*, the following is true:*

$$\begin{aligned}&\texttt{s[p].last\_sent} = \\ &\quad \left|\{\texttt{msg} \in \texttt{s.channel[p][q]} \ : \ \texttt{msg} \in \texttt{AppMsg and msg.Hv} = \texttt{s[p].current\_view}\}\right| + \\ &\quad + \begin{cases} \texttt{s[q].last\_rcvd[p]} & \textit{if } \texttt{s[q].view\_msg[p] = s[p].current\_view} \\ 0 & \textit{otherwise.}\end{cases}\end{aligned}$$

Proof 7.1.6: By induction. Consider steps involving the following critical actions:
$\underline{\text{CO\_RFIFO}.\texttt{lose}(\texttt{p}, \texttt{q})}$: Assume that the last message on `s.channel[p][q]` is an application message `msg` with `msg.Hv = s[p].current_view`. If a step involving CO_RFIFO.$\texttt{lose}(\texttt{p}, \texttt{q})$ action could occur, then the proposition would be false. However, as we are going to argue now, $\texttt{q} \in \texttt{s.reliable\_set[p]}$, so such a step cannot occur.

We can prove by induction that $\texttt{msg} \in \texttt{s.channel[p][q]}$ implies $\texttt{s[p].view\_msg[p]} = \texttt{s[p].current\_view}$. By invariant 7.1.3, $\texttt{s[p].current\_view.set} \subseteq \texttt{s[p].reliable\_set}$. Since $\texttt{q} \in \texttt{s[p].current\_view.set}$ and $\texttt{s[p].reliable\_set} = \texttt{s.reliable\_set[p]}$, it follows that $\texttt{q} \in \texttt{s.reliable\_set[p]}$.

$\underline{\texttt{view}_\texttt{p}(\texttt{v})}$: The proposition remains true for steps involving $\texttt{view}_\texttt{p}(\texttt{v})$ action because its effect sets $\texttt{s}'\texttt{[p].last\_sent}$ to `0` and because both summands of the right hand side of the equation also becomes `0`. Indeed, the first summand becomes `0` because CO_RFIFO channels never have messages tagged with views that are larger then the current views of the messages' senders (as can be shown by a simple inductive proof); the second summand becomes `0` because Invariant 7.1.4 Part 2 implies that $\texttt{s}'\texttt{[q].view\_msg[p]} \neq \texttt{s}'\texttt{[p].current\_view}$.

CO_RFIFO.$\mathtt{deliver}_{\mathtt{p,q}}(\langle\text{‘}\textbf{view\_msg}\text{’}, \mathtt{v}\rangle)$: The proposition remains true for steps involving this action because `s[q].view_msg[p]` $\neq$ `s[p].current_view`, as follows immediately from Invariant 7.1.4.

CO_RFIFO.$\mathtt{send}_{\mathtt{p}}(\mathtt{set}, \langle\text{‘}\textbf{app\_msg}\text{’}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$ and
CO_RFIFO.$\mathtt{deliver}_{\mathtt{p,q}}(\langle\text{‘}\textbf{app\_msg}\text{’}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$: For steps involving these actions the truth of the proposition follows immediately from the effects of these actions, the inductive hypotheses, and Invariant 7.1.5.

■

The history index attached to an original application message `m` sent in a view `Hv` that is in transit on a CO_RFIFO channel to end-point `q` is equal to the number of such messages (including `m`) that precede `m` on that channel, plus those (if any) that `q` has already received.

Invariant 7.1.7 (History Indices Consistency) *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p` *and* `Proc q`*, if* $\langle\langle$‘**app_msg**’$, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle\ \rangle$ = `s.channel[p][q][j]` *for some index* `j`*, then*

$$\mathtt{Hi} \;=\; \left|\{\mathtt{msg} \in \mathtt{s.channel[p][q][\ ..\ j]}\ :\ \mathtt{msg} \in \mathtt{AppMsg}\ \mathtt{and}\ \mathtt{msg.Hv} = \mathtt{Hv}\}\right| + \\ + \begin{cases} \mathtt{s[q].last\_rcvd[p]} & \textit{if}\ \mathtt{s[q].view\_msg[p]\ =\ Hv} \\ 0 & \textit{otherwise.} \end{cases}$$

Proof 7.1.7: In the initial state `s.channel[p][q]` is empty. For the inductive step, we consider steps involving the following critical actions:

CO_RFIFO.$\mathtt{lose}(\mathtt{p}, \mathtt{q})$: The proposition remains true since CO_RFIFO.$\mathtt{lose}(\mathtt{p}, \mathtt{q})$ discards only the last messages from the CO_RFIFO `s.channel[p][q]`.

CO_RFIFO.$\mathtt{deliver}_{\mathtt{p,q}}(\langle\text{‘}\textbf{view\_msg}\text{’}, \mathtt{v}\rangle)$: We have to consider the effects on two types of application messages: those associated with view `s[q].view_msg[p]`, and those associated with view `Hv`. Invariants 7.1.4 Part 1 and 7.1.5 imply that there are no application messages with `msg.Hv` =
`s[q].view_msg[p]` on the CO_RFIFO `channel[p][q]`. Thus, the proposition does not apply for such messages. For those messages that have `msg.Hv = Hv`, the proposition remains true because $\mathtt{s'[q].last\_rcvd[p]}$ is set to `0` as a result of this action.

CO_RFIFO.$\mathtt{deliver}_{\mathtt{p,q}}(\langle\text{‘}\textbf{app\_msg}\text{’}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$: Follows immediately from the effect of this action, the inductive hypothesis, and Invariant 7.1.5.

CO_RFIFO.$\mathtt{send}_{\mathtt{p}}(\mathtt{set}, \langle\text{‘}\textbf{app\_msg}\text{’}, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle)$: The inductive step follows immediately from the inductive hypothesis and Invariant 7.1.6.

■

We now prove a generalization of Invariant 7.1.2, which relates application messages either in transit on the CO_RFIFO channels or at end-points' queues to their corresponding messages on the senders' queues.

Invariant 7.1.8 (General Message Consistency) *In every reachable state* `s` *of* WV_RFIFO*, for all* `Proc p` *and* `Proc q`*, the following are true:*

1. *If* $\langle\langle$‘**app_msg**’$, \mathtt{m}, \mathtt{Hv}, \mathtt{Hi}\rangle \in \mathtt{s.channel[p][q]}$*, then* `s[p].msgs[p][Hv][Hi] = m`.

2. *If* $\langle$‘**fwd_msg**’, r, m, v, i$\rangle \in$ s.channel[p][q], *then* `s[r].msgs[r][v][i] = m`.
3. *If* `s[q].msgs[p][v][i] = m`, *then* `s[p].msgs[p][v][i] = m`.

Proof 7.1.8:

Basis: In the initial state all message queues are empty.

Inductive Step: The following are the critical actions:

$\texttt{send}_\texttt{p}\texttt{(m)}$,
$\texttt{co\_rfifo.send}_\texttt{p}\texttt{(set, }\langle$‘**app_msg**’, m, Hv, Hi$\rangle$),
$\texttt{co\_rfifo.deliver}_\texttt{q,p}\texttt{(}\langle$‘**app_msg**’, m, Hv, Hi$\rangle$),
$\texttt{co\_rfifo.send}_\texttt{p}\texttt{(set, }\langle$‘**fwd_msg**’, r, v, m, i$\rangle$),
$\texttt{co\_rfifo.deliver}_\texttt{q,p}\texttt{(}\langle$‘**fwd_msg**’, r, v, m, i$\rangle$).

For steps involving CO_RFIFO.$\texttt{deliver}_\texttt{q,p}$($\langle$‘**app_msg**’, m, Hv, Hi$\rangle$), we use Invariants 7.1.5 and Invariant 7.1.7, which respectively imply that history view `Hv` equals `s[p].view_msg[q]` and that history index `Hi` equals `s[p].last_rcvd[q] + 1`. Inductive steps involving each of the other actions are straightforward. ■

Invariant 7.1.2 is a private case of this invariant.

7.2 Virtual Synchrony

We now show that automaton VS_RFIFO+TS simulates VSRFIFO : SPEC. We prove this by extending the refinement above using the Proof Extension Theorem of Chapter 3.

7.2.1 Invariants

We prove that end-points that move together from one view to the next consider the same synchronization messages and thus compute the same transitional sets and use the same cuts from the members of the transitional set.

Invariant 7.2.1 *In every reachable state* s *of* VS_RFIFO+TS, *for all* `Proc p`, `Proc q`, *and for every*
`StartChangeId cid`,

if `s[q].sync_msg[p][cid]` $\neq \perp$, *then* `s[q].sync_msg[p][cid] = s[p].sync_msg[p][cid]`.

Proof 7.2.1: The proposition is true in the initial state s_0 as all s_0[q].sync_msg[p][cid] $= \perp$. The inductive step involving a $\texttt{set\_cut}_\texttt{p}()$ action is trivial, for it only affects the case $\texttt{q} = \texttt{p}$. The inductive step involving a CO_RFIFO.$\texttt{deliver}_\texttt{p,q}$($\langle$‘**sync_msg**’, cid, v, cut$\rangle$) action follows immediately from the following proposition:

$$\langle\text{‘}\textbf{sync\_msg}\text{’}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle \in \texttt{s.channel[p][q]} \Rightarrow \texttt{s[p].sync\_msg[p][cid]} = \langle\texttt{v, cut}\rangle,$$

which can be proved by straightforward induction. Indeed, there are two critical actions: $\text{CO\_RFIFO}.\texttt{send}_\text{p}(\texttt{set}, \langle\text{‘}\textbf{sync\_msg}\text{’}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)$ – immediate from the code, and $\text{CO\_RFIFO}.\texttt{deliver}_{\text{p,p}}(\langle\text{‘}\textbf{sync\_msg}\text{’}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)$ – may not occur because end-points do not send synchronization messages to themselves. ■

Corollary 7.2.1 *End-points that move together from one view to the next, use the same sets of synchronization messages to calculate transitional sets and message cuts.*

Proof : Consider two end-points that deliver view $\texttt{v}'$ while in view $\texttt{v}$. At the time of delivering view $\texttt{v}'$, each of these end-points has synchronization messages from all end-points in the intersection of these views (second precondition), and these synchronization messages are the same as those at their original end-points (Invariant 7.2.1). Thus, the two end-points calculate the same transitional sets, and use the same cuts from the members of this transitional set. ■

7.2.2 Simulation

We augment VS_RFIFO+TS with a *global* history variable `H_cut` that keeps track of the cuts used for moving between views.

```
For each View v, v′: $(\texttt{Proc} \rightarrow \texttt{Int})_\perp$ H_cut[v][v′], initially ⊥
OUTPUT $\texttt{view}_\texttt{p}$(v, T) modifies wv_rfifo.$\texttt{view}_\texttt{p}$(v)
pre: ...
eff: ...
     (∀ q ∈ Proc)
           H_cut[current_view][v](q) ← $\max_{r \in T}$ (sync_msg[r][v.startId(r)].cut(q))
```

Variable $\texttt{H\_cut}[\texttt{v}][\texttt{v}']$ is updated every time *any* end-point is delivering view $\texttt{v}'$ while in view $\texttt{v}$. Corollary 7.2.1 implies that whenever this happens after $\texttt{H\_cut}[\texttt{v}][\texttt{v}']$ is set for the first time the value of $\texttt{H\_cut}[\texttt{v}][\texttt{v}']$ remains unchanged.

We now extend the refinement mapping `R()` of Lemma 7.1.1 with the new mapping $\texttt{R}_\texttt{n}()$:

$$\texttt{For each View v, View v}': \texttt{R}_\texttt{n}(\texttt{s.H\_cut}[\texttt{v}][\texttt{v}']) = \texttt{cut}[\texttt{v}][\texttt{v}'].$$

We call the resulting mapping $\texttt{R}'()$. We exploit the Proof Extension Theorem of Chapter 3 in order to prove that $\texttt{R}'()$ is a refinement mapping from VS_RFIFO+TS to VSRFIFO : SPEC.

Lemma 7.2.1 *Function* $\texttt{R}'()$ *defined above is a refinement mapping from* VS_RFIFO+TS *to* VSRFIFO : SPEC.

Proof 7.2.1:

Action Correspondence: The action correspondence is the same as that of WV_RFIFO, except for the steps of the type $(\mathtt{s}, \mathtt{view_p}(\mathtt{v}', \mathtt{T}), \mathtt{s}')$ which involve VS_RFIFO+TS delivering views to the application clients. Among these steps, those that are the first to set variable $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$ (when $\mathtt{s}.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'] = \perp$) simulate two steps of VSRFIFO : SPEC: $\mathtt{set\_cut}(\mathtt{v}, \mathtt{v}', \mathtt{s}'.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'])$ followed by $\mathtt{view_p}(\mathtt{v}')$. The rest (when $\mathtt{s}.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'] \neq \perp$) simulate single steps that involve just $\mathtt{view_p}(\mathtt{v}')$.

Simulation Proof:

First, we show that the refinement mapping of WV_RFIFO (presented in Lemma 7.1.1) is still preserved after the modifications introduced by VSRFIFO : SPEC to WV_RFIFO : SPEC. Automaton VSRFIFO : SPEC adds the following preconditions to the $\mathtt{view_p}(\mathtt{v})$ actions of automaton WV_RFIFO : SPEC:

```
cut[current_view[p]][v] ≠ ⊥
(∀ q)  last_dlvrd[q][p] = cut[current_view[p]][v](q)
```

The first precondition holds since $\mathtt{set\_cut}(\mathtt{current\_view}[\mathtt{p}], \mathtt{v}, \mathtt{s}'.\mathtt{H\_cut}[\mathtt{current\_view}[\mathtt{p}]][\mathtt{v}])$ is simulated before $\mathtt{view_p}(\mathtt{v})$. The second one follows immediately from the precondition on VS_RFIFO+TS.$\mathtt{view_p}(\mathtt{v}, \mathtt{T})$, and the extended mapping $\mathtt{R}'()$.

Second, we show that the mapping $\mathtt{R_n}()$ used to extend $\mathtt{R()}$ to $\mathtt{R}'()$ is also a refinement. For those steps $(\mathtt{s}, \mathtt{view_p}(\mathtt{v}', \mathtt{T}), \mathtt{s}')$ that are the first to set variable $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$, the action correspondence implies that the mapping is preserved. For those steps that are not the first to set variable $\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$, the mapping is preserved because $\mathtt{s}'.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}'] = \mathtt{s}.\mathtt{H\_cut}[\mathtt{v}][\mathtt{v}']$, by Corollary 7.2.1. ■

From Lemmas 7.1.1 and 7.2.1 and from Theorem 2.2.1 we conclude the following:

Theorem 7.2.2 VS_RFIFO+TS *implements* VSRFIFO : SPEC *in the sense of trace inclusion.*

7.3 Transitional Set

We now show that VS_RFIFO+TS simulates TS : SPEC. The proofs makes use of *prophecy variables*. A simulation proof that uses prophecy variables implies only finite trace inclusion, but this is sufficient for proving safety properties, (see Chapter 2).

7.3.1 Invariants

Invariant 7.3.1
In every reachable state `s` *of* VS_RFIFO+TS*, for all* `Proc p` *and for all* `StartChangeId id`*, if* `id` $>$ `s[MBRSHP].start_change[p].id`*, then* `s[p].sync_msg[p][id]` $= \perp$*.*

Proof 7.3.1: The proposition is true in the initial state. It remains true for the inductive step involving MBRSHP.$\mathtt{start\_change_p}(\mathtt{id}, \mathtt{set})$ because `s[mbrshp].start_change[p].id` is increased as a result of this action. For the step involving $\mathtt{set\_cut_p}()$, the proposition remains true because $\texttt{s[p].start\_change.id} = \texttt{s[}\textsc{mbrshp}\texttt{].start\_change[p].id}$, as implied by the following invariant, which can be proved by straightforward induction:

In every reachable state $\texttt{s}$ of VS_RFIFO+TS, for all $\texttt{Proc p}$, if $\texttt{s[p].start\_change.id} \neq \perp$, then $\texttt{s[}\textsc{mbrshp}\texttt{].start\_change[p].id} = \texttt{s[p].start\_change.id}$. This invariant holds in the initial state. Critical action $\textsc{mbrshp}.\texttt{start\_change}_\texttt{p}(\texttt{id}, \texttt{set})$ makes it true; Critical action $\texttt{view}_\texttt{p}(\texttt{v}, \texttt{T})$ makes it vacuously true.

Finally, a step involving $\textsc{co\_rfifo}.\texttt{deliver}_{\texttt{q},\texttt{p}}(\langle\text{‘}\textbf{sync\_msg}\text{’}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)$ does not affect the proposition because the case q=p can not happen since end-points do not send synchronization messages to themselves. ■

Lemma 7.3.1 *For any step* $(\texttt{s}, \textsc{mbrshp}.\texttt{start\_change}_\texttt{p}(\texttt{id}, \texttt{set}), \texttt{s}')$ *of* VS_RFIFO+TS*,*

$$\texttt{s[p].sync\_msg[p][start\_change.id]} = \perp.$$

Proof 7.3.1: Follows from the precondition $\texttt{id} > \texttt{s[}\textsc{mbrshp}\texttt{].start\_change[p].id}$ and Invariant 7.3.1. ■

Invariant 7.3.2 *In every reachable state* s *of* VS_RFIFO+TS*, for all* $\texttt{Proc p}$*,*
if $\texttt{s[p].start\_change} \neq \perp$ *and* $\texttt{s[p].sync\_msg[p][s[p].start\_change.id]} \neq \perp$*, then*

$$\texttt{s[p].sync\_msg[p][s[p].start\_change.id].view} = \texttt{s[p].current\_view}.$$

Proof 7.3.2: The proposition is vacuously true in the initial state. For the inductive step, consider the following critical actions:
$\underline{\textsc{mbrshp}.\texttt{start\_change}_\texttt{p}(\texttt{id}, \texttt{set})}$: The proposition remains vacuously true because
$\texttt{s}'[\texttt{p}].\texttt{sync\_msg}[\texttt{p}][\texttt{start\_change.id}] = \texttt{s}[\texttt{p}].\texttt{sync\_msg}[\texttt{p}][\texttt{start\_change.id}] = \perp$ (Lemma 7.3.1).

$\underline{\texttt{set\_cut}_\texttt{p}()}$: Follows immediately from the code.

$\underline{\textsc{co\_rfifo}.\texttt{deliver}_{\texttt{q},\texttt{p}}(\langle\text{‘}\textbf{sync\_msg}\text{’}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)}$: The proposition is unaffected because the case q=p can not happen since end-points do not send synchronization messages to themselves.

$\underline{\texttt{view}_\texttt{p}(\texttt{v})}$: The proposition becomes vacuously true because $\texttt{s}'[\texttt{p}].\texttt{start\_change} = \perp$.

■

7.3.2 Simulation

We augment VS_RFIFO+TS with a prophecy variable `P_legal_views(p)(id)` for each `Proc p`, and each `StartChangeId id`. At the time a start_change `id` is delivered to an end-point `p`, this variable is set to a *predicted* finite set of future views that are allowed to contain `id` as `p`'s start_change id.

```
Prophecy Variable:
For each Proc p, StartChangeId id: SetOf(View) P_legal_views(p)(id), initially
                                                              arbitrary

INTERNAL mbrshp.start_change_p(id, set)  hidden parameter V, a finite set of views
pre: ...
     choose V such that ∀ v ∈ V: (p ∈ v.set) ∧ (v.startId(p) = id)
eff: ...
     P_legal_views(p)(id) ← V

OUTPUT view_p(v, T)
pre: ...
     (∀ q ∈ v.set) v ∈ P_legal_views(q)(v.startId(q))
eff: ...
```

The VS_RFIFO+TS automaton augmented with the prophecy variable has the same traces as those of the original automaton because, it is straightforward to show that the following conditions required for adding a prophecy variable hold:

1. Every state has at least one value for `P_legal_views(p)(id)`.
2. No step is disabled in the *backward direction* by the new preconditions involving `P_legal_views`.
3. Values assigned to state variables do not depend on the values of `P_legal_views`.
4. If s_0 is an initial state of VS_RFIFO+TS, and $\langle s_0, \texttt{P\_legal\_views}\rangle$ is a state of the automaton VS_RFIFO+TS augmented with the prophecy variable, then this state is an initial state.

Invariant 7.3.3
In every reachable state `s` *of* VS_RFIFO+TS*, for all* `Proc p`*, if* `s[p].start_change` $\neq \perp$*, then, for all* `View v` $\in$ `P_legal_views(p)(s[p].start_change.id)`*, it follows that* `p` $\in$ `v.set` *and* `v.startId(p)` $=$ `s[p].start_change.id`*.*

Proof 7.3.3: By induction. The only critical actions are MBRSHP.$\texttt{start\_change}_\texttt{p}(\texttt{id}, \texttt{set})$ and $\texttt{view}_\texttt{p}(\texttt{v}, \texttt{T})$. The proposition is true after the former, and is vacuously true after the latter. ■

Lemma 7.3.2 *The following function* `TS()` *is a* refinement mapping *from automaton* VS_RFIFO+TS *to automaton* TS : SPEC *with respect to their reachable states.*

`TS(s` $\in$ `ReachableStates(`VS_RFIFO+TS`))` $=$ `t` $\in$ `ReachableStates(`TS : SPEC`)`, where

For each Proc p: `t.current_view[p]` $=$ `s[p].current_view`

For each Proc p, View v: `t.prev_view[p][v]` $=$

$$= \begin{cases} \perp & \textit{if } \texttt{v} \notin \texttt{s.P\_legal\_views[p][v.startId(p)]} \\ \texttt{s[p].sync\_msg[p][v.startId(p)].view} & \textit{otherwise} \end{cases}$$

Proof 7.3.2:

Action Correspondence: A step (s, $\texttt{set\_cut}_\texttt{p}$(), s′) of VS_RFIFO+TS simulates a sequence of steps of TS : SPEC. The sequence consists of steps that invlove one $\texttt{set\_prev\_view}_\texttt{p}$(v′) action for each v′ ∈ s.P_legal_views(p)(cid), where cid = s[p].start_change.id. A step (s, $\texttt{view}_\texttt{p}$(v, T), s′) of VS_RFIFO+TS simulates (TS(s), $\texttt{view}_\texttt{p}$(v, T), TS(s′)) of TS : SPEC.

Simulation Proof: Consider the following critical actions:
MBRSHP.$\texttt{start\_change}_\texttt{p}$(id, set): A step involving this action simulates an empty step of TS : SPEC. The simulation holds because s′[p].sync_msg[p][id] = s[p].sync_msg[p][id] = ⊥ (Lemma 7.3.1).
$\texttt{set\_cut}_\texttt{p}$(): simulates a sequence of steps of TS : SPEC that involve one $\texttt{set\_prev\_view}_\texttt{p}$(v′) for each v′ ∈ s.P_legal_views(p)(cid), where cid = s[p].start_change.id. Each such step is enabled as can be seen from the following derivation:

$$\begin{aligned} &\texttt{TS(s).prev\_view[p][v']} = \\ &\quad = \texttt{s[p].sync\_msg[p][v'.startId(p)].view} \text{ (Refinement mapping)} \\ &\quad = \texttt{s[p].sync\_msg[p][cid].view} \text{ (Invariant 7.3.3)} \\ &\quad = \bot. \text{ (Precondition of } \texttt{set\_cut}_\texttt{p}\text{())} \end{aligned}$$

In the post-state, s'[p].sync_msg[p][cid].view and all TS(s').prev_view[p][v'] are equal to s[p].current_view, thus the simulation step holds.
CO_RFIFO.$\texttt{deliver}_\texttt{q,p}$(⟨'**sync_msg**', cid, v, cut⟩): A step involving this action does not affect any of the variables of the refinement mapping and thus simulates an empty step of TS : SPEC. In particular, note that the case of q=p may not happen because end-points do not send synchronization messages to themselves.
$\texttt{view}_\texttt{p}$(v, T): A step involving this action simulates a step of TS : SPEC that involves $\texttt{view}_\texttt{p}$(v, T). The key thing is to show that it is enabled (since it is straightforward to see that, if it is, the refinement is preserved). Action $\texttt{view}_\texttt{p}$(v, T) of TS : SPEC has three preconditions. The fact that they are enabled follows directly from the inductive hypothesis, the code, the refinement mapping, and Invariants 7.3.2 and 7.3.3. ■

From Lemma 7.3.2 and Theorem 2.2.1 we conclude the following:

Theorem 7.3.3 VS_RFIFO+TS *implements* TS : SPEC *in the sense of* finite *trace inclusion.*

7.4 Self Delivery

We now prove that the complete GCS end-point automaton simulates SELF : SPEC. In order to prove this, we need to formalize our assumptions about the behavior of the clients of a GCS end-point: we assume that a client eventually responds to every block request with a block_ok response and subsequently refrains from sending messages until a view is delivered to it. We formalize this requirement by specifying an abstract client automaton in Figure 7.4. In this automaton, each locally controlled action is defined to be a task by itself, which means that it eventually happens if it becomes enabled unless it is subsequently disabled by another action.

```
AUTOMATON CLIENTp : SPEC

Signature:
 Input:   deliverp(q, m), Proc q, AppMsg m         Output:  sendp(m), AppMsg m
          viewp(v), View v                                  block_okp()
          blockp()

State:    block_status ∈ {unblocked, requested, blocked}, initially unblocked

Transitions:
 INPUT  blockp()                                   OUTPUT  sendp(m)
 eff: block_status ← requested                     pre: block_status ≠ blocked
                                                   eff: none
 OUTPUT  block_okp()
 pre: block_status = requested                     INPUT  deliverp(q, m)
 eff: block_status ← blocked                       eff: none

                                                   INPUT  viewp(v)
                                                   eff: block_status ← unblocked
```

Figure 7.1: Abstract specification of a blocking client at end-point p

7.4.1 Invariants

The following invariant states that GCS end-points and their clients have the same perception of what their block_status is.

Invariant 7.4.1 *In every reachable state* s *of* GCS, *for all* Proc p,
$s[\text{GCS}_p].\texttt{block\_status} = s[\texttt{client}_p].\texttt{block\_status}$.

Proof 7.4.1: Trivial induction. ■

Invariant 7.4.2 *In every reachable state* s *of* GCS, *for all* Proc p, *if* s[p].start_change $\neq \perp$ *and*

s[p].block_status $\neq$ blocked, *then* s[p].sync_msg[p][s[p].start_change.id] $= \perp$.

Proof 7.4.2: In the initial state s_0, $s_0[p].\texttt{start\_change} = \perp$; so the proposition is vacuously true. For the inductive step, consider the following critical actions:

$\underline{\text{MBRSHP}.\texttt{start\_change}_p(\texttt{id}, \texttt{set})}$: The proposition remains true because of Lemma 7.3.1.

$\underline{\texttt{block}_p()}$: The proposition is true in the post-state if it is true in the pre-state.

$\underline{\texttt{block\_ok}_p()}$: The proposition becomes vacuously true, as $s'[p].\texttt{block\_status} = \texttt{blocked}$.

$\underline{\texttt{set\_cut}_p()}$: The proposition remains vacuously true because
$s[p].\texttt{block\_status} = s'[p].\texttt{block\_status} = \texttt{blocked}$.

$\underline{\text{CO\_RFIFO}.\texttt{deliver}_{q,p}(\langle\text{'}\textbf{sync\_msg}\text{'}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)}$: The proposition is unaffected because the case q=p can not happen since end-points do not send synchronization messages to themselves.

$\underline{\texttt{view}_p(\texttt{v}, \texttt{T})}$: The proposition becomes vacuously true because $s'[p].\texttt{start\_change} = \perp$.

■

Invariant 7.4.3 *In every reachable state* $\mathtt{s}$ *of* GCS*, for all* $\mathtt{Proc\ p}$*, if* $\mathtt{s[p].start\_change} \neq \perp$ *and*
$\mathtt{s[p].sync\_msg[p][s[p].start\_change.id]} \neq \perp$*, then*
$\mathtt{s[p].sync\_msg[p][s[p].start\_change.id].cut[p]} =$
$=\mathtt{LastIndexOf(s[p].msgs[p][s[p].current\_view])}$.

Proof 7.4.3: In the initial state $\mathtt{s_0}$, $\mathtt{s_0[p].start\_change} = \perp$, so the proposition is vacuously true. For the inductive step, consider the following critical actions:
$\underline{\mathtt{send_p(m)}}$: The proposition is vacuously true because $\mathtt{s'[p].sync\_msg[p][s[p].start\_change.id]} = \perp$, as follows from the precondition $\mathtt{s[client_p].block\_status} \neq \mathtt{blocked}$ on this action at $\mathtt{client_p}$, and from Invariants 7.4.1 and 7.4.2.

$\underline{\text{MBRSHP}.\mathtt{start\_change_p(id, set)}}$: The proposition is vacuously true because $\mathtt{s'[p].sync\_msg[p][id]} = \mathtt{s[p].sync\_msg[p][id]}$, which by Lemma 7.3.1 is $\perp$.

$\underline{\mathtt{set\_cut_p()}}$: Follows from $\mathtt{p} \in \mathtt{current\_view.set}$ (Invariant 7.1.1) and the precondition $(\forall \mathtt{q} \in \mathtt{current\_view.set})\ \mathtt{cut(q)} = \mathtt{LongestPrefixOf(msgs[q][v])}$.

$\underline{\text{CO\_RFIFO}.\mathtt{deliver_{q,p}}(\langle\text{‘}\mathtt{sync\_msg}\text{’}, \mathtt{cid}, \mathtt{v}, \mathtt{cut}\rangle)}$: The proposition is unaffected because the case q=p can not happen since, as can be proved by straightforward induction, end-points do not send synchronization messages to themselves.

$\underline{\mathtt{view_p(v, T)}}$: The proposition becomes vacuously true because $\mathtt{s'[p].start\_change} = \perp$.
■

7.4.2 Simulation

Lemma 7.2.1 in Section 7.2 on page 90 establishes function $\mathtt{R'}()$ as a refinement mapping from automaton VS_RFIFO+TS to automaton VSRFIFO : SPEC. We now argue that $\mathtt{R'}()$ is also a refinement mapping from automaton GCS to automaton SELF : SPEC.

Lemma 7.4.1 *Refinement mapping* $\mathtt{R'}()$ *from* VS_RFIFO+TS *to* VSRFIFO : SPEC *(given in Lemma 7.2.1) is also a refinement mapping from automaton* GCS *to automaton* SELF : SPEC*, under the assumption that clients at each end-point* $\mathtt{p}$ *satisfy the* $\text{CLIENT}_\mathtt{p}$: SPEC *specification for blocking clients.*

Proof : Automaton SELF : SPEC modifies automaton WV_RFIFO : SPEC by adding a precondition, $\mathtt{last\_dlvrd[p][p] = LastIndexOf(msgs[p][current\_view[p]])}$, to the steps involving $\mathtt{view_p}()$ actions. We have to show that this precondition is enabled when a step of GCS involving $\mathtt{view_p(v, T)}$ attempts to simulate a step of SELF : SPEC involving $\mathtt{view_p(v)}$. Indeed:

$$\begin{aligned}
\mathtt{s[p].last\_dlvrd[p]} &= \mathtt{max_{r\in T} sync\_msg[r][v.startId(r)].cut[p]} \text{ (a precondition)}\\
&= \mathtt{s[p].sync\_msg[p][v.startId(p)].cut[p]} \text{ (Invariant 7.2.1.)}\\
&= \mathtt{s[p].sync\_msg[p][s[p].start\_change.id].cut[p]} \text{ (a precondition)}\\
&= \mathtt{LastIndexOf(s[p].msgs[p][s[p].current\_view])} \text{ (Invariant 7.4.3).}
\end{aligned}$$

Thus, $\mathtt{R}'(\mathtt{s}).\mathtt{last\_dlvrd}[\mathtt{p}][\mathtt{p}] = \mathtt{LastIndexOf}(\mathtt{R}'(\mathtt{s}).\mathtt{msgs}[\mathtt{p}][\mathtt{R}'(\mathtt{s}).\mathtt{current\_view}[\mathtt{p}]])$ and the precondition is satisfied. ■

From Lemmas 7.1.1, 7.2.1, and 7.4.1 and Theorem 2.2.1 we conclude the following:

Theorem 7.4.2 *Automaton* GCS *implements automaton* SELF : SPEC *in the sense of trace inclusion, under the assumption that clients at each end-point* $\mathtt{p}$ *satisfy the* $\text{CLIENT}_{\mathtt{p}}$: SPEC *specification for blocking clients.*

As a child of VS_RFIFO+TS, GCS also satisfies all the safety property that VS_RFIFO+TS does, in particular TS : SPEC. Thus, from Theorems 7.3.3, and 7.4.2 we conclude the following:

Theorem 7.4.3 *Automaton* GCS *implements each of the* WV_RFIFO : SPEC*,* VSRFIFO : SPEC*,* TS : SPEC*, and* SELF : SPEC *automata in the sense of trace inclusion, under the assumption that clients at each end-point* $\mathtt{p}$ *satisfy the* $\text{CLIENT}_{\mathtt{p}}$: SPEC *specification for blocking clients.*

Chapter 8

Correctness Proof: Liveness Property

In this chapter we prove that fair executions of our group communication service GCS satisfy Liveness property 5.2.2 of Section 5.2. In order to show that a certain action eventually happens, we argue that the preconditions on this action eventually become and stay satisfied, and thus the action eventually occurs, by fairness of the execution. Subsection 8.1 below presents a number of invariant that are used in the proof of Liveness property 5.2.2 in Subsection 8.2.

8.1 Invariants

The following invariant captures the fact that, before an end-point computes who the members of its transitional set are, it does not deliver to its client application messages other than those committed by its own synchronization message. Afterwards, the end-point delivers only the messages committed to delivery by the members of the transitional set.

Invariant 8.1.1 *In every reachable state* `s` *of* GCS*, for all* `Proc p`*, if* `s[p].start_change` $\neq \perp$ *and* `s[p].sync_msg[p][s[p].start_change.id]` $\neq \perp$*, then for all* `Proc q` $\in$ `s[p].current_view.set`*,*

1. *If* `s[p].start_change.id` $\neq$ `s[p].mbrshp_view.startId(p)`*, then* `s[p].last_dlvrd[q]` $\leq$ `s[p].sync_msg[p][s[p].start_change.id].cut[q]`.
2. *Otherwise, let* `v` = `s[p].current_view`*,* `v'` = `s[p].mbrshp_view`*, and let* `T` = {`q` $\in$ `v'.set` $\cap$ `v.set` | `sync_msg[q][v'.startId(q)].view` = `v` }*, then* `s[p].last_dlvrd[q]` $\leq max_{r \in T}$ `s[p].sync_msg[r][v'.startId(r)].cut[q]`.

Proof 8.1.1: The proposition is true in the initial state s_0, since s_0[p].start_change = $\perp$. For the inductive step, consider the following critical actions:

$\underline{\texttt{deliver}_\texttt{p}(\texttt{q},\texttt{m})}$: The proposition remains true because the precondition on this action mimics the statement of this proposition.

$\underline{\text{MBRSHP}.\texttt{start\_change}_\texttt{p}(\texttt{id},\texttt{set})}$: The proposition is vacuously true because $\texttt{s}'[\texttt{p}].\texttt{sync\_msg}[\texttt{p}][\texttt{id}] = \texttt{s}[\texttt{p}].\texttt{sync\_msg}[\texttt{p}][\texttt{id}]$, which by Lemma 7.3.1 is equal to $\perp$.

$\underline{\text{MBRSHP}.\texttt{view}_\texttt{p}(\texttt{v})}$: In the post-state, $\texttt{s}[\texttt{p}].\texttt{start\_change.id} = \texttt{s}[\texttt{p}].\texttt{mbrshp\_view.startId}(\texttt{p})$, so we must consider the second proposition. Its truth follows from the inductive hypothesis and the fact that $\texttt{p} \in \texttt{T}$, as implied by Invariant 7.1.1.

$\underline{\texttt{set\_cut}_\texttt{p}()}$: The proposition holds since index $\texttt{s}[\texttt{p}].\texttt{last\_dlvrd}[\texttt{q}]$ is bounded by $\texttt{LongestPrefixOf}(\texttt{s}[\texttt{p}].\texttt{msgs}[\texttt{q}][\texttt{s}[\texttt{p}].\texttt{current\_view}])$ in every reachable state of the system for any $\texttt{Proc q} \in \texttt{s}[\texttt{p}].\texttt{current\_view.set}$ (this fact can be straightforwardly proved by induction), and from the precondition, $(\forall \texttt{q} \in \texttt{s}[\texttt{p}].\texttt{current\_view.set})$ $\texttt{cut}(\texttt{q}) = \texttt{LongestPrefixOf}(\texttt{s}[\texttt{p}].\texttt{msgs}[\texttt{q}][\texttt{s}[\texttt{p}].\texttt{current\_view}])$.

$\underline{\text{CO\_RFIFO}.\texttt{deliver}_{\texttt{q},\texttt{p}}(\langle\text{‘}\textbf{sync\_msg}\text{’}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)}$: The proposition is unaffected because the case $\texttt{q} = \texttt{p}$ is impossible since end-points do not send cuts to themselves.

$\underline{\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})}$: The proposition becomes vacuously true because $\texttt{s}'[\texttt{p}].\texttt{start\_change} = \perp$.

■

The following Invariant states that if an end-point p has end-point q's cut committing certain messages sent by end-point r in view v, then end-point q has those messages buffered.

Invariant 8.1.2 *In every reachable state* $\texttt{s}$ *of* GCS*, for all* $\texttt{Proc p}$*,* $\texttt{Proc q}$*,* $\texttt{Proc r}$*, and* $\texttt{StartChangeId cid}$*, if* $\texttt{s}[\texttt{p}].\texttt{sync\_msg}[\texttt{q}][\texttt{cid}] \neq \perp$*, then, for every integer* $\texttt{i}$ *between* 1 *and* $\texttt{s}[\texttt{p}].\texttt{sync\_msg}[\texttt{q}][\texttt{cid}].\texttt{cut}[\texttt{r}]$*,* $\texttt{s}[\texttt{q}].\texttt{msgs}[\texttt{r}][\texttt{s}[\texttt{p}].\texttt{sync\_msg}[\texttt{q}][\texttt{cid}].\texttt{view}][\texttt{i}] \neq \perp$*.*

Proof 8.1.2: The truth of the invariant follows from Invariant 7.2.1 if we can prove that an end-point's cut commits the end-point to deliver only those messages that it already has on its msgs queue. Formally, this proposition means that, in every reachable state s of GCS, for all $\texttt{Proc q}$, if $\texttt{s}[\texttt{q}].\texttt{start\_change} \neq \perp$ and $\texttt{s}[\texttt{q}].\texttt{sync\_msg}[\texttt{q}][\texttt{s}[\texttt{q}].\texttt{start\_change.id}] \neq \perp$, then, for all $\texttt{Proc r}$ and all $\texttt{Int i}$ such that $1 \leq \texttt{i} \leq \texttt{s}[\texttt{q}].\texttt{sync\_msg}[\texttt{q}][\texttt{s}[\texttt{q}].\texttt{start\_change.id}].\texttt{cut}[\texttt{r}]$, $\texttt{s}[\texttt{q}].\texttt{msgs}[\texttt{r}][\texttt{s}[\texttt{q}].\texttt{current\_view}][\texttt{i}] \neq \perp$. This proposition can be straightforwardly proved by induction: The only interesting action is $\texttt{set\_cut}_\texttt{q}()$. The truth of the proposition after this action is taken follows immediately from the precondition: $(\forall \texttt{r} \in \texttt{s}[\texttt{q}].\texttt{current\_view.set})$ $\texttt{cut}(\texttt{r}) = \texttt{LongestPrefixOf}(\texttt{s}[\texttt{q}].\texttt{msgs}[\texttt{r}][\texttt{s}[\texttt{q}].\texttt{current\_view}])$. ■

Invariant 8.1.3 *In every reachable state* $\texttt{s}$ *of* GCS*, for all* $\texttt{Proc p}$ *and* $\texttt{Proc q}$*, if* $\texttt{q} \in \texttt{s}[\texttt{p}].\texttt{sync\_set}$ *then (a)* $\texttt{q} \in \texttt{s}[\texttt{p}].\texttt{start\_change.set}$ *and (b)* $\texttt{q} \in \texttt{s}[\texttt{p}].\texttt{reliable\_set}$*.*

Proof 8.1.3: The proposition is vacuously true in the initial state, where $\texttt{s}[\texttt{p}].\texttt{sync\_set}$ is empty. The inductive steps for the critical actions $\text{MBRSHP}.\texttt{start\_change}_\texttt{p}(\texttt{id},\texttt{set})$, $\text{GCS}.\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$, and $\text{CO\_RFIFO}.\texttt{send}_\texttt{p}(\texttt{set}, \langle\text{‘}\textbf{sync\_msg}\text{’}, \texttt{cid}, \texttt{v}, \texttt{cut}\rangle)$ follow immediately from their code in Figure 6.5. The inductive step for $\text{CO\_RFIFO}.\texttt{reliable\_set}_\texttt{p}(\texttt{set})$ follows straightforwardly from the precondition-effect code in Figures 6.2 and 6.5. The inductive step for the critical action GCS.$\texttt{set\_cut}_\texttt{p}()$ follows from the code, which sets $\texttt{sync\_set}$ to $\{\texttt{p}\}$, and from the fact that $\texttt{p}$ is always in its own $\texttt{reliable\_set}$ and $\texttt{start\_change.set}$ (provided $\texttt{start\_change} \neq \perp$), which can be straightforwardly proved by induction. ■

8.2 Liveness Proof

The following lemma states that, in any execution of GCS, every GCS.$\texttt{view}_\texttt{p}$ event is preceded by the right MBRSHP.$\texttt{view}_\texttt{p}$ event, which itself is preceded by the right MBRSHP.$\texttt{start\_change}_\texttt{p}$ event.

Lemma 8.2.1 *In every execution sequence α of* GCS*, the following are true:*

1. *For every* GCS.$\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$ *event, there is a preceding* MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ *event. Moreover, neither a* MBRSHP.$\texttt{start\_change}_\texttt{p}$ *nor a* MBRSHP.$\texttt{view}_\texttt{p}$ *event occurs between* MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ *and* GCS.$\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$*.*

2. *For every* MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ *event, there is a preceding* MBRSHP.$\texttt{start\_change}_\texttt{p}(\texttt{id},\texttt{set})$ *event with* $\texttt{id} = \texttt{v.startId(p)}$ *and* $\texttt{set} \supseteq \texttt{v.set}$*, such that neither a* MBRSHP.$\texttt{start\_change}_\texttt{p}$*, nor a* MBRSHP.$\texttt{view}_\texttt{p}$*, nor a* GCS.$\texttt{view}_\texttt{p}$ *event occurs in α between* MBRSHP.$\texttt{start\_change}_\texttt{p}(\texttt{id},\texttt{set})$ *and* MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$*.*

Proof 8.2.1:

1. Assume that GCS.$\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$ occurs in α. Two of the preconditions on GCS.$\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$ are $\texttt{v} = \texttt{p.mbrshp\_view}$ and $\texttt{v.startId(p)} = \texttt{p.start\_change.id}$, which can only become satisfied as a result of a preceding MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ event, followed by no MBRSHP.$\texttt{start\_change}_\texttt{p}$ and MBRSHP.$\texttt{view}_\texttt{p}$ events.

2. Assume that MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ occurs in α. Then a MBRSHP.$\texttt{start\_change}_\texttt{p}(\texttt{id},\texttt{set})$ event with $\texttt{id} = \texttt{v.startId(p)}$ and $\texttt{set} \supseteq \texttt{v.set}$ must precede MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ because, by the MBRSHP specification, it is the only possible event that can cause the preconditions for MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$ to become true, and because these preconditions do not hold in the initial state of MBRSHP.

   There maybe several MBRSHP.$\texttt{start\_change}_\texttt{p}(\texttt{id},\texttt{set})$ events with the same $\texttt{id}$ and different $\texttt{set}$ arguments. After the last such event, an occurrence of a different MBRSHP.$\texttt{start\_change}_\texttt{p}$ event or a MBRSHP.$\texttt{view}_\texttt{p}$ event would violate one of the preconditions of MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$; thus, such events may not happen. As a corollary from this and part 1 of this Lemma, a GCS.$\texttt{view}_\texttt{p}(\texttt{v}',\texttt{T}')$ event cannot occur between the last MBRSHP.$\texttt{start\_change}_\texttt{p}(\texttt{id},\texttt{set})$ and MBRSHP.$\texttt{view}_\texttt{p}(\texttt{v})$.

■

Lemma 8.2.2 (Liveness) *Let α be a fair execution of a group communication service* GCS *in which view* v *becomes eventually stable as defined by Property 5.2.1. Then at each endpoint* $\texttt{p} \in \texttt{v.set}$*,* GCS.$\texttt{view}_\texttt{p}(\texttt{v},\texttt{T})$*, with some* T*, eventually occurs. Furthermore, for every*

GCS.$\mathtt{send_p(m)}$ *that occurs after* GCS.$\mathtt{view_p(v,T)}$ *and for every* $\mathtt{q} \in \mathtt{v.set}$, GCS.$\mathtt{deliver_q(p,m)}$ *also occurs.*

Proof 8.2.2:
Part I We first prove that GCS.$\mathtt{view_p(v,T)}$ eventually occurs. Our task is to show that, for each $\mathtt{p} \in \mathtt{v.set}$ and some transitional set $\mathtt{T}$, action GCS.$\mathtt{view_p(v,T)}$ becomes enabled at some point after $\mathtt{p}$ receives MBRSHP.$\mathtt{view_p(v)}$ and that it stays enabled forever thereafter unless it is executed. The fact that α is a fair execution of GCS then implies that GCS.$\mathtt{view_p(v,T)}$ is in fact executed.

In order for GCS.$\mathtt{view_p(v,T)}$ to become enabled, its preconditions (see Figures 6.2 and 6.5) must eventually become and stay satisfied until GCS.$\mathtt{view_p(v,T)}$ is executed. We now consider each of these preconditions:
$\underline{\mathtt{v = p.mbrshp\_view \neq current\_view}}$: This precondition ensures that view $\mathtt{v}$ that is attempted to be delivered to the client at $\mathtt{p}$ is the latest view produced by MBRSHP and has not yet been delivered to the client. The precondition becomes satisfied as a result of MBRSHP.$\mathtt{view_p(v)}$. Since in any reachable state of the system MBRSHP.$\mathtt{mbrshp\_view} = \mathtt{p.mbrshp\_view} \geq \mathtt{p.current\_view}$ (Local Monotonicity), this precondition remains satisfied forever, unless GCS.$\mathtt{view_p(v,T)}$ is executed. This is because, by our assumption, α does not contain any subsequent MBRSHP.$\mathtt{view_p(v')}$, and hence, by contrapositive of part 1 of Lemma 8.2.1, it also does not contain any subsequent GCS.$\mathtt{view_p(v',T')}$ with $\mathtt{v'} \neq \mathtt{v}$.

$\underline{\mathtt{v.startId(p) = p.start\_change.id}}$: This precondition prevents delivery of obsolete views: it ensures that the MBRSHP service has not issued a new $\mathtt{start\_change}$ notification since the time it produced view $\mathtt{v}$. If this condition is not already satisfied before the last MBRSHP.$\mathtt{start\_change_p(id,set)}$ event with $\mathtt{id} = \mathtt{v.startId(p)}$ and $\mathtt{set} \supseteq \mathtt{v.set}$, then it becomes satisfied as a result of this event, which, by part 2 of Lemma 8.2.1, must precede MBRSHP.$\mathtt{view_p(v)}$ in α.

This condition stays satisfied from the time of the last MBRSHP.$\mathtt{start\_change_p(id,set)}$ at least until GCS.$\mathtt{view_p(v,T)}$ occurs because the only two types of actions,
MBRSHP.$\mathtt{start\_change_p(id',set')}$ and GCS.$\mathtt{view_p(v',T')}$ with $\mathtt{v'} \neq \mathtt{v}$, that may affect the value of $\mathtt{p.start\_change}$ cannot occur in α after MBRSHP.$\mathtt{start\_change_p(id,set)}$, as implied by the assumption on this lemma and Lemma 8.2.1.

$\underline{\mathtt{v.set - sync\_set} = \emptyset}$: This precondition ensures that prior to delivering view $\mathtt{v}$, end-point $\mathtt{p}$ sends out its synchronization message to every member of $\mathtt{v}$.

Notice that if this precondition becomes satisfied any time after the occurence of the last MBRSHP.$\mathtt{start\_change_p(id,set)}$ event with $\mathtt{id} = \mathtt{v.startId(p)}$ and $\mathtt{set} \supseteq \mathtt{v.set}$, then it stays satisfied from then on until GCS.$\mathtt{view_p(v,T)}$ is executed. If the precondition is not already satisfied right after the MBRSHP.$\mathtt{start\_change_p}$ action, it becomes satisfied as a result of CO_RFIFO.$\mathtt{send_p(set, \langle}$'**sync_msg**'$\mathtt{, v.startId(p), v, cut\rangle)}$ with $\mathtt{set} = \mathtt{p.start\_change.set} - \mathtt{p.sync\_set}$. This CO_RFIFO.$\mathtt{send_p}$ action must eventually occurs in α because its two preconditions, $(\mathtt{p.sync\_msg[p][id]} \neq \perp)$ and $(\mathtt{set} \subseteq \mathtt{reliable\_set})$, eventually become satisfied, for the following reasons:

- If the first precondition holds any time after the last MBRSHP.$\mathtt{start\_change_p(id,set)}$ event with $\mathtt{id} = \mathtt{v.startId(p)}$ and $\mathtt{set} \supseteq \mathtt{v.set}$ occurs, then it stays satisfied from that point on. If it is not already satisfied right after the MBRSHP.$\mathtt{start\_change_p}$

action, it becomes satisfied as a result of $\texttt{set\_cut}_\texttt{p}()$. In order for $\texttt{set\_cut}_\texttt{p}()$ to occur, its precondition, $\texttt{block\_status} = \texttt{blocked}$, has to becomes satisfied (see Figure 6.6). This occurs as a result of a $\texttt{block\_ok}_\texttt{q}()$ input from the client at $\texttt{q}$. If $\texttt{block\_status}$ equals $\texttt{blocked}$ at anytime after MBRSHP.$\texttt{start\_change}_\texttt{q}(\texttt{v.startId(q)}, \texttt{set})$, then it remains such until GCS.$\texttt{view}_\texttt{q}(\texttt{v})$ happens because $\texttt{block}_\texttt{q}()$ is not enabled after that, and because GCS.$\texttt{view}_\texttt{q}(\texttt{v})$ is the only possible GCS view event (by the contrapositive of part 2 of Lemma 8.2.1). To see that $\texttt{block\_status}$ does in fact become $\texttt{blocked}$ consider the three possible values of $\texttt{block\_status}$ right after MBRSHP.$\texttt{start\_change}_\texttt{q}(\texttt{v.startId(q)}, \texttt{set})$ occurs:

1. $\texttt{block\_status} = \texttt{blocked}$: We are done.
2. $\texttt{block\_status} = \texttt{requested}$: By Invariant 7.4.1, $\texttt{client.block\_ok}_\texttt{q}()$ is enabled. It stays enabled until it is executed because the actions, $\texttt{block}_\texttt{q}()$ and GCS.$\texttt{view}_\texttt{q}()$, which would disable it, cannot occur. When it is executed, the precondition becomes satisfied.
3. $\texttt{block\_status} = \texttt{unblocked}$: When MBRSHP.$\texttt{start\_change}_\texttt{q}(\texttt{v.startId(q)}, \texttt{set})$ occurs, $\texttt{block}_\texttt{q}()$ becomes and stays enabled until it is executed. After that, the value of $\texttt{block\_status}$ becomes $\texttt{requested}$ and the same reasoning as in the previous case applies.

- The second precondition, $\texttt{set} \subseteq \texttt{reliable\_set}$, becomes satisfied as a result of action CO_RFIFO.$\texttt{reliable}_\texttt{q}(\texttt{set})$ with $\texttt{set} = \texttt{current\_view.set} \cup \texttt{start\_change.set}$. This action becomes enabled when $\texttt{q}$ receives MBRSHP.$\texttt{start\_change}_\texttt{q}(\texttt{v.startId(q)}, \texttt{set})$, and therefore it eventually occurs. Afterwards, $\texttt{reliable\_set}$ remains unchanged because CO_RFIFO.$\texttt{reliable}_\texttt{q}(\texttt{set})$ remains disabled; this is because of the precondition $\texttt{reliable\_set} \neq \texttt{set}$ and the fact that $\texttt{q}$'s $\texttt{current\_view}$ and $\texttt{start\_change}$ remain unchanged.

  When CO_RFIFO.$\texttt{send}_\texttt{p}(\texttt{set}, \langle$'**sync_msg**'$, \texttt{v.startId(p)}, \texttt{v}, \texttt{cut}\rangle)$ occurs, $\texttt{p.sync\_set}$ is set to $\texttt{p.start\_change.set}$. Since $\texttt{v.set}$ is a subset of $\texttt{p.start\_change.set}$, this implies that $\texttt{v.set} - \texttt{p.sync\_set}$ eventually becomes and stays $\emptyset$.

$\underline{(\forall \texttt{q} \in \texttt{v.set} \cap \texttt{p.current\_view.set})\ \texttt{p.sync\_msg[q][v.startId(q)]} \neq \perp}$: This precondition ensures that $\texttt{p}$ has received the right synchronization message from every $\texttt{q}$ in $\texttt{v.set} \cap \texttt{p.current\_view.set}$. The argument above implies that $\texttt{q}$ eventually sends to $\texttt{p}$ a synchronization message tagged with $\texttt{v.startId(q)}$ and, at the same time, adds $\texttt{p}$ to $\texttt{q.sync\_set}$, where $\texttt{p}$ remains forever, unless GCS.$\texttt{view}_\texttt{p}(\texttt{v}, \texttt{T})$ with some $\texttt{T}$ occurs. In order to conclude that CO_RFIFO eventually delivers this synchronization message to $\texttt{p}$, we argue that, from the time the last synchronization message from $\texttt{q}$ to $\texttt{p}$ is placed on CO_RFIFO.$\texttt{channel[q][p]}$ and at least until it is delivered to $\texttt{p}$, end-point $\texttt{p}$ is in both CO_RFIFO.$\texttt{reliable\_set[q]}$ and CO_RFIFO.$\texttt{live\_set[q]}$. The former implies that CO_RFIFO does not lose any messages (in particular, this synchronization message) from $\texttt{q}$ to $\texttt{p}$. In conjunction with α being a fair execution, the latter implies that CO_RFIFO eventually delivers every message (in particular, this synchronization message) on the channel from $\texttt{q}$ to $\texttt{p}$.

- From the time $\texttt{q}$ sends to $\texttt{p}$ the last synchronization message tagged with $\texttt{v.startId(q)}$ until GCS.$\texttt{view}_\texttt{q}(\texttt{v}, \texttt{T})$ occurs, $\texttt{p}$ is included in $\texttt{q.sync\_set}$. Invariant 8.1.3 implies that in that period $\texttt{p}$ is included in CO_RFIFO.$\texttt{reliable\_set[q]}$. After GCS.$\texttt{view}_\texttt{q}(\texttt{v}, \texttt{T})$ occurs, $\texttt{p}$ is still included in CO_RFIFO.$\texttt{reliable\_set[q]}$, since $\texttt{p} \in \texttt{v.set}$.
- End-point $\texttt{p}$ becomes a member of CO_RFIFO.$\texttt{live\_set[q]}$ at the time of MBRSHP.$\texttt{view}_\texttt{q}(\texttt{v})$, because MBRSHP.$\texttt{view}_\texttt{q}(\texttt{v})$ is linked to CO_RFIFO.$\texttt{live\_set}_\texttt{q}(\texttt{v.set})$ and because $\texttt{p} \in$

v.set. This property remains true afterward because α does not contain any subsequent MBRSHP events at end-point q.

Thus, end-point p eventually receives the right synchronization messages from every q in v.set $\cap$ p.current_view.set.

<u>last_sent $\geq$ sync_msg[p][v.startId(p)].cut(p):</u> This precondition ensures that before delivering view v, p sends to others all of its own messages indicated in its own cut. This precondition eventually becomes satisfied because sending of of application messages via CO_RFIFO.send$_p$, which increments p.last_sent, is enabled at least until p.last_sent reaches sync_msg[p][v.startId(p)].cut(p), as implied by Invariant 8.1.2.

<u>($\forall$q $\in$ current_view.set) p.last_dlvrd[q] = max$_{r \in T}$p.sync_msg[r][v.startId(r)].cut[q]:</u> This precondition verifies that p has delivered to its client exactly the application messages that it needs to deliver in order for Virtually-Synchronous Delivery to be satisfied. By Invariant 8.1.1, p.last_dlvrd[q] never exceeds max$_{r \in T}$ {p.sync_msg[r][v.startId(r)].cut[q]} for any q. It is left to show that p.last_dlvrd[q] does not remain smaller than max$_{r \in T}$.

We have shown above that all the other preconditions for delivering view v by p eventually become and remain satisfied until the view is delivered. Consider the part of α after all of these preconditions hold. Let q be an end-point in current_view.set such that p.last_dlvrd[q] < max$_{r \in T}$p.sync_msg[r][v.startId(r)].cut[q]. Let i = p.last_dlvrd[q] + 1. We now argue that p.last_dlvrd[q] eventually becomes i, that is, that p eventually delivers the next message from q. Applying this argument inductively, implies that p.last_dlvrd[q] eventually reaches max$_{r \in T}$ {p.sync_msg[r][v.startId(r)].cut[q]}.

All the preconditions (except perhaps p.msgs[q][p.current_view][i] $\neq \perp$) for delivering the i'th message from q are eventually satisfied because they are the same as the preconditions for p delivering view v, which we have shown to be satisfied. Thus, if the i'th message is already on p.msgs[q][p.current_view][i], then delivery of this message eventually occurs by fairness, resulting in p.last_dlvrd[q] being incremented; in this case, we are done.

Therefore, consider the case when p lacks the i'th message, m, from q. There are two possibilities:

1. If end-point q is in p's transitional set T for view v, then we know the following:
   - q's view prior to installing view v is the same as p's current view (by definition of T and Invariant 7.3.2).
   - q's reliable_set contains p starting before q sent any messages in that view and continuing for the rest of α.
   - Invariant 8.1.2 implies that q has this message and all the messages that precede it in
     q.msgs[q][p.current_view].
   - End-point q is enabled to send these messages to p in FIFO order. The only event that could prevent q from sending these messages is gcs.view$_q$(v), as it would change the value of q.current_view. However, as we argued above, q must send all of the messages it committed in its cut before delivering view gcs.view$_q$(v). Self Delivery (Invariant 7.4.3) implies that q's cut includes all of the messages q sent while in v. Thus, q would eventually send m to p.
   - The fact that the connection between q and p is live at least after MBRSHP.view$_q$(v) occurs implies that CO_RFIFO eventually delivers this message to p.

2. Otherwise, if end-point q is not in p's transitional set T for view v, we know by the fact that i is $\leq \max_{r \in T}$ {p.sync_msg[r][v.startId(r)].cut[q]}, that there exist some end-points in T whose synchronization messages commit to deliver the i'th message from q in view p.current_view. Let r be an end-point with a smallest identifier among these end-points. Here is what we know:
   - Invariant 8.1.2 implies that r has this message on its r.msgs[r][p.current_view] queue.
   - r's reliable_set contains p starting before r sent any messages in that view and continuing for the rest of α.
   - Upon examination of each of the ForwardingStrategyPredicate s in Section 6.2, we see that the preconditions for r forwarding the i'th message of q to a set including p eventually become and stay satisfied.
   - Since in both forwarding strategies there is only a finite number of messages from q sent in this view that can be forwarded, fairness implies that the i's message is eventually forwarded to p.
   - The fact that the connection between r and p is live at least after MBRSHP.view_q(v) occurs implies that CO_RFIFO eventually delivers this message to p.

Therefore, the i'th message from q is eventually delivered to end-point p, and since, as a result of this, the preconditions on delivering this message to the client at p are satisfied, this delivery eventually occurs, and p.last_dlvrd[q] is incremented. By applying this argument inductively, we conclude that p.last_dlvrd[q] eventually reaches
$\max_{r \in T}$ p.sync_msg[r][v.startId(r)].cut[q] for every q in current_view.set.

We have shown that each precondition on p delivering GCS.$\text{view}_p(v, T)$ eventually becomes and stays satisfied. Fairness implies that GCS.$\text{view}_p(v, T)$ eventually occurs.

Part II We now consider the second part of the lemma. The following argument proves that, after GCS.$\text{view}_p(v, T)$ occurs at p, for every subsequent GCS.$\text{send}_p(m)$ event at p, there is a corresponding GCS.$\text{deliver}_q(p, m)$ event that occurs at every $q \in$ v.set:

1. For the rest of α, after GCS.$\text{view}_p(v, T)$ occurs, CO_RFIFO.live_set[p] is equal to v.set.

   This is true because CO_RFIFO.live_set[p] is set to v.set when MBRSHP.$\text{view}_p(v)$ occurs and remains unchanged thereafter because of the assumption that α does not contain any subsequent MBRSHP events at end-point p.

2. After GCS.$\text{view}_p(v, T)$ occurs and before any CO_RFIFO.send_p event involving a ViewMsg or an AppMsg occurs, p eventually executes CO_RFIFO.reliable_p(v.set). Moreover, after that and forever thereafter, both p.reliable_set and CO_RFIFO.reliable_set[p] equal v.set.

   This is true because GCS.$\text{view}_p(v, T)$ sets p.start_change to $\perp$ and p.current_view.set to v.set, thus enabling CO_RFIFO.reliable_p(v.set). This action eventually happens because α is a fair execution and because for the rest of α there are no subsequent MBRSHP.start_change_p and GCS.$\text{view}_p(v', T')$ events. Because of the latter reason, p.start_change and p.current_view.set remain unchanged. Therefore, CO_RFIFO.$\texttt{reliable}_\texttt{p}$ remains disabled and both variables CO_RFIFO.$\texttt{reliable\_set}[\texttt{p}]$ and $\texttt{p.reliable\_set}$ remain equal to $\texttt{v.set}$.

From the above argument and from fairness, it follows that any kind of message that end-point $\texttt{p}$ sends subsequently to $\texttt{q}$ via CO_RFIFO will eventually reach end-point $\texttt{q}$.

3. Action CO_RFIFO.$\texttt{send}_\texttt{p}(\texttt{v.set} - \{\texttt{p}\}, \langle\text{‘}\textbf{view\_msg}\text{’}, \texttt{v}\rangle)$ eventually occurs after action CO_RFIFO.$\texttt{reliable}_\texttt{p}(\texttt{v.set})$ occurs, as follows from the code in Figure 6.5. By the reasoning above, CO_RFIFO delivers this $\texttt{ViewMsg}$ to every end-point $\texttt{q} \in \texttt{v.set} - \{\texttt{p}\}$, resulting in $\texttt{q.view\_msg}[\texttt{p}]$ being set to $\texttt{v}$ for the remainder of α (Invariant 7.1.4).

4. When GCS.$\texttt{send}_\texttt{p}(\texttt{m})$ event occurs at $\texttt{p}$, $\texttt{m}$ is appended to $\texttt{p.msgs}[\texttt{p}][\texttt{v}]$.

5. After sending the $\texttt{ViewMsg}$, for the rest of α, if $\texttt{p.msgs}[\texttt{p}][\texttt{v}][\texttt{p.last\_sent} + 1]$ contains a message (say $\texttt{m}'$), action CO_RFIFO.$\texttt{send}_\texttt{p}(\texttt{v.set} - \{\texttt{p}\}, \langle\text{‘}\textbf{app\_msg}\text{’}, \texttt{m}'\rangle)$ is enabled, and hence eventually occurs by fairness. Since $\texttt{p.last\_sent}$ is incremented after each application message is sent using CO_RFIFO.$\texttt{send}_\texttt{p}$, any message on $\texttt{p.msgs}[\texttt{p}][\texttt{v}]$ is eventually sent to $\texttt{v.set} - \{\texttt{p}\}$. As was argued above, these messages are eventually delivered to every end-point $\texttt{q} \in \texttt{v.set} - \{\texttt{p}\}$. Since $\texttt{q.view\_msg}[\texttt{p}] = \texttt{v}$ at the time $\texttt{q}$ receives $\texttt{m}'$, $\texttt{q}$ puts $\texttt{m}'$ in $\texttt{q.msgs}[\texttt{p}][\texttt{v}][\texttt{q.last\_rcvd} + 1]$ (Invariant 7.1.5) and increments $\texttt{q.last\_rcvd}$. Therefore, all messages that end-point $\texttt{p}$ sends in view $\texttt{v}$ are eventually inserted with no gaps in the end-point $\texttt{q}$'s queue, $\texttt{q.msgs}[\texttt{p}][\texttt{v}]$, for every $\texttt{q} \in \texttt{v.set} - \{\texttt{p}\}$.

6. Once GCS.$\texttt{view}_\texttt{q}(\texttt{v}, \texttt{T})$ happens (by Part I of the lemma), end-point $\texttt{q} \in \texttt{v.set}$ is continuously enabled to deliver a message, $\texttt{m}'$, from $\texttt{q.msgs}[\texttt{p}][\texttt{v}][\texttt{q.last\_dlvrd} + 1]$; by fairness, such delivery eventually occurs, resulting in $\texttt{q.last\_dlvrd}[\texttt{p}]$ being incremented. Therefore, every messages on $\texttt{q.msgs}[\texttt{p}][\texttt{v}]$ is eventually delivered to client at $\texttt{p}$, including the case of $\texttt{q} = \texttt{p}$.

It follows from this argument that every GCS.$\texttt{send}_\texttt{p}(\texttt{m})$ event at end-point $\texttt{p}$ that occurs after GCS.$\texttt{view}_\texttt{p}(\texttt{v}, \texttt{T})$ in α is eventually followed by a GCS.$\texttt{deliver}_\texttt{q}(\texttt{p}, \texttt{m})$ at every $\texttt{q} \in \texttt{v.set}$. ■

Chapter 9

Performance Analysis

In the previous chapter we formally proved that the GCS system satisfies Liveness property 5.2.2 of Section 5.2. Specifically, we showed that the GCS system guarantees that, if each of the end-points comprising a view receives the view as its last one from the MBRSHP service, then each of these end-points eventually delivers the view to its application client. In this chapter, we evaluate performance characteristics of the GCS system by quantifying what "eventually" means; we determine an upper-bound on the time by which all of the end-points comprising the last view deliver it to their application clients.

In the next section, we outline the formalism used for analyzing our GCS system. Specifically, we extend the I/O automaton model with the notions of timed executions and upper-bound constraints. Using these notions, one can derive upper-bounds on how fast certain high-order events must happen in a timed execution provided the execution satisfies upper-bound constraints on the timing of certain low-order events.

Section 9.2 gives an overview of the analysis. It presents the key definitions and assumptions of the analysis and outlines the high-level results derived in the three subsequent sections.

In Section 9.3, we apply the formalism of Section 9.1 to the analysis of the Virtual Synchrony algorithm run by GCS end-points. We derive an upper-bound on the time each end-point of a stabilized view delivers the view to its clients, in terms of the timing of the relevant MBRSHP events, and under certain timing assumptions about end-points' local steps, clients' responses, and message latencies (CO_RFIFO); we assume the timing assumptions hold after stability is reached.

Expressing performance characteristics of the Virtual Synchrony algorithm in terms of the timing of MBRSHP events constitutes the key part of our performance analysis. This is because the focus of our design is a novel algorithm for implementing Virtual Synchrony, given a membership service that satisfies the MBRSHP specification of Chapter 4.

Different membership services may have different performance characteristics. The advantage of our approach to performance analysis is that it yields naturally to composition: upper-bound results about the performance of a specific membership service can be composed with the upper-bound results about the performance of the Virtual Synchrony algorithm to yield upper-bound results about the performance of the GCS service in terms of the timings of the underlying network events.

In Section 9.4, we consider a specific membership service – the optimistic service of Keidar et al. [58] – and express reasonable time bounds on the events of this service in terms of the timings of the underlying network events. The reasonable bounds correspond to the "fast path" of [58]; according to the empirical tests reported in [58], the fast path case was observed in almost all of the view changes. We then compose these bounds with those derived in Section 9.3 to yield an upper-bound on the times when GCS end-points deliver new views following a network event, provided the assumed bounds on the MBRSHP and CO_RFIFO services hold; this is done in Section 9.5.

9.1 Formal Model

We analyze the time complexity of our GCS system within the same model in which we described the system — the I/O automaton model. We associate events in an execution with real time. We then assume upper-bound constraints on how fast certain events (such as local processing time and message delivery time) happen in the execution. By relying on these assumptions, we prove that certain higher-order events (such as a delivery of a new view) must happen within certain time.

Timed executions

Definition 9.1.1 *A* timed execution *of an automaton is a fair execution of the automaton, in which every event is associated with a real-valued time, so that the times are monotone nondecreasing and, in an infinite execution, approach infinity.*

If π_i is an event in a timed execution $\alpha = \mathtt{s}_0, (\pi_1, \mathtt{t}_1), \mathtt{s}_1, (\pi_2, \mathtt{t}_2), \ldots$, we use $\mathtt{T}[\pi_\mathtt{i}]$ to denote its time. Formally, an event in an execution is a pair consisting of an action and its index. Sometimes we will refer to events by their actions when the actions' positions in the execution are unambiguous, as for example π_i's position in α.

We order events in an execution α so that $\pi_i < \pi_j$ if $i < j$, that is, if π_i precedes π_j in α.

For simplicity, in our analysis, we assume that the considered timed executions are infinite.

We also let $\mathtt{t}_0 = 0$ and π_0 be an empty action.

Upper-bound constraints

Given specific upper-bound constraints on how fast certain events must happen in an execution, there may be many timed executions that satisfy these upper-bound constraints, that is, many ways to associate times with the events in an execution so that they satisfy the upper-bound constraints. We measure the *time* until some designated event π by the supremum of the times that can be assigned to π in all such executions. Likewise, we measure the *time* between two events by the supremum of the differences between the times that can be assigned to those two events.

Notice that placing upper-bound constraints on how fast events happen does not restrict the set of possible executions of an automaton. Also notice that we are working still in the context of the I/O automaton model for *asynchronous networks*; The processes of our system do not have any notion of time. This is unlike the I/O automaton model for *partially-synchronous networks* ([71], Ch. 23), where system component do have some information about time and can use this information.

Section 8.6 of [71] contains a similar approach to time analysis. In this approach, upper time bounds are associated with the equivalence classes in the task partition. Specifically, any task C may be associated with an upper-bound $\texttt{upper}_{\texttt{C}}$, which can be either a positive real number of ∞. A timed execution is defined the same way as in Definition 9.1.1, except for the following additional requirement: from any point in the execution, each task C can be enabled for time at most $\texttt{upper}_{\texttt{C}}$ before some action in C must occur.

We use a less restricted approach for the following reasons. We want to state upper-bounds as assumptions on particular types of events in particular situations in a particular execution. That is, we do not want to make general timing assumptions that always apply. Moreover, some of the timing assumptions we would like to make cannot be expressed by associating upper time bounds with tasks. In particular, we would want to assume an upper-bound on the delivery time of *each* CO_RFIFO message. That is, we would want to say that a CO_RFIFO.deliver event must happen within a certain time from the corresponding CO_RFIFO.send event. (We define what *corresponding* means below.) Associating an upper-bound d with each task $\texttt{C}_{\texttt{p,q}}$ of CO_RFIFO, would result in an upper-bound for the oldest message in transit from p to q; this would mean that, when a message is sent from p to q (and q is in both p.live_set and p.reliable_set), the message would be guaranteed to be delivered to q within time kd, where k is the number of messages on channel[p][q].

Identifying corresponding messages

In order to formally express that CO_RFIFO.deliver events must occur within a certain bound from their corresponding CO_RFIFO.send events, we need to formally define what *corresponding* events mean. That is, we need to express somehow that the deliver action delivers the m that was sent by the send action (and not just some other m with the same content). In the remainder of this section, we define the notion of *message correspondence* formally. These definitions are provided for completeness of the presentation; they are not essential for following the material in the subsequent sections.

If message loss was not possible, then we could have expressed the correspondence between send and deliver events simply by requiring the number of intermediate deliver events between the given send and deliver to be equal to the number of messages in transit from p to q at the time of the send event (i.e., the number of messages on CO_RFIFO.channel[p][q] in the state prior to the send event).

With message loss being possible, this definition is incorrect. For example, there maybe two CO_RFIFO.$\mathtt{send_p(set,m)}$ events, and one $\mathtt{lose(p,q)}$ event. Depending on whether the $\mathtt{lose(p,q)}$ event occurred before or after the second `send`, the subsequent `deliver` event corresponds either to the second or to the first `send`. However, if we were to adopt the above definition of correspondence, the `deliver` event would correspond to each of the `send` events.

The following definitions of message correspondence take into account message loss. Definition 9.1.2 below considers two states of the same CO_RFIFO channel and defines the correspondence for the messages that are in transit on this channel. Definition 9.1.3 then builds on top of Definition 9.1.2 to define the correspondence between CO_RFIFO.`send` and CO_RFIFO.`deliver` events.

Definition 9.1.2 (Corresponding CO_RFIFO messages)
Let $\beta = \mathtt{s_r}, (\pi_{\mathtt{r+1}}, \mathtt{t_{r+1}}), \mathtt{s_{r+1}}, (\pi_{\mathtt{r+2}}, \mathtt{t_{r+2}}), \ldots$ *be a timed execution fragment of* GCS *and* Ω *be a given set of* GCS *end-points.*

Consider the $\mathtt{i}$*th message on* CO_RFIFO.$\mathtt{channel[p][q]}$ *in state* $\mathtt{s_k}$ *and the* $\mathtt{j}$*th messages on* CO_RFIFO.$\mathtt{channel[p][q]}$ *in state* $\mathtt{s_l}$*, where* $\mathtt{r} \leq \mathtt{k} \leq \mathtt{l}$, $\mathtt{i} \geq \mathtt{j}$*, and* $\mathtt{p}$ *and* $\mathtt{q}$ *are members of* Ω*. These two messages* correspond*, provided:*

- $\mathtt{s_k}$ [CO_RFIFO].$\mathtt{channel[p][q](i)} = \mathtt{s_l}$ [CO_RFIFO].$\mathtt{channel[p][q](j)}$
- *The number of* CO_RFIFO.$\mathtt{deliver_{p,q}()}$ *events between the states* $\mathtt{s_k}$ *and* $\mathtt{s_l}$ *is* $(\mathtt{i} - \mathtt{j})$.
- *For every state* $\mathtt{s_n}$, $\mathtt{k} \leq \mathtt{n} \leq \mathtt{l}$*, the number of* CO_RFIFO.$\mathtt{lose(p,q)}$ *events between* $\mathtt{s_k}$ *and* $\mathtt{s_n}$ *is less than or equal to the sum of two terms:*
  - *the number of* CO_RFIFO.$\mathtt{send_p(set,m)}$ *events,* $\mathtt{q} \in \mathtt{set}$*, between* $\mathtt{s_k}$ *and* $\mathtt{s_n}$*; and*
  - $\mathtt{LengthOf}(\mathtt{s_k}$[CO_RFIFO].$\mathtt{channel[p][q]}) - \mathtt{i}$.

Definition 9.1.3 (Corresponding CO_RFIFO.`send` and CO_RFIFO.`deliver` events)
Let $\beta = \mathtt{s_r}, (\pi_{\mathtt{r+1}}, \mathtt{t_{r+1}}), \mathtt{s_{r+1}}, (\pi_{\mathtt{r+2}}, \mathtt{t_{r+2}}), \ldots$ *be a timed execution fragment of* GCS *and* Ω *be a given set of* GCS *end-points.*

Consider a CO_RFIFO.$\mathtt{send_p(set,m)}$ *event that occurs in* β *as some* $\pi_{\mathtt{k}}$*, and consider a* CO_RFIFO.$\mathtt{deliver_{p,q}(m)}$ *event,* $\mathtt{q} \in \mathtt{set}$*, that occurs in* β *as some* $\pi_{\mathtt{l+1}}$*, where* $\mathtt{r} \leq \mathtt{k} \leq \mathtt{l}$*, and* $\mathtt{p}$ *and* $\mathtt{q}$ *are members of* Ω*.*

The two events correspond *provided the last message on* $\mathtt{s_k}$ [CO_RFIFO].$\mathtt{channel[p][q]}$ *and the first message on* $\mathtt{s_l}$ [CO_RFIFO].$\mathtt{channel[p][q]}$ *correspond, as defined by Definition 9.1.2.*

Notice, that the definition is expressed simply in terms of the number of messages on the $\mathtt{channel[p][q]}$ queue and the numbers of `send`, `deliver` and `lose` events. It does not check whether the messages that are in transit are in fact those that are later delivered; this is ensured by the CO_RFIFO specification (Chapter 4).

Similarly to Definition 9.1.3, we define correspondence between a CO_RFIFO.$\mathtt{deliver_{p,q}(m)}$ event that occurs in β as some $\pi_{\mathtt{l+1}}$ and the $\mathtt{i}$th message on CO_RFIFO.$\mathtt{channel[p][q]}$ in state

$\mathtt{s_k}$ (the latter has to correspond to the first messages on $\mathtt{s_l}$ [CO_RFIFO].channel[p][q]); also between a CO_RFIFO.$\mathtt{send_p}$(set, m) event and the jth messages on $\mathtt{s_l}$ [CO_RFIFO].channel[p][q]. Likewise, we extend the definition of correspondence to relate CO_RFIFO.lose(p, q) events to the messages on CO_RFIFO.channel[p][q] queue and then to CO_RFIFO.$\mathtt{send_p}$(set, m) events.

9.2 High-level Overview

One of the most important performance characteristics of a GCS is how fast it delivers new views to its clients after the client's view membership changes. How fast this happens depends on how quickly the GCS detects a change in the current view membership, forms a new view that reflects the change, and establishes Virtual Synchrony.

In our design, the detection of membership changes and the formation of views is handled by an external membership service; the establishment of Virtual Synchrony is accomplished by the novel algorithm run by the end-points of the GCS. The design does not fix the membership service to a particular one; it can work with any membership service satisfying the MBRSHP specification of Chapter 4. As part of this specification, the membership service issues two types of notifications to the end-points: start_change and view. The start_change notifications are assumed to be issued as soon as the membership service detects a change in the membership; this allows the Virtual Synchrony algorithm to proceed in parallel with the membership service forming the new view. Once the end-points are notified of the new view, they complete the Virtual Synchrony algorithm and pass the new view to their clients.

Thus, the time from when a change in the view membership occurs until the GCS clients receive a view that reflects the change depends on

1. the performance characteristics of the Virtual Synchrony algorithm relative to the times of the start_change and view notifications, and
2. the performance characteristics of the specific membership service used in the design; that is, the times of the start_change and view notifications relative to the time of the view membership change.

Following this observation, we split our performance analysis into the following three parts: In Section 9.3, we analyze the performance characteristics of the Virtual Synchrony algorithm in terms of the times of the relevant start_change and view inputs from the membership service. Then, in Section 9.4, we state reasonable performance assumptions on the membership service, and, then, in Section 9.5, combine these assumptions with the performance properties derived for the Virtual Synchrony algorithm to obtain a performance property for the system as whole.

In the remainder of this section we present the key definitions and assumptions of the analysis and then outline the high-level results derived in the three subsequent sections.

9.2.1 Definitions

Group Events

A specific membership service may change views for a variety of reasons, such as for example new members joining, current members crashing, disconnecting, or partitioning, views merging together, communication links becoming temporarily congested, etc. We refer to such events, generally, as *group events* and denote all possible such events by an abstract type, `GE`.

Our specification, MBRSHP, of a membership service that is appropriate for the GCS design does not include the notion of group events. This is because the specification does not cover the liveness properties of the MBRSHP service, but only the safety properties of the interface between the membership service and the GCS end-points. The GCS design does not specify the liveness properties of the underlying membership service because liveness properties of the GCS system are made conditional on the behavior of the membership service; see discussion in Chapter 5 on page 65.

For the purpose of the performance analysis, we add group events as events of the GCS system. Formally, we use the signature extension construct, defined in Chapter 3, to add `group_event(e)`, $\mathtt{e} \in \mathtt{GE}$, actions to the input signature of the MBRSHP automaton. No other changes are made to the GCS system.

Intuitively, extending the signature of MBRSHP with the `group_event` inputs does not change the behavior of the GCS system. The sets of executions of the original GCS system and of the signature extended one are identical, except the executions of the latter may include `group_events`. Lemma 3.4.1 justifies this claim formally.

From here on, we use MBRSHP and GCS to refer to the signature extended versions of these automata, that is, we assume that `group_events` occur in executions of the GCS system.

We introduce an abstract function `ge()` that links membership events in an execution to the sets of group events that lead to the membership events. We do not define explicitly what it means to "lead" to a membership event; `ge()` is just an abstract function that allows us to talk about group events, and in particular the time of the last group event, that lead to a particular membership event.

Definition 9.2.1 *Given a* MBRSHP *event* π_i *in an execution* $s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots$ *of the* GCS *system,* $\mathtt{ge}(\pi_i)$ *is a finite set of events in the execution consisting of* `group_event`*s that lead to the* MBRSHP *event* π_i*.*

Fixed execution of GCS and relevant events

We restrict our analysis to executions in which a view becomes stable, as defined by Property 5.2.1 on page 66.

Definition 9.2.2 (Execution α and stable view $\mathtt{v}$)
For the rest of this chapter fix $\alpha = \mathtt{s}_0, (\pi_1, \mathtt{t}_1), \mathtt{s}_1, (\pi_2, \mathtt{t}_2), \ldots$ and $\mathtt{v}$ to be respectively a timed execution of the GCS *system and a view such that $\mathtt{v}$ becomes view stable in α (Property 5.2.1).*

Different end-points of view $\mathtt{v}$ may have different views prior to view $\mathtt{v}$. We denote the view of an end-point $\mathtt{p}$ prior to view $\mathtt{v}$ as $\mathtt{v}_\mathtt{p}^-$.

Definition 9.2.3 (Previous view notation) *For each $\mathtt{p} \in \mathtt{v.set}$, let $\mathtt{v}_\mathtt{p}^-$ refer to $\mathtt{p}$'s view prior to $\mathtt{v}$, that is, the value of $\mathtt{p.current\_view}$ in the state preceding the* MBRSHP.$\mathtt{view(v)_p}$ *event in α.*

View $\mathtt{v}$ being stable in α means that, for every end-point $\mathtt{p} \in \mathtt{v.set}$, the MBRSHP.$\mathtt{view_p(v)}$ event occurs in α and is followed by neither MBRSHP.$\mathtt{view_p}$ nor MBRSHP.$\mathtt{start\_change_p}$ events. Lemma 8.2.1 establishes that the MBRSHP.$\mathtt{view_p(v)}$ event is preceded by at least one MBRSHP.$\mathtt{start\_change_p(cid, set)}$ event, and that the last of these events has $\mathtt{cid} = \mathtt{v.startId[p]}$ and $\mathtt{set} \supseteq \mathtt{v.set}$. Lemma 8.2.2 establishes that MBRSHP.$\mathtt{view_p(v)}$ is followed by exactly one GCS.$\mathtt{view_p(v, *)}$ event. Our analysis relies on these MBRSHP and GCS events and their timings. The following definition introduces shortcut names for these events:

Definition 9.2.4 (Relevant MBRSHP and GCS events in α)
For each $\mathtt{p} \in \mathtt{v.set}$, let

- $\mathtt{mview_p}$ *refer to the* MBRSHP.$\mathtt{view_p(v)}$ *event in α.*
- $\mathtt{mstart_p}$ *refer to the last* MBRSHP.$\mathtt{start\_change_p(*, *)}$ *event in α. More formally,*

$$\mathtt{mstart_p} \overset{\text{def}}{=} \mathtt{max}\{\pi_\mathtt{i} : \pi_\mathtt{i} = \text{MBRSHP}.\mathtt{start\_change_p(*, *)}\}.$$

- $\mathtt{gcview_p}$ *refer to the* GCS.$\mathtt{view_p(v, *)}$ *event in α.*

Furthermore, let $\mathtt{first\text{-}mstart}$ *and* $\mathtt{last\text{-}mstart}$ *refer respectively to the first and the last events among the* $\mathtt{mstart_p}$ *events for all* $\mathtt{p} \in \mathtt{v.set}$*. More formally:*

$$\begin{aligned}
\mathtt{first\text{-}mstart} &\overset{\text{def}}{=} \mathtt{min}\{\pi_\mathtt{i} : \pi_\mathtt{i} = \mathtt{mstart_p} \textit{ for some } \mathtt{p} \in \mathtt{v.set}\}. \\
\mathtt{last\text{-}mstart} &\overset{\text{def}}{=} \mathtt{max}\{\pi_\mathtt{i} : \pi_\mathtt{i} = \mathtt{mstart_p} \textit{ for some } \mathtt{p} \in \mathtt{v.set}\}.
\end{aligned}$$

Likewise, for the $\mathtt{first\text{-}mview}$*,* $\mathtt{last\text{-}mview}$*,* $\mathtt{first\text{-}gcview}$*, and* $\mathtt{last\text{-}gcview}$ *events.*

Recall from Section 9.1, that, if π is an event in a timed execution, $\mathtt{T}[\pi]$ denotes its time. We use this notation to denote the times of the events identified in Definitions 9.2.4 and 9.2.5. For example, $\mathtt{T[last\text{-}mview]}$ denotes the time of the very last $\mathtt{mview}$ event in α at any of the end-points in $\mathtt{v.set}$.

The fact that view $\mathtt{v}$ becomes stable in α means that, at some point in the execution, there is a final group event that leads to view $\mathtt{v}$ becoming stable; after this final event, no other group events affecting the view occur. For the rest of this chapter, we fix $\mathtt{last\text{-}ge}$ to denote the final group event leading to stability of view $\mathtt{v}$.

Definition 9.2.5 (Final group event) *Let*

$$\texttt{last-ge} \overset{\text{def}}{=} \texttt{max}\{\texttt{max}(\texttt{ge}(\texttt{mview}_\texttt{p}))\ :\ \texttt{p} \in \texttt{v.set}\}.$$

One of the assumptions (Assumption 9.4.1) that we will make when considering the performance of the underlying membership algorithm will be that all the membership servers of view `v` perceive the final group event `last-ge` and produce `mstart` outputs within one message latency from the time `last-ge` occurs.

9.2.2 Timing Assumptions

The time it takes the GCS service to output new views to its clients after the clients' view changes depends on how fast the relevant low-order events occur. In particular, this time depends on message latency, local speed of GCS processes (i.e., GCS end-points and membership servers), and client latency during the `block`/`block_ok` synchronization. In our analysis, we assume that, after a view becomes stable, message latency in the underlying network component is δ, process speed is negligible, and the `block`/`block_ok` synchronization takes at most c units of time.

We use these assumptions in our derivation of upper time bounds for the high-order events, such as the deliveries of views to the GCS clients. For example, in our analysis of the Virtual Synchrony algorithm we show that the $\texttt{gcview}_\texttt{p}$ event must occur at every view member either within one δ and one c from $\texttt{T}[\texttt{mstart}]$ or immediately after `mview`, whichever of the two occurs later.

We formalize the low-order timing assumptions as generic constraints. The constraints apply to a given timed execution fragment of GCS.

Constraint 9.2.1 expresses our assumption that end-points' local steps take zero time. It states that a local action cannot remain enabled for a non-zero unit of time without being executed.

Constraint 9.2.1 (The zero bound on end-points' local steps)
Given a timed execution fragment, $\beta = \texttt{s}_\texttt{r}, (\pi_{\texttt{r}+1}, \texttt{t}_{\texttt{r}+1}), \texttt{s}_{\texttt{r}+1}, (\pi_{\texttt{r}+2}, \texttt{t}_{\texttt{r}+2}), \ldots$ *of* GCS*, a time index* $\texttt{t}_\texttt{r}$*, and a set of end-points* Ω*, the following holds:*

For all $\texttt{p} \in \Omega$*, and for each locally controlled action* π *of* $\text{GCS}_\texttt{p}$*, if* π *is enabled in a state* $\texttt{s}_\texttt{k}$*,* $\texttt{k} \geq \texttt{r}$*, then* $\exists\ \texttt{l} > \texttt{k}$ *such that* $\texttt{t}_\texttt{l} = \texttt{t}_\texttt{k}$ *and either* $\pi_\texttt{l} = \pi$*, or* π *is disabled in* $\texttt{s}_\texttt{l}$*.*

Constraint 9.2.2 requires a client to issue a `block_ok` response within c units of time from receiving a `block` request.

Constraint 9.2.2 (The c bound on client's `block_ok`)
Given a timed execution fragment, $\beta = \texttt{s}_\texttt{r}, (\pi_{\texttt{r}+1}, \texttt{t}_{\texttt{r}+1}), \texttt{s}_{\texttt{r}+1}, (\pi_{\texttt{r}+2}, \texttt{t}_{\texttt{r}+2}), \ldots$ *of* GCS*, a time index* $\texttt{t}_\texttt{r}$*, and a set of end-points* Ω*, the following holds:*

For all $\mathtt{p} \in \Omega$*, if* $\text{GCS}_\mathtt{p}.\mathtt{block\_ok}_\mathtt{p}()$ *is enabled in some state* $\mathtt{s_k}$*,* $\mathtt{k} \geq \mathtt{r}$*, then* $\exists\, \mathtt{l} > \mathtt{k}$ *such that* $\mathtt{t_l} \leq \mathtt{t_k} + c$ *and either* $\pi_\mathtt{l} = \text{GCS}_\mathtt{p}.\mathtt{block\_ok}_\mathtt{p}()$*, or* $\text{GCS}_\mathtt{p}.\mathtt{block\_ok}_\mathtt{p}()$ *is disabled in* $\mathtt{s_l}$*.*

Finally, Constraint 9.2.3 says that delivery of CO_RFIFO messages between lively connected end-points should take no more than δ units of time. The statement of this constraint uses the notion of message correspondence (see page 110).

Constraint 9.2.3 (The δ bound on CO_RFIFO message delivery)
Given a timed execution fragment, $\beta = \mathtt{s_r}, (\pi_\mathtt{r+1}, \mathtt{t_{r+1}}), \mathtt{s_{r+1}}, (\pi_\mathtt{r+2}, \mathtt{t_{r+2}}), \ldots$ *of* GCS*, a time index* $\mathtt{t_r}$*, and a set of end-points* Ω*, the following holds for every* $\mathtt{p}$ *and* $\mathtt{q} \in \Omega$*:*

For every message $\mathtt{m}$ *on the* $\mathtt{s_k}[\text{CO\_RFIFO}].\mathtt{channel}[\mathtt{p}][\mathtt{q}]$ *queue, if the corresponding* $\text{CO\_RFIFO}.\mathtt{deliver}_\mathtt{p,q}(\mathtt{m})$ *event occurs as some* $\pi_\mathtt{l}$*,* $\mathtt{l} \geq \mathtt{k} \geq \mathtt{r}$*, then* $\mathtt{t_l} \leq \mathtt{t_k} + \delta$*, provided* $\mathtt{q} \in \mathtt{s_i}[\text{CO\_RFIFO}].\mathtt{live}[\mathtt{p}]$ *for all* $\mathtt{k} \leq \mathtt{i} \leq \mathtt{l}$*.*

Constraint 9.2.3 implies that, for every $\text{CO\_RFIFO}.\mathtt{send}_\mathtt{p}(\mathtt{set}, \mathtt{m})$ event occurring in β as some $\pi_\mathtt{k}$, if the corresponding $\text{CO\_RFIFO}.\mathtt{deliver}_\mathtt{p,q}(\mathtt{m})$ event occurs as some $\pi_\mathtt{l}$, $\mathtt{l} \geq \mathtt{k}$, and if $\mathtt{q} \in \mathtt{s_i}[\text{CO\_RFIFO}].\mathtt{live}[\mathtt{p}]$ for all $\mathtt{k} \leq \mathtt{i} \leq \mathtt{l}$, then $\mathtt{t_l} \leq \mathtt{t_k} + \delta$.

Notice that Constraint 9.2.3 applies only if the corresponding CO_RFIFO.`deliver` events occur in β, and only when communication is live. This fits with our strategy of constraining only the times of events, and not the execution itself.

9.2.3 Key Results

In this section we summarize the high-level results of the performance analysis appearing in the next three sections.

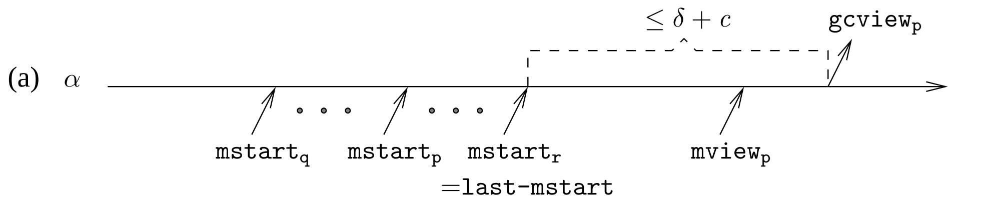
Virtual Synchrony Algorithm

The first part of the analysis upper-bounds the time by which every end-point `p` of the stable view `v` outputs the view to its clients. The upper-bound is expressed in terms of the times of the relevant membership events, `last-mstart` and $\mathtt{mview}_\mathtt{p}$, and in terms of the assumed upper-bounds on message latency δ and client response time c. Recall that in our analysis we consider end-points' local steps to cause insignificant delay.

The top-level result, appearing as Corollary 9.3.2, of Section 9.3 will prove that every end-point in `v.set` will produce the $\mathtt{gcview}_\mathtt{p}$ event either within $\delta + c$ from `last-mstart` or immediately after the $\mathtt{mview}_\mathtt{p}$ event, whichever is later. Stated more formally,

$$\mathtt{T}[\mathtt{gcview}_\mathtt{p}] \leq \max \begin{cases} \mathtt{T}[\mathtt{mview}_\mathtt{p}] \\ \mathtt{T}[\mathtt{last\text{-}mstart}] + c + \delta \end{cases} .$$

The upper-bound is consistent with the intuitive understanding of how the Virtual Synchrony algorithm works. The algorithm is initiated by every end-point at the time the



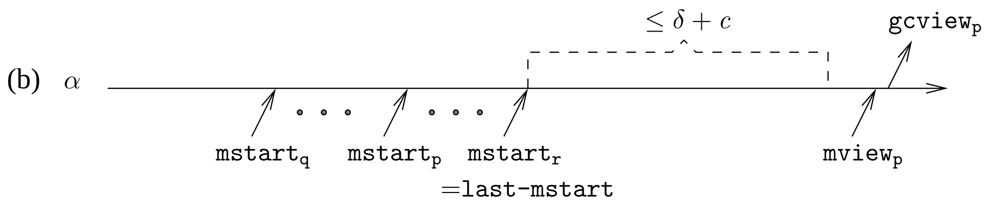


Figure 9.1: Analysis of the Virtual Synchrony algorithm. The $\mathtt{gcview_p}$ event must occur either (a) within $\delta + c$ from `last-mstart` or (b) right after $\mathtt{mview_p}$, whichever is later.

end-point receives a MBRSHP.`start_change` notification from the membership service. The algorithm involves blocking the client and exchanging synchronization and application messages with other end-points. The algorithm terminates when the end-point receives a view from the membership service and the right synchronization and application messages from other end-points. The latest times by which this happens are $\mathtt{T}[\mathtt{mview_p}]$ and $\mathtt{T}[\mathtt{last\text{-}mstart}] + c + \delta$ respectively.

Figure 9.1 illustrates this upper-bound result. It has two parts, each depicting execution sequence α, the `mstart` events, including the `last-mstart` event, the $\mathtt{mview_p}$ event, and the $\mathtt{gcview_p}$ event. The first part shows the situation when the membership service produces the final view before end-point `p` is ready to deliver the view to its client; in this case, the algorithm is guaranteed to terminate within $\delta + c$ from `last-mstart`. The second part shows the situation when the membership algorithm takes longer than $\delta + c$ from `last-mstart` to produces the final view; in this case end-point `p` delivers the view to its client immediately after the $\mathtt{mview_p}$ event.

Membership Service

The second part of the analysis suggests reasonable assumptions on the timing of the membership events, `T[last-mstart]` and `T[mview]`, relative to time of the final group event, `last-ge`. These assumptions are based on the typical properties of an optimistic membership algorithm of Keidar et al. [58].

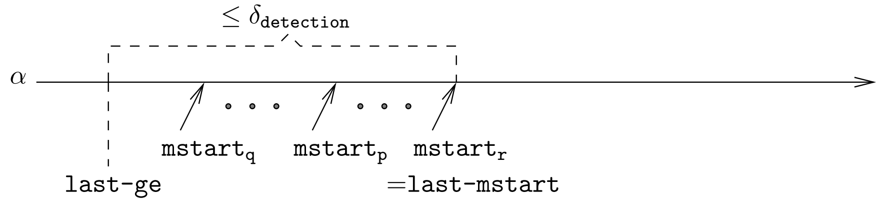
The first assumption (Assumption 9.4.1) about the performance properties of the underlying membership service concerns the time by which the membership service perceives the final group events and produces the `mstart` events. We assume that

$$\mathtt{T}[\mathtt{mstart_p}] \leq \mathtt{T}[\mathtt{last\text{-}ge}] + \delta_{\mathtt{detection}},$$

where $\delta_{\texttt{detection}}$ is an upperbound on group event detection latency. The exact value of $\delta_{\texttt{detection}}$ depends on the types of the final group events and on how the group event detection mechanism is implemented. However, in most cases the value of $\delta_{\texttt{detection}}$ is similar to the message latency δ.

For example, when final group events involve members leaving and/or joining the group, it is reasonable to assume that $\delta_{\texttt{detection}}$ is δ. Other types of group events may involve different, somewhat bigger bounds. For example, when timeout mechanisms are used to detect failed and disconnected members, $\delta_{\texttt{detection}}$ corresponds to the timeout values, which typically approximate δ but may be bigger.

Figure 9.2 illustrates this upper-bound. It shows execution α, the final group event `last-ge`, and the `mstart` events.



$\leq \delta_{\texttt{detection}}$
α
$\texttt{mstart}_\texttt{q}$
$\texttt{mstart}_\texttt{p}$
$\texttt{mstart}_\texttt{r}$
last-ge
=last-mstart

Figure 9.2: Reasonable assumption on the Membership Service. The `last-mstart` event occurs within $\delta_{\texttt{detection}}$ from the final event `last-ge` that leads to stability of view `v`.

The second assumption (Assumption 9.4.2) about the performance properties of the underlying membership service concerns the time by which the membership service produces the final view outputs relative to the times when the membership service perceives the final group event. We assume that

$$\texttt{T}[\texttt{mview}_\texttt{p}] \leq \texttt{T}[\texttt{last-mstart}] + \delta.$$

This assumption is justified by the fast path behavior of the membership algorithm of Keidar et al. [58]. We discuss the justifications for this assumption in Section 9.4.

Figure 9.3 illustrates this upper-bound. It shows execution α, the the `mstart` events, including the `last-mstart` event, and the $\texttt{mview}_\texttt{p}$ event occurring at end-point `p`.

Composing the Membership and Virtual Synchrony bounds

The final part of the analysis combines the upper-bound assumptions about the membership events relative to the time of the final group event and the upper-bound result derived for the final GCS view outputs relative to the times of the membership events. The combined upper-bound, appearing in Corollary 9.5.1, states that all the end-points of view `v` output view `v` to its clients within one detection and one message latencies $\delta_{\texttt{detection}} + \delta$ and within

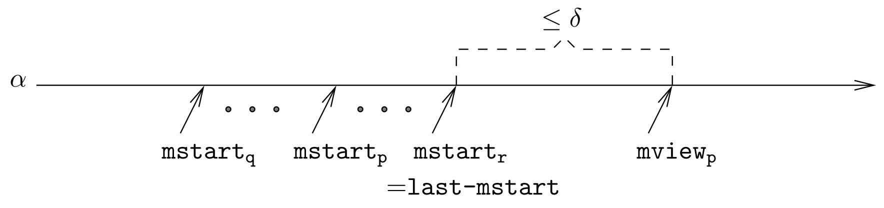


Figure 9.3: Reasonable assumption on the Membership Service. The $\mathtt{mview_p}$ event occurs within δ from the `last-mstart` event.

one client delay c after the final group event occurs:

$$\mathtt{T}[\mathtt{gcview_p}] \leq \mathtt{T}[\mathtt{last\text{-}ge}] + \delta_{\mathtt{detection}} + \delta + c.$$

Figure 9.4 illustrates this composed bound.

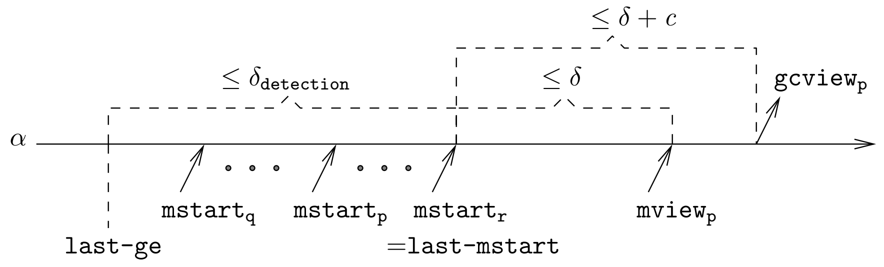


Figure 9.4: Composition of Membership and Virtual Synchrony bounds. The $\mathtt{last\text{-}gcview_p}$ event must occur within $\delta_{\mathtt{detection}}+\delta+c$ from the final event `last-ge` that leads to stability of view `v`.

Even though the bound is conditioned on various assumptions, we believe that the assumptions correspond to what is typically observed in real network environments. Therefore, the bound provides useful information in assessing performance characteristics of our GCS design.

9.3 Virtual Synchrony Algorithm

In this section, we analyze performance characteristics of the Virtual Synchrony algorithm run by GCS end-points. We restrict our attention to execution α, in which view `v` becomes stable (see Definition 9.2.2). Given α and `v`, we derive an upper-bound on the time by which every end-point that is a member of `v` delivers view `v` to its client. The upper-bound is expressed in terms of the timing of the relevant MBRSHP events (defined in Subsection 9.2.1) and the assumed upper-bound constraints on message latency, client response time, and end-points' speed; the constraints were defined in Subsection 9.2.2.

According to our algorithm, an end-point $\mathtt{p}$, that is in view $\mathtt{v}_{\mathtt{p}}^{-}$, is able to deliver view $\mathtt{v}$ to its client as soon as the following three milestones are accomplished:

- The end-point receives new view $\mathtt{v}$ from the MBRSHP service. Recall that the view contains a function `v.startId` linking every member of $\mathtt{v}$ to a start_change identifier.
- The end-point completes its participation in the synchronization protocol; that is it receives the synchronization message tagged with $\mathtt{v.startId(q)}$ from every end-point $\mathtt{q} \in \mathtt{v}_{\mathtt{p}}^{-}.\mathtt{set} \cap \mathtt{v.set}$. Recall that the synchronization message from $\mathtt{q}$ informs $\mathtt{p}$ of
  - whether $\mathtt{q}$ is in the transitional set of $\mathtt{v}_{\mathtt{p}}^{-}$ and $\mathtt{v}$, and
  - which application messages $\mathtt{q}$ commits to deliver in view $\mathtt{v}_{\mathtt{p}}^{-}$.
- The end-point delivers to its client all of the application messages identified by the synchronization protocol. These are the messages that are committed to delivery by the members of the transitional set of $\mathtt{v}_{\mathtt{p}}^{-}$ and $\mathtt{v}$.

Thus, the time by which end-point $\mathtt{p}$ delivers the new view to its client is bounded by the maximum of the times by which each of these milestones is accomplished; Theorem 9.3.3 of Subsection 9.3.3 formalizes this. The bounds on the times by which end-point $\mathtt{p}$ completes the synchronization protocol and delivers all of the application messages are established in subsections 9.3.1 and 9.3.2 respectively. Corollary 9.3.2 of Subsection 9.3.3, then, combines these bounds to produce the bound for the time when end-point $\mathtt{p}$ delivers $\mathtt{v}$ to its client in terms of the membership events and end-point independent constants. Roughly speaking, it states that end-point $\mathtt{p}$ delivers new view $\mathtt{v}$ to its client after it receives $\mathtt{v}$ from the MBRSHP service and within about one message latency away from the time everyone in `v.set` receives the final `start_change` notification.

9.3.1 Synchronization Protocol

In this subsection, we establish an upper-bound on the time an end-point completes its participation in the synchronization protocol.

Recall that the synchronization protocol involves end-points exchanging synchronization messages. An end-point $\mathtt{p}$, that is currently in some view $\mathtt{v}_{\mathtt{p}}^{-}$, completes its participation in the synchronization protocol for view $\mathtt{v}$ once it receives the synchronization message tagged with $\mathtt{v.startId(q)}$ from every end-point $\mathtt{q} \in \mathtt{v}_{\mathtt{p}}^{-}.\mathtt{set} \cap \mathtt{v.set}$.

The following definition pinpoints the earliest event in α after which $\mathtt{p}$'s participation in the synchronization protocol for view $\mathtt{v}$ is completed.

Definition 9.3.1 *For each end-point* $\mathtt{p} \in \mathtt{v.set}$*, we define* $\mathtt{sync}_{\mathtt{p}}$ *to refer to the earliest event in* α *after which* $\mathtt{p}$ *has in its* `sync_msg` *buffer the "right" synchronization message from every member in* $\mathtt{v}_{\mathtt{p}}^{-}.\mathtt{set} \cap \mathtt{v.set}$. *More formally,*

$$\begin{aligned}\mathtt{sync}_{\mathtt{p}} \overset{\text{def}}{=} \mathtt{min}\{\pi_k \quad &: \quad \pi_k \in \mathtt{acts}(\text{GCS}_{\mathtt{p}}) \\ &\wedge \quad \forall\, \mathtt{q} \in (\mathtt{v.set} \cap \mathtt{v}_{\mathtt{p}}^{-}.\mathtt{set})\ s_k[p].\mathrm{sync\_msg}[q][\mathtt{v.startId(q)}] \neq \perp\},\end{aligned}$$

where $\mathtt{v}_\mathtt{p}^-$ *denotes* $\mathtt{p}$*'s view prior to* $\mathtt{v}$ *(see Definition 9.2.4).*

Typically, $\mathtt{sync}_\mathtt{p}$ would correspond to the delivery of the last synchronization message completing the synchronization protocol at $\mathtt{p}$ for view $\mathtt{v}$. However, when the new view $\mathtt{v}$ is a singleton view, $\mathtt{sync}_\mathtt{p}$ would correspond to the $\mathtt{set\_cut}_\mathtt{p}()$ event because $\mathtt{v}_\mathtt{p}^-.\mathtt{set} \cap \mathtt{v.set} = \{\mathtt{p}\}$. This is why we did not define $\mathtt{sync}_\mathtt{p}$ in terms of CO_RFIFO.deliver events.

The following lemma establishes that an end-point $\mathtt{p}$ completes its participation in the synchronization protocol for view $\mathtt{v}$ within one message latency δ away from the time everyone in $\mathtt{v.set}$ receives the final $\mathtt{start\_change}$ notification, plus the client's blocking c.

Lemma 9.3.1 *The following bound holds for each* $\mathtt{p} \in \mathtt{v.set}$*:*

$$\mathtt{T}[\mathtt{sync}_\mathtt{p}] \le \mathtt{T}[\mathtt{last\text{-}mstart}] + c + \delta,$$

provided the suffix of α *following the* $\mathtt{last\text{-}mstart}$ *event satisfies Constraints 9.2.1, 9.2.2, and 9.2.3 with the given time index* $\mathtt{T}[\mathtt{last\text{-}mstart}]$ *and set of end-points* $\mathtt{v.set}$.

Proof 9.3.1: Liveness Proof 8.2.2 establishes that each end-point $\mathtt{q} \in \mathtt{v.set}$ eventually sends to every end-point in $\mathtt{v.set}$ a synchronization message tagged with the start-change identifier $\mathtt{v.startId(q)}$. By following that part of the proof and using the fact that α satisfies Constraints 9.2.1 and 9.2.2 after the $\mathtt{last\text{-}mstart}$ event, we derive that every end-point $\mathtt{q} \in \mathtt{v.set}$ sends the right synchronization message to everyone in $\mathtt{v.set}$ at most by $\mathtt{T}[\mathtt{last\text{-}mstart}] + c$. This is because the longest time it can take an end-point to send a synchronization message after receiving a $\mathtt{start\_change}$ notification is to issue a $\mathtt{block}()$ request to the client, wait until the client responds with the $\mathtt{block\_ok}()$, and then execute $\mathtt{set\_cut}_\mathtt{q}()$ and CO_RFIFO.$\mathtt{send}_\mathtt{p}(\mathtt{set}, \langle \text{`}\mathbf{sync\_msg}\text{'}, \mathtt{v.startId(q)}, \mathtt{v}_\mathtt{q}^-, \mathtt{cut}\rangle)$.

Liveness Proof 8.2.2 also establishes that an end-point $\mathtt{p} \in \mathtt{v.set}$ eventually receives all the right synchronization messages from every end-point in $\mathtt{v}_\mathtt{p}^-.\mathtt{set} \cap \mathtt{v.set}$. The desired bound for $\mathtt{T}[\mathtt{sync}_\mathtt{p}]$ follows from the derived in the previous paragraph upper-bound on the time by which the synchronization messages are sent to $\mathtt{p}$ and the fact that α satisfies Constraint 9.2.3 after the $\mathtt{last\text{-}mstart}$ event.limited ■

9.3.2 Message Delivery

In this subsection, we derive an upper-bound on the time an end-point delivers to its client all of the application messages identified by the synchronization protocol.

The time by which an end-point $\mathtt{p}$ delivers messages to its client is affected by the following two processes:

- Assuming that the message originated from the client of some other end-point, the end-point must first receive the actual message from CO_RFIFO before it can deliver it. How fast this occurs depends on the CO_RFIFO latency and also on the parameters of the forwarding strategy used to recover messages lost from disconnected end-points.

- Moreover, while the end-point participates in the synchronization protocol, it delivers only those application messages beyond its own cut that it has already determined appropriate for delivery. For this, the application messages have to be identified by the synchronization message from some end-point that has already been determined to belong to $\mathtt{p}$'s transitional set. Thus, in the worst case, an application message that is ready to be delivered, can remain in $\mathtt{p}$'s $\mathtt{msgs}$ buffer until $\mathtt{p}$ receives the view from the MBRSHP service and until it receives the synchronization message from the last member of its transitional set.

We formalize this relationship in Lemma 9.3.2, after making two auxiliary definitions. Then, after the lemma, we consider a particular message forwarding strategy and its implications on the time of message delivery. We finish this subsection with a Corollary 9.3.1 that states the upper-bound on the time by which $\mathtt{p}$ completes delivering messages in view $\mathtt{v}_\mathtt{p}^-$.

The messages that are identified by the synchronization protocol are those that comprise the maximal cut obtained from the members of the transitional set of $\mathtt{v}_\mathtt{p}^-$ and $\mathtt{v}$; the following definition expresses the transitional set and the maximal cut in terms of the $\mathtt{p}$'s state preceding the $\mathtt{mview}_\mathtt{p}$ event.

Definition 9.3.2 *For each* $\mathtt{p} \in \mathtt{v.set}$ *we define* $\mathtt{TS}_\mathtt{p}$ *and* $\mathtt{cut}_\mathtt{p}$ *to be respectively the transitional set and the cut that* $\mathtt{p}$ *calculates when it delivers view* $\mathtt{v}$*. More formally, assuming that the* $\mathtt{mview}_\mathtt{p}$ *event occurs as some* $\pi_\mathtt{k}$ *in* α*,*

$$\mathtt{TS}_\mathtt{p} \stackrel{\text{def}}{=} \{\mathtt{q} \in \mathtt{v}_\mathtt{p}^-.\mathtt{set} \cap \mathtt{v.set} \ : \ \mathtt{s}_{\mathtt{k}-1}[\mathtt{p}].\mathtt{sync\_msg}[\mathtt{q}][\mathtt{v.startId(q)}].\mathtt{view} = \mathtt{v}_\mathtt{p}^-\}$$

$$(\forall\ \mathtt{q} \in \mathtt{v}_\mathtt{p}^-.\mathtt{set})\ \mathtt{cut}_\mathtt{p}(\mathtt{q}) \stackrel{\text{def}}{=} \mathtt{max}\{\mathtt{s}_{\mathtt{k}-1}[\mathtt{p}].\mathtt{sync\_msg}[\mathtt{r}][\mathtt{v.startId(r)}].\mathtt{cut(q)} \ : \ \mathtt{r} \in \mathtt{TS}_\mathtt{p}\}$$

Definition 9.3.3 pinpoints the earliest events, $\mathtt{rcvd}_\mathtt{p}$, in the execution after which end-point $\mathtt{p}$ has on its $\mathtt{msgs}$ queue all of the messages it is supposed to deliver in view $\mathtt{v}_\mathtt{p}^-$; likewise, event $\mathtt{msgs}_\mathtt{p}$ pinpoints the earliest event after which all of these messages are delivered to $\mathtt{p}$'s client.

Definition 9.3.3 *For each* $\mathtt{p} \in \mathtt{v.set}$*, we define* $\mathtt{rcvd}_\mathtt{p}$ *to refer to the earliest event in* α *after which* $\mathtt{p}$ *has on its* $\mathtt{msgs}$ *queue all the messages it is supposed to deliver to its client in view* $\mathtt{v}_\mathtt{p}^-$*. More formally,*

$$\begin{aligned}\mathtt{rcvd}_\mathtt{p} \stackrel{\text{def}}{=} \mathtt{min}\{\pi_\mathtt{k} \ \ : &\ \ \pi_\mathtt{k} \in \mathtt{acts}(\mathrm{GCS}_\mathtt{p}) \\ \wedge &\ \ (\forall\ \mathtt{q} \in \mathtt{v}_\mathtt{p}^-.\mathtt{set})\ (\forall\ 0 \le \mathtt{i} \le \mathtt{cut}_\mathtt{p}(\mathtt{q}))\ \mathtt{s}_\mathtt{k}[\mathtt{p}].\mathtt{msgs}[\mathtt{q}][\mathtt{v}_\mathtt{p}^-](\mathtt{i}) \neq \perp\},\end{aligned}$$

where $\mathtt{cut}_\mathtt{p}$ *is the cut that* $\mathtt{p}$ *calculates when it delivers view* $\mathtt{v}$ *(see Definition 9.3.2).*

Likewise, for each $\mathtt{p} \in \mathtt{v.set}$*, we define* $\mathtt{msgs}_\mathtt{p}$ *to refer to the earliest event in* α *after which* $\mathtt{p}$ *has delivered to its client all the messages it is supposed to deliver in view* $\mathtt{v}_\mathtt{p}^-$*. More formally,*

$$\begin{aligned}\mathtt{msgs}_\mathtt{p} \stackrel{\text{def}}{=} \mathtt{min}\{\pi_\mathtt{k} \ \ : &\ \ \pi_\mathtt{k} \in \mathtt{acts}(\mathrm{GCS}_\mathtt{p}) \\ \wedge &\ \ (\forall\ \mathtt{q} \in \mathtt{v}_\mathtt{p}^-.\mathtt{set})\ \mathtt{s}_\mathtt{k}[\mathtt{p}].\mathtt{last\_dlvrd}[\mathtt{q}] = \mathtt{cut}_\mathtt{p}(\mathtt{q})\}.\end{aligned}$$

Note that, like $\mathtt{sync_p}$, we did not define $\mathtt{rcvd_p}$ and $\mathtt{msgs_p}$ in terms of the last CO_RFIFO and GCS `deliver` events. This is because such definitions would not be well defined for the executions in which `p` delivers no messages in view $\mathtt{v_p^-}$. For such executions, $\mathtt{rcvd_p} = \mathtt{msgs_p} = \pi_0$, and $\mathtt{T}[\mathtt{rcvd_p}] = \mathtt{T}[\mathtt{msgs_p}] = 0$ (as assumed in Section 9.1 on page 108).

The following lemma upper-bounds the time by which an end-point delivers to its client all of the application messages it is supposed to deliver in view $\mathtt{v_p^-}$. The upper-bound is stated as some constant number of local steps plus the maximum of the times by which end-point `p`: has all of these messages in its `msgs` queue, receives the MBRSHP view, and completes the synchronization protocol. This bound follows immediately from the preconditions placed on the GCS.$\mathtt{deliver_p}$ action.

Lemma 9.3.2 *The following bound holds for each* $\mathtt{p} \in \mathtt{v.set}$*:*

$$\mathtt{T}[\mathtt{msgs_p}] \leq \mathtt{max} \begin{cases} \mathtt{T}[\mathtt{mview_p}] \\ \mathtt{T}[\mathtt{synch_p}] \\ \mathtt{T}[\mathtt{rcvd_p}] \end{cases},$$

provided the suffix of α *following the* `last-mstart` *event satisfies Constraints 9.2.1 with the given time index* $\mathtt{T}[\mathtt{last\text{-}mstart}]$ *and set of end-points* `v.set`*.*

Proof : Once an end-point enters a synchronization protocol it restricts delivery of application messages only to those identified for delivery by the synchronization protocol. According to the preconditions on the GCS.$\mathtt{deliver_p}$ events, even when the end-point has a message on its `msgs` queue, in the worst case the end-point may delay the delivery of this message until it receives MBRSHP.`view` and the final synchronization message completing the synchronization protocol. Once message delivery is enabled, it takes zero time to deliver messages to the clients, according to Constraint 9.2.1. ■

The bound on $\mathtt{T}[\mathtt{rcvd_p}]$ depends on the message forwarding strategy that end-points use to recover lost messages sent originally by disconnected end-points. For the purpose of the analysis, we assume the message forwarding strategy that minimizes this bound at the cost of increased message complexity; existing designs of Virtually Synchronous GCSs suggest similar strategies.

According to this strategy, end-points periodically inform others about the messages they have received in their current view; upon receiving a `start_change` notification, each end-point forwards to other end-points those messages that were not reported as received by these end-points. The strategy can be straightforwardly modeled using I/O Automata. With this strategy, it takes at most $\mathtt{T}[\mathtt{last\text{-}mstart}] + \delta$ for the forwarded messages to arrive to their destinations. It takes at most the same time for all the original messages to arrive. Thus, $\mathtt{rcvd_p}$ occurs about one message latency away from the final `start_change` notification., as formalized in the following assumption:

Assumption 9.3.1 *The following bound holds for each* $\mathtt{p} \in \mathtt{v.set}$*:*

$$\mathtt{T}[\mathtt{rcvd_p}] \leq \mathtt{T}[\mathtt{last\text{-}mstart}] + \delta,$$

provided the suffix of α following the `last-mstart` *event satisfies Constraints 9.2.1 and 9.2.3 with the given time index* T[`last-mstart`] *and set of end-points* `v.set`*.*

By combining Lemmas 9.3.1 and 9.3.2 and Assumption 9.3.1, we get the following Corollary.

Corollary 9.3.1 *The following bound holds for each* $\texttt{p} \in \texttt{v.set}$*:*

$$\texttt{T}[\texttt{msgs}_\texttt{p}] \leq \texttt{max} \begin{cases} \texttt{T}[\texttt{mview}_\texttt{p}] \\ \texttt{T}[\texttt{last-mstart}] + c + \delta \end{cases},$$

provided Assumption 9.3.1 holds and the suffix of α following the `last-mstart` *event satisfies Constraints 9.2.1, 9.2.2, and 9.2.3 with the given set of end-points* `v.set` *and time index* T[`last-mstart`]*.*

9.3.3 View Delivery

In this section we combine the results of the preceding sections to derive an upper-bound on the time an end-point `p` delivers view `v` to its client. We first state a high-level Theorem 9.3.3 that expresses this upper-bound in terms of the times when the end-point receives the view from the MBRSHP service, completes the synchronization protocol, and delivers to its client all of the messages identified by the synchronization protocol. Then, we plug in the bounds for these events to yield a low-level Corollary 9.3.2 upper-bounding $\texttt{T}[\texttt{gcview}_\texttt{p}]$ solely in terms of the timing of the relevant MBRSHP events and the assumed timing constraints.

Theorem 9.3.3 (High Level) *The following bound holds for each* $\texttt{p} \in \texttt{v.set}$*:*

$$\texttt{T}[\texttt{gcview}_\texttt{p}] \leq \texttt{max} \begin{cases} \texttt{T}[\texttt{mview}_\texttt{p}] \\ \texttt{T}[\texttt{synch}_\texttt{p}] \\ \texttt{T}[\texttt{msgs}_\texttt{p}] \end{cases},$$

provided the suffix of α following the `last-mstart` *event satisfies Constraint 9.2.1 with the given time index* T[`last-mstart`] *and set of end-points* `v.set`*.*

Proof 9.3.3: Follows from the preconditions on $\texttt{gcview}_\texttt{p}$ and Liveness Proof 8.2.2. By Constraint 9.2.1, end-point `p` must deliver view `v` to its client within zero time after it receives the view from the MBRSHP service ($\texttt{T}[\texttt{mview}_\texttt{p}]$), completes the synchronization protocol ($\texttt{T}[\texttt{sync}_\texttt{p}]$), and delivers all of the application messages identified by the synchronization protocol ($\texttt{T}[\texttt{msgs}_\texttt{p}]$), as well as sends out via CO_RFIFO all of its client's application messages and its synchronization message (all local steps). ■

When we plug in bounds for $\texttt{T}[\texttt{sync}_\texttt{p}]$ and $\texttt{T}[\texttt{msgs}_\texttt{p}]$ from Lemma 9.3.1 and Corollary 9.3.1, we get the following bound for $\texttt{T}[\texttt{mview}_\texttt{p}]$:

Corollary 9.3.2 (Low level) *The following bound holds for each* $\texttt{p} \in \texttt{v.set}$*:*

$$\texttt{T}[\texttt{gcview}_\texttt{p}] \leq \texttt{max} \begin{cases} \texttt{T}[\texttt{mview}_\texttt{p}] \\ \texttt{T}[\texttt{last-mstart}] + c + \delta \end{cases},$$

provided Assumption 9.3.1 holds, and the suffix of α *following the* `last-mstart` *event satisfies Constraints 9.2.1, 9.2.2, and 9.2.3 with the given set of end-points* `v.set` *and time index* T[`last-mstart`].

9.4 Membership Service

In the previous section we have analyzed the performance characteristics of the Virtual Synchrony algorithm. In particular, in Corollary 9.3.2, we have stated an upper-bound on the time by which every end-point of a stabilized view delivers the view to its client; the upper-bound on $\texttt{T}[\texttt{gcview}_\texttt{p}]$ is stated in terms of the timing of the relevant MBRSHP events: $\texttt{mview}_\texttt{p}$ and `last-mstart`. The times by which these MBRSHP events happen in an execution depend on the performance characteristics of the specific membership algorithm used in the design.

In this section, we consider one such algorithm, [58], and express upper-bounds on the times of the $\texttt{mview}_\texttt{p}$ and `last-mstart` events in terms of the timing of the final group event affecting the view's membership. In our analysis, we concentrate only on certain common executions of the membership algorithm. We compose the Membership Service upper-bounds with the Virtual Synchrony upper-bounds (given by Corollary 9.3.2) in Section 9.5.

The membership algorithm of [58] is implemented by a set of membership servers running on top of a network event notification service (NE) and a reliable FIFO service (RFIFO). All servers run the same algorithm. Each server handles a number of clients – end-points in our design. The service of [58] satisfies our MBRSHP specification of Chapter 4; the `start_change` interface was specifically built into [58] to accommodate our GCS design.

The NE service informs membership servers of the changes in the group's membership. The changes may occur because of clients joining and leaving the group, clients crashing, and communication links failing and recovering. The NE service guarantees that, if a membership set Ω becomes stable, every membership server of Ω eventually perceives the group's membership as Ω. The RFIFO service guarantees that each message sent from one server `r` to another `u` is either eventually delivered or the NE service eventually notifies `r` that all the clients of `u` are unreachable. The RFIFO service also guarantees that messages sent from one server to another are delivered in FIFO order. This means that, if a server sends a message `m` and then another message `m'` both to server `u`, and if `u` receives both of these messages, then it receives `m` before `m'`.

Each membership server executes an algorithm that has two paths: fast and slow. The fast path addresses scenarios that are typical in real networks; empirical testing of [58] showed that the fast path was taken in more than 99% of the view changes. The fast path requires one message exchange among the affected servers. However, certain out-of-sync scenarios cannot be resolved by the fast path; they are handled by the slow path, which requires several message exchanges.

In our analysis, we concentrate solely on the fast path; we assume that, after a membership set becomes stable, the membership algorithm succeeds in forming the last view using the fast path. We will make the same assumption when evaluating performance characteristics of other GCS services in the next section.

Briefly, the fast path of [58] is as follows. Whenever the server is informed by the NE service about the changes in its group's membership, the server sends a `start_change` notification to its clients and a proposal to other servers. The `start_change` notification and the proposal both include a fresh start_change identifier and the updated membership set, say Ω. The server outputs a new view to its clients when it collects proposals for the new membership Ω from all the servers of Ω. The view identifier is one more than the maximum of the collected start_change identifiers; the membership set is Ω; and the `startId` mapping maps every client to its start_change identifier.

In executions in which a membership set Ω becomes stable, the NE service guarantees that all the servers of Ω eventually perceive the group's membership as Ω. We assume that it takes at most one detection latency $\delta_{\texttt{detection}}$ from the time the final group event `last-ge` affecting the membership occurs until every membership server of Ω perceives the group's membership as Ω. Thus, within one detection latency from T[`last-ge`], every membership server of Ω sends its last `start_change` notification and its last proposal; we formalize this in Assumption 9.4.1 below. Within one more message latency, every server of Ω must receive all the proposals and output its last view; we formalize this in Assumption 9.4.2.

Assumption 9.4.1 *The following bound holds for each* $\texttt{p} \in \texttt{v.set}$*:*

$$\texttt{T}[\texttt{mstart}_\texttt{p}] \leq \texttt{T}[\texttt{last-ge}] + \delta_{\texttt{detection}},$$

where `last-ge` *is the final group event that results in the group's membership stabilizing to* `v.set`*, and* $\delta_{\texttt{detection}}$ *is the detection latency – similar to message latency* δ*.*

Notice that the $\texttt{mstart}_\texttt{p}$ events correspond to the events at the clients of the MBRSHP service, i.e., at the GCS end-points. In the bound for $\texttt{T}[\texttt{mstart}_\texttt{p}]$, the time interval it takes for the `start_change` notification to travel from `p`'s membership server to `p` is included in $\delta_{\texttt{detection}}$. The same applies to the $\texttt{mview}_\texttt{p}$ events in the next assumption.

Assumption 9.4.2 *The following bound holds for each* $\texttt{p} \in \texttt{v.set}$*:*

$$\texttt{T}[\texttt{mview}_\texttt{p}] \leq \texttt{T}[\texttt{last-mstart}] + \delta,$$

where δ *approximates message latency – the same constant as in Constraint 9.2.3.*

9.5 Composition of the Membership and the Virtual Synchrony bounds

Now that we have expressed reasonable upper-bounds on the behavior of the underlying membership service, we can compose them with the performance result for the Virtual Synchrony algorithm (given by Corollary 9.3.2).

The composition yields an upper-bound on how long it takes the GCS end-points to deliver their last views after the final group event occurs. Roughly speaking, this upper-bound states that every end-point delivers the last view about two latencies – one detection latency and one message latency – away from the time the final group event occurs. The detection latency upper-bounds the time it takes for the information about the group event to reach the membership servers and the GCS end-points; the message latency upper-bounds the time it takes the membership algorithm and the Virtual Synchrony algorithm to execute.

Corollary 9.5.1 *Assuming that* `last-ge` *is the final group event that results in the group's membership stabilizing to* `v.set`*, the following bound holds for each* $\mathtt{p} \in \mathtt{v.set}$*:*

$$\mathtt{T[gcview_p]} \le \mathtt{T[last\text{-}ge]} + \delta_{\mathtt{detection}} + \delta + c,$$

provided Assumptions 9.3.1, 9.4.1, and 9.4.2 hold, and the suffix of α *following the* `last-mstart` *event satisfies Constraints 9.2.1, 9.2.2, and 9.2.3 with the given set of end-points* `v.set` *and time index* `T[last-mstart]`*.*

Proof : Follows immediately from Corollary 9.3.2 and Assumptions 9.4.1 and 9.4.2. ■

Corollary 9.5.1 expresses an upper-bound on how long it takes the GCS end-points of a stabilized membership to deliver the last views to their application clients after the final group event affecting the membership occurs. The upper-bound applies only to those executions that satisfy the abovementioned constraints and assumptions. Since such executions correspond to what we believe is typically observed in real network environments, the derived upper-bounds provide useful information in assessing performance characteristics of our GCS service.

In comparison, existing Virtual Synchrony algorithms, such as [43, 6, 82, 12, 47, 8], identify synchronization messages by tagging them with a common identifier. Some initial communication is performed first, before synchronization messages are communicated, in order to agree upon a common identifier and to distribute it to the members of the forming view. Therefore, existing algorithms require an additional communication round to deliver the last view after the final group event affecting the stable component occurs.

An additional advantage of our algorithm is that, unlike existing solutions (e.g., [8, 47, 12, 82]), our algorithm is able to respond dynamically to cascading connectivity changes, without wasting resources on handling obsolete network situations. This is illustrated by Example 6.2.1 in Chapter 6. Expressing the performance characteristics of our algorithm formally for this case is the subject of future work.

Illustration

Figure 9.5 illustrates Corollary 9.5.1. It depicts the key messages and events produced by the GCS system in response to the final group event that leads to stability of a component $\{\mathtt{p}, \mathtt{q}\}$.

Horizontal lines represent passage of time at different components of the system. There are six components (listed on the left); three per each member: application CLIENT, GCS end-point, and MBRSHP server. Bold lines represent events and messages of the GCS end-points.

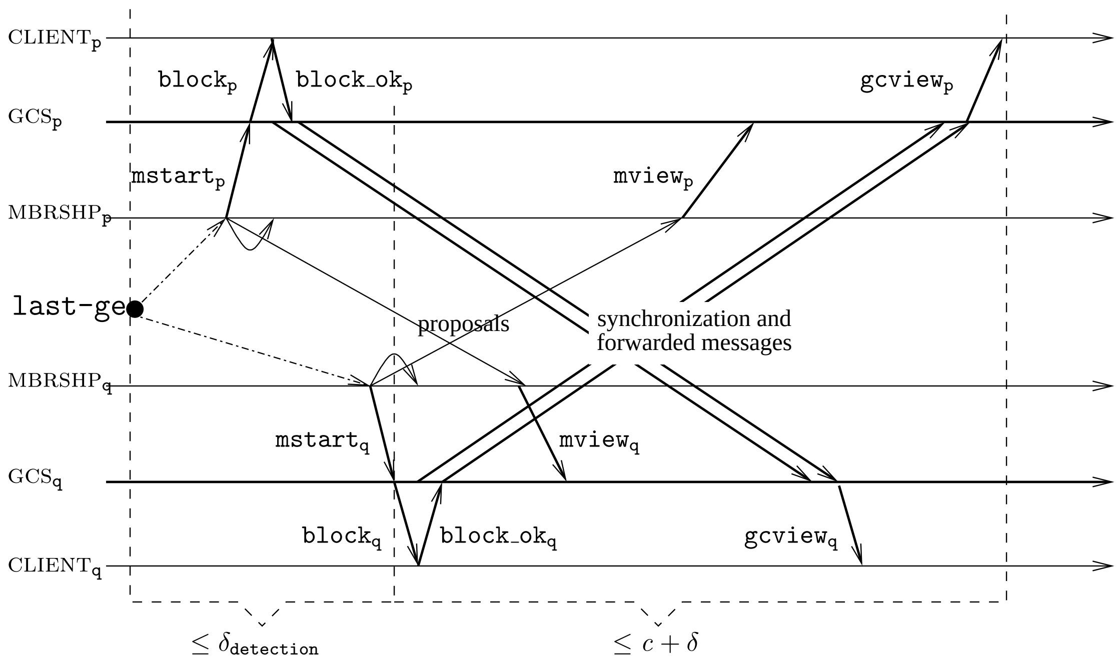


Figure 9.5: Flow of messages and events in the GCS system after a component stabilizes.

The black dot on the left represents the final group event `last-ge`. The dashed-dotted lines originating at `last-ge` represent propagation of information about the `last-ge` to the membership servers; the lines do not necessarily correspond to messages. The specifics of how the information about group events propagates to membership servers depends on the specific event detection mechanism employed by the GCS and on the specific types of the final group events. According to Assumption 9.4.1, the membership servers detect the final group event `last-ge` within $\delta_{\mathtt{detection}}$ time after the `last-ge`; the dashed bracket at the bottom of the figure identifies this time interval.

When a membership server detects the `last-ge` it issues the `mstart` notification to the GCS end-points that it serves and sends a view proposal to the membership servers handling the new membership. Both the `mstart` and the proposals include a fresh start_change identifier and the new membership set. Once the membership server collects proposals for the new membership from all the servers of the membership, the server outputs `mview` to its GCS end-points; the view identifier in the `mview` is one more than the maximum of the collected start_change identifiers. According to Assumption 9.4.2, the `mview` events occur at every end-point within δ from the `last-mstart`.

In parallel to the MBRSHP servers exchanging proposals for the new view, the GCS end-points exchange synchronization and forwarded messages among themselves. Synchronization messages are sent after the end-points complete the `block`/`block_ok` synchronization with their clients; this takes at most c time after `last-mstart` (Assumption 9.2.2). The synchronization messages arrive within δ time from the time they are sent out (Assumption 9.2.3). The forwarded messages are sent according to the specific forwarded strategy used in the GCS system; for the sake of the performance analysis, we assume a strategy that starts forwarding (unstable) messages as soon as the `mstart` notifications arrive; it completes within δ time (Assumption 9.3.1).

According to Theorem 9.3.3, the GCS end-point delivers `gcview` to its application client after it receives the view form the membership server, completes the synchronization protocol, and delivers to its client all of the application messages identified by the synchronization protocol. Taking into account all of the results and assumptions we have established about the GCS system, Corollary 9.5.1 upperbounds $\texttt{T}[\texttt{gcview}_\texttt{p}]$ by $\texttt{T}[\texttt{last-ge}] + \delta_{\texttt{detection}} + \delta + c$; see the dashed bracket on the figure.

Chapter 10

Application Example: Interim-Atomic Data Service

In this chapter, we illustrate the utility of our GCS system by describing a simple application that can be effectively built using GCS. The application implements a variant of a data service that allows a dynamic group of clients to access and modify a replicated data object. The application is prototypical of some collaborative computing applications, such as a shared white-board application (e.g., [74, 84]).

The application we present implements particular consistency guarantees regarding how different clients perceive the data object. We call this type of consistency *interim atomicity*; and the data service that satisfies interim atomicity — *interim-atomic data service* (IADS). Roughly speaking, interim atomicity guarantees that, while the underlying network component is stable, clients perceive the data object as *atomic* [64, 71, Ch. 13][1]. During periods of instability, the clients' perceptions of the object may diverge from the atomic one. The non-atomic semantics may persist until after the underlying component becomes stable again. When stability is regained, the atomic semantics is restored within some finite amount of time: The clients comprising the stable component are informed about the current membership of the client group and the new state of the data object. The new state is computed as an application-specified merge of the states of the members' object replicas. From that point on while stability lasts, the clients again perceive the object as an atomic one.

The IADS application can be conveniently built using GCS, as we demonstrate by presenting a simple algorithm, IADS, that operates atop GCS. The algorithm follows the *active replication/state-machine* approach [66, 83] and utilizes the state-transfer protocol of Amir, et al. [5].

Active replication, also known as a *state-machine* approach, is one of the standard ways to implement replicated data services [66, 83]. In a nutshell, according to this approach, the service is provided by a collection of servers, each maintaining a replica of the data

[1]Atomic objects are also known as *linearizable* [50] and sometimes are referred to as *strongly-consistent*, *non-replicated*, and *one-copy equivalent* [15, 3]

object. Clients' requests to modify the data object are sent to all the servers. The servers apply these requests to their object replicas all in the same order, and send responses to the clients. Since different object replicas undergo the same modifications, the object replicas stay consistent, and the clients perceive the replicated data object as an atomic one.

In fault-prone, asynchronous environments, replicated data services are typically organized to alternate between two modes of operation: *normal* and *state-transfer* (e.g., [83, 41, 23]). In normal mode, clients' requests are processed according to the active replication approach. When previously disconnected replicas reconnect, state-transfer is used to bring these replicas to a common state. This common state has to be consistent with the current states of the replicas and is suitable for the resumption of the normal mode of operation. After state-transfer completes, the servers resume normal mode, i.e., process their clients' operations according to the active replication approach.

Replicated data services can be conveniently built using GCSs [17, 40, 41, 42, 57, 7, 39, 63]. In GCS-based solutions for replication, servers hosting object replicas are organized into a group. When a server receives a request from a client to modify the data object, it disseminates the request to other servers using the group communication primitives. The servers exploit the reliability and ordering properties of the group communication primitives to process the client's request in a way that would maintain mutual consistency of the replicas; one of the servers then sends the response back to the client. Processing of requests is done during normal mode of operation, which corresponds to the periods when the servers interact in stable views. When servers' views change, the servers may switch to state-transfer mode, in which they synchronize their replica states with those of other view members.

In a straightforward setup, the replicated services built using GCSs switch to state-transfer mode whenever the GCSs form new views; e.g., [41, 42, 57, 7, 39, 63]. In this mode, every server multicasts the state of its object replica (and other relevant information, such as pending requests) to the members of the new view. The server then collects a state-transfer message from each member and uses these messages to compute a new, common state for its object replica. Every server that completes the state-transfer protocol gathers the same messages, and thus computes the same new state.

There are two types of overhead associated with state-transfer: First, messages communicated during state-transfer are typically large, as they include the entire object states. Second, normal mode of operation is suspended during state-transfer. In light of this, the straightforward setup, in which state-transfer is performed every time there is a view change and in which every server participates by multicasts its own state-transfer message, is quite costly.

The Virtual Synchrony semantics provided by our GCS allows the application to sometimes avoid state-transfer when views change and also to reduce the number of state messages exchanged during a state-transfer protocol. Recall that the set of group members that transitions together from $\mathtt{v}$ to $\mathtt{v}'$ is known as the transitional set $\mathtt{T}$ of $\mathtt{v}$ and $\mathtt{v}'$. The Virtually Synchronous Delivery property guarantees that every server in $\mathtt{T}$ receives the same set of messages while in view $\mathtt{v}$, before receiving view $\mathtt{v}'$ and set $\mathtt{T}$ from GCS. Thus, if the object replicas of $\mathtt{T}$ were mutually consistent upon entering normal mode in view $\mathtt{v}$, they remain mutually consistent when view $\mathtt{v}'$ is delivered. This leads to two observations: First, it is enough for only one member of $\mathtt{T}$ to communicate the state of its replica during state-transfer protocol. Second, state-transfer is unnecessary in situations when the entire membership of the new view $\mathtt{v}'$ has transitioned together from view $\mathtt{v}$ (i.e., $\mathtt{v.set} = \mathtt{T}$). These two observations lie at the heart of the state-transfer protocol of Amir et al. [5].

Note that the state-transfer protocol of Amir, et al. is only an example of the kinds of optimizations enabled by Virtual Synchrony; others are possible. For example, Bartoli et al., in [14], discuss reconfiguration issues arising in replicated *databases* built using GCSs, and in particular those that implement Virtual Synchrony.

An alternative approach to using Group Communication for building replicated data services is to use Consensus (e.g., [67, 83]). In this approach, during normal mode of operation, the servers hosting object replicas run Consensus to agree on the order in which to process clients' requests. Different replicated services differ in the particular data consistency semantics that they guarantee. For the ones that provide strong consistency semantics (such as, atomicity), the approaches based on Consensus and Group Communication are competitive in their performance characteristics during normal mode of operation. However, for the services that provide weaker consistency semantics, such as the IADS application, the approach based on Group Communication allows for solutions that are significantly more efficient than those that use Consensus. We explain this on page 152 below.

The rest of this chapter is organized as follows: Section 10.1 describes the interface and semantics of the IADS application. Section 10.2 then describes an algorithm for implementing the application. The algorithm operates atop GCS and utilizes the state-transfer protocol of Amir, et al.; the protocol has been precisely stated and carefully verified in the TR version of [5]. We convert the protocol into an I/O automaton, but otherwise leave everything as in [5], including variable names.

For the underlying computational model, we assume an asynchronous system enriched with an *eventually perfect failure detector* ($\diamond P$), as defined by Chockler et al. in [27]. According to [27], an eventually perfect failure detector is a failure detector that, in executions in which a set Ω of clients becomes stable from some point on, eventually stabilizes to the exact set Ω at every process in Ω from some point onward.

We assume this particular model of computation in order keep the definition of the IADS application general. Our algorithm for IADS will use the GCS system, which itself uses an external membership service. We will assume that the membership service is *precise*, i.e., that the membership service delivers a stable view to the members of a stable component (Property 5.2.1). Chockler et al., in [27], prove that an asynchronous system with an eventually perfect failure detector is equivalent to an asynchronous system with a precise group membership service. Thus, in essence, our IADS algorithm will match the assumed model of computation.

Note that, when we defined the GCS system, we did not need to explicitly assume the existence of an eventually perfect failure detector because we made the liveness property of GCS (Property 5.2.2, page 66) conditional on the stability of the underlying MBRSHP service (Property 5.2.1).

The material in this chapter is presented less formally then in the rest of the dissertation. Part of the reason is that the material in this chapter contains mostly well-established results, such as active replication techniques and the state-transfer algorithm of [5]. The other reason is that the material in this chapter is presented simply for the purpose of demonstrating how one would use our GCS system to build effective distributed applications. Going beyond this simple demonstration into the domain of formal specification, verification, and analysis of IADS is outside the scope of this dissertation.

10.1 Application Description

In this section, we describe the interim-atomic data service application, IADS. We define a data type, called `Obj`, for the data object managed by IADS. We then specify the interface of IADS with its clients and the semantics of the provided service.

10.1.1 The `Obj` Data Type

The application manages deterministic data objects whose serial behavior is specified by some data type, `Obj`. The `Obj` data type defines possible states of the objects and operators on the objects; it is defined similarly to the variable type of [71] and the serial data type of [37, 38]. Formally, the `Obj` type consists of:

- a set S of *object states*;
- a distinguished initial state $s_0 \in S$;
- a set R of *response values*; and
- a set O of operations, each of the type $S \rightarrow (S \times R)$.

Furthermore, we assume an application-defined function merge: $\mathtt{SetOf}(\mathtt{Proc} \times S) \rightarrow S$. This function is used during state-transfer to compute a new, common, state of the object based on, possibly different, states of the participating object replicas. We assume that the merge function has the identity property, i.e., $\mathtt{merge}(\{\langle \mathtt{p_1}, \mathtt{x}\rangle, \langle \mathtt{p_2}, \mathtt{x}\rangle, \ldots, \langle \mathtt{p_k}, \mathtt{x}\rangle\}) = \mathtt{x}$.

For simplicity we assume that the application manages a single data object and all the operations requested by clients pertain to this object.

10.1.2 Application Interface

The interface between the IADS application and its clients consists of the typical `request`, and `response` actions: The application receives client `p`'s request to process operation $\mathtt{o} \in O$ via input action $\mathtt{request_p(o)}$, and it eventually responds to the operation via $\mathtt{response_p(o, r)}$, where $\mathtt{r} \in R$ is the return value resulting from applying operation `o` to the underlying data object.

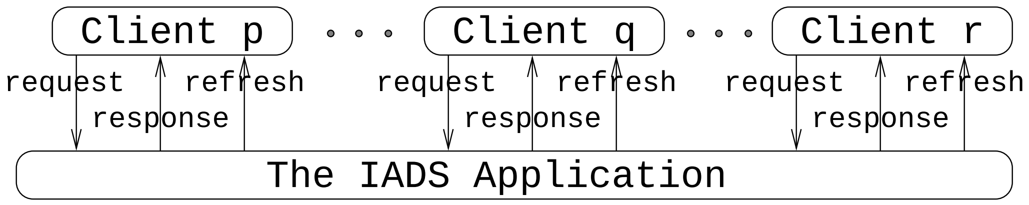


Figure 10.1: Interaction of the application with its clients.

In addition to the `request`/`reply` actions, the interface with client `p` includes special $\texttt{refresh}_\texttt{p}(\texttt{set}, \texttt{x})$ actions, where $\texttt{set} \in \texttt{SetOf}(\texttt{Proc})$ and $\texttt{x} \in S$. The application uses these actions to refresh the client's perception of its collaboration group (`set`) and the state of the underlying data object (`x`). The interface is summarized in Figure 10.2.

IADS **Signature**:

```
Input:                                     Output:
  request_p(o), Proc p, O o                  response_p(o, r), Proc p, O o, R r
                                             refresh_p(s, x), Proc p, SetOf(Proc) s, S x
```

Figure 10.2: Interface specification.

For simplicity, we do not include the `join` and `leave` actions as part of the interface. Such actions can be processed by the group communication service as requests to join or leave a specified application group.

10.1.3 Application Semantics

We now define a collection of properties that specify behavior of IADS. Among them, there are some basic properties that are not specific to interim atomicity; these include properties such as correspondence between requests and responses, and processing of requests submitted by a given client in gap-free FIFO order. The properties that *are* specific to interim atomicity are Stabilization and Interim Atomicity. The Stabilization property is a liveness property that requires IADS to eventually stabilize after a set of clients becomes stable. The Interim Atomicity property is a combination of safety and liveness; it requires IADS to behave as a strongly-consistent data service in situations when IADS is stable.

Correspondence between requests and responses, and FIFO ordering

Our IADS application supports non-blocking interaction with its clients and guarantees processing of requests in FIFO order with no gaps. Figure 10.3 contains an I/O automaton, EO_FIFO_DS, that captures these basic properties.

```
AUTOMATON EO_FIFO_DS

Signature:
  Input:
    request_p(o), Proc p, O o
  Output:
    response_p(o, r), Proc p, O o, R r
    refresh_p(s, x), Proc p, SetOf(Proc) s S x

State:
  (∀ p ∈ Proc) QueueOf(O) ops[p], initially empty

Transitions:
  INPUT request_p(o)
  eff: append o to ops[p]

  OUTPUT response_p(o, r)
  pre: o is first on ops[p]
       let x⁻ be some state in S
       let x in S be s.t. ⟨x, r⟩ = o(x⁻)
  eff: remove first element from ops[p]

  OUTPUT refresh_p(s, x)
  pre: s is some element in SetOf(Proc)
       p ∈ s
       x is some state in S
  eff: none
```

Figure 10.3: Specification EO_FIFO_DS of an abstract data service that supports non-blocking processing of requests in the gap-free FIFO order of their submission.

Every $\texttt{response}_p$ event has a unique corresponding $\texttt{request}_p$ event, and every $\texttt{request}_p$ event has a unique corresponding $\texttt{response}_p$ event provided EO_FIFO_DS keeps delivering responses to p. When client p submits operation requests, they are inserted into a queue, ops[p], and are then processed in the FIFO order defined by the queue. Automaton EO_FIFO_DS does not specify any data consistency semantics: when a request is processed, it is allowed to be applied to an arbitrary object state; see the I/O code for $\texttt{response}_p$. Likewise, the automation does not specify the semantics of $\texttt{refresh}_p$ events; the refresh information is allowed to be arbitrary.

We could define the notion of *reachable states* of an object replica and require the service to use only such states when applying operations and when issuing refresh notifications. Roughly speaking, a reachable state of an object replica for a given IADS state in the execution of can be defined recursively as an application of a sequence of some subset of operations previously submitted by clients to either the initial state of the object or a merge of some set of reachable states of object replicas for some earlier IADS states in the execution.

Note that so far we have not defined any liveness properties that require our application to eventually respond to clients' requests. One such property will be defined below as part of Interim Atomicity: The property will require liveness in situations when a set of clients becomes and remains stable from some point forever on. Looking ahead to Section 10.2, our algorithm for IADS, will also satisfy another, less restrictive, liveness property (Theorem 10.2.5 on page 148): Fair executions of our algorithm would guarantee that every server p eventually responds to every request submitted by its client as long as the server does not stay blocked from some point forever on by the underlying GCS service.

Stabilization

We now define what we mean by a set of clients being stable, and then state the IADS Stabilization property.

Definition 10.1.1 (Stable set of clients) *A set Ω of clients of* IADS *is* stable *if from some point on no new clients join Ω, no client in Ω leave the* IADS *application, and the underlying network component involving the clients in Ω remains stable (i.e., the processes in the component remain mutually connected, and the component does not connect with any other components).*

In situations when a set of clients becomes and remains stable from some point on, the IADS application is required to eventually stabilize to the same set. Recall from page 131 that, in the context of the IADS application, we assume the underlying computational model to be an asynchronous system enriched with an eventually perfect failure detector ($\diamond P$), as defined by Chockler et al. in [27]. According to [27], an eventually perfect failure detector is a failure detector that, in executions in which a set Ω of clients becomes stable from some point on, eventually stabilizes to the exact set Ω at every process in Ω from some point onward. We assume this particular model of computation in order keep the definition of the IADS application general.

Property 10.1.1 (IADS Stabilization) *Let α be an execution of* IADS *in which a set Ω of clients becomes and remains stable from some point on.* IADS *guarantees that, eventually after the set Ω of clients becomes stable, a* $\texttt{refresh}_{\texttt{p}}(\Omega, \textit{base\_state})$ *event occurs at every client* $\texttt{p} \in \Omega$*, and the following two conditions hold:*

- *following the* $\texttt{refresh}_{\texttt{p}}(\Omega, \textit{base\_state})$ *event, no subsequent* $\texttt{refresh}_{\texttt{p}}$ *events occur;*
- *the refreshed state, base_state, is the same for all the clients in Ω.*

Our algorithm for IADS will use the GCS system, which itself uses an external membership service. We will assume that the membership service is *precise*, i.e., that the membership service delivers a stable view to the members of a stable component (Property 5.2.1). Chockler et al., in [27], prove that an asynchronous system with an eventually perfect failure detector is equivalent to an asynchronous system with a precise group membership service. Thus, in essence, our IADS algorithm will match the assumed model of computation.

Note that for non-triviality, we can require *base_state* to be one of the reachable states of $\texttt{p}$'s object replica at the time of the $\texttt{refresh}_{\texttt{p}}(\Omega, \textit{base\_state})$ event (see above).

At the time the final $\texttt{refresh}_{\texttt{p}}$, $\texttt{p} \in \Omega$, event occurs in α, client $\texttt{p}$ possibly has a number of pending operation requests to which IADS has not responded yet; we denote the sequence of these operations as $\textit{init\_ops}(\texttt{p})$.

Interim Atomicity

Figure 10.4 contains an I/O automaton AO$[\Omega, s_b, O_b]$ modeling an atomic object of type `Obj` with the initial state s_b, set of clients Ω, and initial sequence of operations O_b(`p`) from each client `p` $\in \Omega$. Requests from a client `p` are put into a queue `ops`[`p`]. An internal action `do` picks the first operation request from `ops`[`p`] for some client `p` and applies it to the current state of the data object; the operation and the resulting return value are placed into a queue `out`[`p`]; this queue contains responses to be delivered to client `p`. The Tasks component specifies that fair executions of AO have to keep processing requests from each client and have to keep delivering responses to each client (provided the client keeps issuing requests).

```
AUTOMATON AO[$\Omega$, $s_b$, $O_b$], where $\Omega \in$ SetOf(Proc), $s_b \in S$ , $O_b$: $\Omega \rightarrow$Seq($O$)

Signature:
  Input:  request$_p$(o), Proc p $\in \Omega$, $O$ o
  Output: response$_p$(o, r), Proc p $\in \Omega$, $O$ o, $R$ r
  Internal: do(p, o), Proc p $\in \Omega$, $O$ o

State:
  $S$ obj, initially $s_b$
  ($\forall$ p $\in \Omega$) QueueOf($O$) ops[p], initially $O_b$(p)
  ($\forall$ p $\in \Omega$) QueueOf($O \times R$) out[p], initially empty

Transitions:
  INPUT  request$_p$(o)
  eff: append o to ops[p]

  OUTPUT  response$_p$(o, r)
  pre: $\langle$o, r$\rangle$ is first on out[p]
  eff: remove $\langle$o, r$\rangle$ from out[p]

  INTERNAL  do(p, o)
  pre: o = First(ops[p])
       let x and r be s.t. $\langle$x, r$\rangle$ = o(obj)
  eff: remove o from ops[q]
       obj $\leftarrow$ x
       append $\langle$o, r$\rangle$ to out[p]

Tasks: For every p $\in \Omega$:
  C$_{p,1}$ = {response$_p$(o, r) | o $\in O$; r $\in R$ }  and C$_{p,2}$ = {do(p, o) | o $\in O$ }
```

Figure 10.4: Specification AO of an atomic object of type `Obj`, with initial state s_b, set of clients Ω, and initial operations O_b.

Property 10.1.2 (Interim Atomicity) *Let α be an execution sequence in which set Ω of clients becomes and remains stable from some point on. The subsequence of the trace of* IADS *involving clients in Ω following the final* `refresh`$(\Omega, base\_state)$ *events (as defined by Property 10.1.1) is a trace of an atomic object* AO$[\Omega, base_state, init_ops]$.

It is important to note the following: Even though the Interim Atomicity property is defined for the executions in which a network component remains stable forever, the stability condition is external to the application. Thus, the application has to attempt to satisfy this property in every execution, as it can never know whether there is a stable component.

Therefore, whenever a set of clients stabilizes and remains stable for a sufficiently long period of time, an implementation that satisfies Interim Atomicity would deliver the same `refresh`(`set`, `x`) to all the clients in the set and would then behave as an atomic service (while the set of clients remains stable). If instabilities occur, the implementation may cease to provide atomic perception of the data object. This would last until the client's component becomes stable again (and remains stable for a sufficiently long time). Then, the implementation would refresh the client with the new collaboration set and the new state of the object; from that point, while stability lasts, the client would again perceive the object as an atomic one.

Looking ahead to Section 10.2, our algorithm for IADS, will always apply operations in the same order at different object replicas. However, at the times when it does not preserve atomicity, the actual sequences of operations applied to different object replicas may be different.

10.2 GCS-based Algorithm

In this section, we present a distributed algorithm, IADS, for the interim-atomic data service defined in Section 10.1. The algorithm follows the standard active replication (state-machine) approach [66, 83] and utilizes the state-transfer protocol of Amir et al. [5].

The IADS algorithm is composed of a collection of application end-points, which run the same algorithm. The application end-points operate as clients of the GCS system — as members of the same process group.

The application end-points act as servers: each application end-point maintains a replica of the data object. We use the term "application end-point" rather than "server" to highlight the fact that, in our algorithm, servers and clients are tightly coupled: Each client, c_p interacts with the application through the underlying application end-point IADS_p. We assume that the client, its application end-point, and the underlying GCS end-point all operate in the same location p, and therefore, share their fate when network events occur. We will often refer to client c_p and to application end-point IADS_p simply by their location p, as "client p" and "application end-point p".

Figure 10.5 shows interaction of an application end-point with its client and with the underlying GCS end-point. The interaction between the application end-point and its client is consistent with the description of the application interface in Section 10.1. The interaction between the application end-point and the underlying GCS end-point is consistent with the GCS end-point's interface defined in Chapter 5.

10.2.1 Algorithm Description

Every application end-point maintains a replica of the data object. The object replicas are modified during normal mode of operation when clients' requests are processed, and as a result of state-transfer when a new state of the object is computed from the merge of the object replicas of different application end-points.

Figure 10.6 depicts a state-transition diagram that governs transitions between normal and

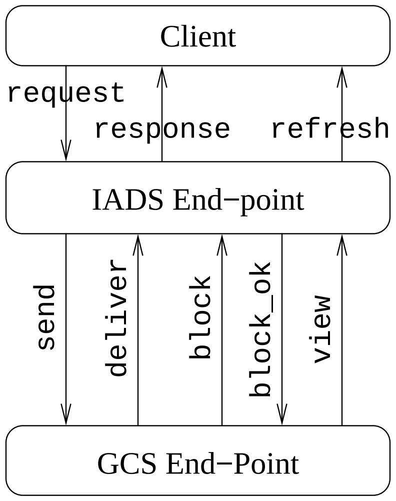


Figure 10.5: Application design architecture: An application end-point interacts with its client and the underlying GCS end-point.

state-transfer modes. Initially, the mode is normal. An application end-point may switch from normal to state-transfer when it receives a new view from GCS; in some situations, discussed below, the application end-point is able to rely on the guarantees provided by GCS to avoid state-transfer and remain in normal mode. When an application end-point completes state-transfer, it switches back to normal mode. If GCS delivers a new view before the application end-point completes state-transfer, the application end-point typically remains in state-transfer mode, but in some situations, discussed below, it may again rely on the guarantees provided by GCS to immediately switch to normal mode.

Figure 10.7 contains an I/O automaton, $\text{IADS}_\texttt{p}$, modeling the application end-point at process `p`. The application end-point maintains a replica, `obj`, of the data object. Requests submitted by the client are placed into a queue, `inp`, and later multicast using GCS to the application end-points comprising the current view. GCS delivers these requests within the same view and in FIFO order. The application end-points append the requests delivered by GCS into a queue, `ops`[`q`], according to the sender `q`. The requests stored in the `ops` queues are processed during normal mode, according to a total order on all requests communicated in a given view; as we explain below, the algorithm establishes this total order by tagging requests with logical timestamps [66] (see also [83] and [71, page 607]).[2] Processing of requests is done by an internal action `do` and as a result of receiving a `view` input from GCS. When an application end-point processes an operation request, it applies the operation to its object replica. If the operation request that is being processed was submitted by the application end-point's own client, the application end-point places the operation and the

[2]Note that we implement total ordering within the application algorithm to make it easier to visualize how the algorithm works. In general, however, total ordering would be implemented as a separate layer, above GCS and below the application; many different algorithms for implementing total-ordering exist, e.g., [26, 25] is a practical algorithm that dynamically adapts message delivery order to the transmission rates of the participating processes.

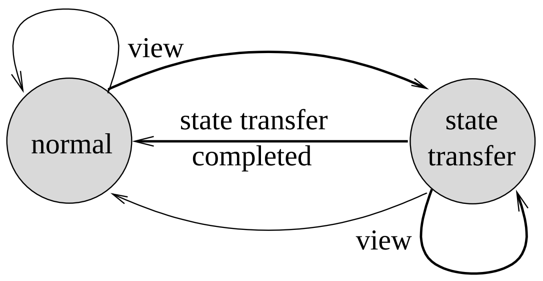


Figure 10.6: Application modes of operation: `view`-labeled transitions leading to `normal` mode correspond to circumstances when an application end-point avoids state-transfer by relying on the GCS semantics.

resulting return value into an output queue, `out`, to be later reported to the client.

Consider application end-points belonging to some stable view. Assume that, at the times when the application end-points start processing requests in the view, the states of their object replicas are the same. In order for their object replicas to stay mutually consistent, the object replicas should undergo the same modifications in the same order at different application end-points; that is, different application end-points should apply the same sequences of operations to their object replicas.

Total order through logical time

The algorithm establishes a total ordering of all requests communicated through GCS in a given view using logical timestamps, as in [66] and [71, Sec. 18.3.3]. Application end-point $\mathtt{p}$ maintains an integer $\mathtt{lt}[\mathtt{p}]$ which corresponds to $\mathtt{p}$'s logical time within the current view; the initial value of $\mathtt{lt}[\mathtt{p}]$ is $\mathtt{0}$ and it is reset to $\mathtt{0}$ when $\mathtt{p}$ receives view inputs. When $\mathtt{p}$ starts processing a request by multicasting it to other application end-points, $\mathtt{p}$ increments $\mathtt{lt}[\mathtt{p}]$ and tags the request with the timestamp. Also, whenever $\mathtt{p}$ receives a request tagged with a timestamp $\mathtt{ts}$ from some application end-point $\mathtt{q}$, $\mathtt{p}$ updates its logical time to $\mathtt{max}(\mathtt{lt}[\mathtt{p}], \mathtt{ts}) + \mathtt{1}$. The total order on messages communicated within the same view is defined as their ordering by the "timestamp, application end-point identifier" pair. That is, $\mathtt{o_p}$ *totally precedes* $\mathtt{o_q}$ if and only if $((\mathtt{ts_p} < \mathtt{ts_q}) \lor ((\mathtt{ts_p} = \mathtt{ts_q}) \land (\mathtt{p} < \mathtt{q}))$, where $\mathtt{o_p}$ is an operation sent by $\mathtt{p}$ and tagged with timestamp $\mathtt{ts_p}$, and $\mathtt{o_q}$ is an operation sent by $\mathtt{q}$ in the same view as $\mathtt{o_p}$ and tagged with timestamp $\mathtt{ts_q}$. Note that this total order is consistent with both FIFO and causal orderings of requests and responses.

Normal mode

In normal mode, application end-point $\mathtt{p}$ processes the operations in its `ops` queues according to the total order defined above. Internal action $\mathtt{do_p}(\mathtt{q}, \mathtt{o})$ models processing of an operation

```
AUTOMATON IADS_p
Type:
AppMsg  = (O × Int) ∪ Int ∪ (SetOf(Proc) × S)      // operations, heartbeats, and state-transfer
OutType = (O × R) ∪ (SetOf(Proc) × S)              // operation replies and refresh information

Signature:
 Input:    request_p(o), O o
           gcs.deliver_p(q, m), Proc q, m ∈ AppMsg
           gcs.view_p(v, T), View v, SetOf(Proc) T
           gcs.block_p()
 Internal: do_p(q, o), Proc q, O o
 Output:   response_p(o, r), O o, R r
           refresh_p(s, x), SetOf(Proc) s, S x
           gcs.send_p(m), m ∈ AppMsg
           gcs.block_ok_p()

State:
S obj, initially s_0
QueueOf(O) inp, initially empty
QueueOf(OutType) out, initially empty
(∀ q ∈ Proc) Int lt[q], initially 0
(∀ q ∈ Proc) QueueOf(O × Int) ops[q], initially empty
block_status ∈ {unblocked, requested, blocked},
               initially unblocked
View myview, initially v_p
Bool mode ∈ {normal, st}
Bool send_state, initially false
SetOf(Proc)  SS, initially {p}
SetOf(Proc × S) StatesV, initially empty
SetOf(Proc) States_Await, initially {}

Transitions:
INPUT  request_p(o)
eff: append o to inp

OUTPUT  gcs.send_p(⟨'op_msg', o, ts⟩)
pre: block_status ≠ blocked
     o = First(inp)  ∧  ts = lt[p] + 1
eff: remove o from inp
     lt[p] ← lt[p]+1
     append ⟨o, ts⟩ to ops[p]

INPUT  gcs.deliver_p(q, ⟨'op_msg', o, ts⟩)
eff: if (q ≠ p) then
        lt[q] ← ts
        lt[p] ← max(lt[p], ts) + 1
        append ⟨o, ts⟩ to ops[q]

INTERNAL  do_p(q, o)
pre: mode = normal
     q = min{t ∈ myview.set : First(ops[t]).ts =
         = (min{First(ops[r]).ts : r ∈ myview.set})}
     (∀ t ∈ myview.set) lt[t] > First(ops[q]).ts
     ⟨o, ts⟩ = First(ops[q])
     let x and r be s.t. ⟨x, r⟩ =  o(obj)
eff: remove ⟨o, ts⟩ from ops[q]
     obj ← x
     if(p = q) then append ⟨o, r⟩ to out

OUTPUT  response_p(o, r)
pre: ⟨o, r⟩ is first on out
eff: remove ⟨o, r⟩ from out

OUTPUT  refresh_p(set, x)
pre: ⟨set, x⟩ is first on out
eff: remove ⟨set, x⟩ from out

OUTPUT  gcs.send_p(⟨'lt_msg', ts⟩)
pre: block_status ≠ blocked  ∧  ts = lt[p]

INPUT  gcs.deliver_p(q, ⟨'lt_msg', ts⟩)
eff: if (q ≠ p) then
        lt[q] ← ts
        lt[p] ← max(lt[p], ts) + 1

INPUT  gcs.block_p()
eff: block_status ← requested

OUTPUT  gcs.block_ok_p()
pre: block_status = requested
eff: block_status ← blocked

INPUT  gcs.view_p(v, T)
eff: // process all operations in the ops queue
     while (∃ q) such that
     (q = min{t ∈ myview.set : First(ops[t]).ts =
         = (min{First(ops[r]).ts : r ∈ myview.set})})
        remove first element ⟨o, ts⟩ from ops[q]
        ⟨obj, r⟩ ←  o(obj)
        if(p = q) then append ⟨o, r⟩ to out
     end
     (∀ t ∈ myview.set) lt[t] ← 0
     myview ← v
     block_status ← unblocked
     // state transfer decision:
     (mode = normal ? SS ← T : SS ← SS ∩ T)
     if(v.set = SS) then  // normal mode
         append ⟨v.set, obj⟩ to out
         mode ← normal
     else                 // state-transfer
         State_Await ← v.set; StatesV ← empty
         send_state ← (p = min(SS))
         mode ← st

|---------------| state transfer |---------------|

OUTPUT  gcs.send_p(⟨'st_msg', set, x⟩)
pre: block_status = unblocked
     send_state = true  ∧ ⟨set, x⟩ = ⟨SS, obj⟩
eff: send_state ← false

INPUT  gcs.deliver_p(q, ⟨'st_msg', set, x⟩)
eff: (∀ t ∈ set) add ⟨t, x⟩ to StatesV
     States_Await ← States_Await - set
     if (States_Await = {}) then
       obj ←  merge(StatesV)
       append ⟨myview.set, obj⟩ to out
       mode ← normal
```

Figure 10.7: Application end-point IADS$_p$ of an Interim-Atomic Data Service.

o submitted by client at q. Operation o is processed if

(a) operation o totally precedes all other operations currently in the ops queues; and

(b) p knows that the logical times of all other application end-points in the view exceed the timestamp associated with o.

Condition (b) guarantees that p has received all the operations that precede o, and thus, o is the next one in the totally ordered sequence of operations sent in the current view; see [71, Sec. 18.3.3]. The algorithm implements condition (b) by keeping track of the known logical time of every application end-point q in the current view. The application end-point updates lt[q] whenever it receives operation requests sent by q. In addition to communicating the values of their logical times through operations, application end-points also let others know of their logical times by periodically sending special heartbeat messages, ⟨'**lt_msg**', ts⟩.

While the current view remains stable, the application end-points process the same sequences of operations, and thus, remain mutually consistent.

When an application end-point receives a new view from GCS, the application end-point processes all of the operations in its ops queues according to the total order, even though condition (b) may not hold for these messages. The sequence of operations processed at this point may diverge from the global sequence because the application end-point may have received only a subset of all of the operations sent in the current view; for example, it may be missing some of the operations sent by disconnected application end-points. However, what is guaranteed by GCS is that members of the transitional set of the new view receive the same set of messages, and hence process the same sequence, if they receive the new view. Thus, after processing the operations in their ops queues, the members of the transitional set have the same states of their object replicas.

Deciding whether to enter state-transfer

After the operations are processed, application end-point p decides whether or not to enter the state-transfer protocol. The decision algorithm is taken from [5]. We describe it here.

Variable SS is used for keeping track of the set of application end-points whose object replicas are synchronized with p. If SS is the same as the membership of a new view v, then everyone in the new view is already synchronized and p does not need to participate in a state-transfer protocol for view v; it may resume its normal mode of operation. Otherwise, p enters the state-transfer protocol, which we discuss below.

According to [5], variable SS is computed as the intersection of all the transitional sets delivered since normal mode: When an application end-point p receives $\mathtt{gcs.view}_\mathtt{p}(\mathtt{v}, \mathtt{T})$ while in a normal mode, set SS is initialized to be the transitional set T. If p receives subsequent views while in state-transfer mode, set SS is intersected with the new transitional sets.

Indeed, consider an application end-point p in view $\mathtt{v}^-$ that receives $\mathtt{gcs.view}_\mathtt{p}(\mathtt{v}, \mathtt{T})$. Assume that p's mode is normal prior to receiving the new view. If the membership v.set of the new view is the same as transitional set $\mathtt{T}$, then all of the members of $\mathtt{v}$ enter the new view directly from $\mathtt{v}^-$ (provided they do enter $\mathtt{v}$). The Virtual Synchrony semantics guarantees that these members have received the same sets of messages while in view $\mathtt{v}^-$, and hence have applied the same operations in the same order to their object replicas. Since the states of the object replicas of the members of $\mathtt{T}$ were the same when they began normal mode in view $\mathtt{v}^-$, their object replicas are the same after receiving view $\mathtt{v}$ from GCS.

Now consider a situation in which an application end-point $\mathtt{p}$ receives $\mathtt{gcs.view}_\mathtt{p}(\mathtt{v}', \mathtt{T}')$ while engaged in state-transfer in view $\mathtt{v}$. Even though all the application end-points may be transitioning together from $\mathtt{v}$ to $\mathtt{v}'$, it may be the case that these application end-points had inconsistent object replicas prior to entering view $\mathtt{v}$. Since the state-transfer protocol was interrupted, they did not have a chance to synchronize their object replicas. Thus, it is not sufficient to simply consider transitional set $\mathtt{T}$. The intersection of $\mathtt{SS}$ and $\mathtt{T}$ gives a set of application end-points that a) were synchronized when they switched from normal mode to state-transfer, and b) have been synchronized since.

Hence, if the membership of the new view $\mathtt{v}$ is the same as set $\mathtt{SS}$, the application end-point places the refresh information onto the `out` queue and continues normal mode of operation. The refresh information contains the new membership set and the current object state.[3] Otherwise, if $\mathtt{v.set} \neq \mathtt{SS}$, application end-point $\mathtt{p}$ enters the state-transfer protocol.

State-transfer protocol

The state-transfer protocol involves each application end-point collecting the states of the object replicas of the members of the new view, and then computing a new state for its replica as a merge of the collected states. After the object replica is updated with the result of the merge, the refresh information is placed on the `out` queue. The refresh information contains the new membership set and the new state of the object.

The part of the code in Figure 10.7 that implements state-transfer follows the algorithm of [5]; we use the same names for variables, except for variable $\mathtt{SS}$, which is named $\mathtt{S}$ in [5]. As was explained above, variable $\mathtt{SS}$ defines the set of application end-points whose object replicas are synchronized; it is computed as the intersection of all the transitional sets delivered since normal mode.

The GCS semantics allows us to reduce the number of messages and the amount of information communicated during the protocol: Only one application end-point among the members of $\mathtt{SS}$ needs to send the state of its object replica to others [5]. This is because our algorithm maintains a property that after receiving a view, all members of $\mathtt{SS}$ have their object replicas in the same state. The optimization is important because state-transfer messages are typically “costly” due to their large size.

Boolean variable `send_state` controls whether application end-point $\mathtt{p}$ has to send its object replica's state on behalf of the application end-points in set $\mathtt{SS}$. As in [5], if $\mathtt{p}$ is the smallest one in $\mathtt{SS}$, it sets its `send_state` to `true` and enables $\mathtt{gcs.send}_\mathtt{p}(\langle \text{'}\mathbf{st\_msg}\text{'}, \mathtt{SS}, \mathtt{obj}_\mathtt{p}\rangle)$, which

[3]At this point, the application could have also informed the client that the object state has not been modified.

multicasts $\mathtt{obj_p}$ on behalf of application end-points in SS to the members of v.set.[4]

Set StatesV is used for collecting object replicas' states of the view members, and set States_Await is used for keeping track of the list of application end-points from whom p has not yet received a state-transfer message. If p decides to participate in state-transfer when $\mathtt{gcs.view_p(v, T)}$ occurs, States_Await is initialized to v.set, and StatesV is emptied. Whenever application end-point p receives a state x on behalf of set of application end-points set, p adds $\langle \mathtt{t}, \mathtt{x} \rangle$ to StatesV for every t in set, and removes members of set from States_Await.

If States_Await becomes empty, implying that all the states have been collected in StatesV, application end-point p computes a new object state by applying the merge function to StatesV. The application end-point then updates the state of its object replica with the result of the merge, and places the refresh information onto the out queue. The refresh information contains the current view's membership and the new object state.

Note that application end-points keep multicasting their clients' operations to one another in the new view, in parallel with the state-transfer protocol. The only part of the algorithm that is blocked during state-transfer is the actual processing of the operations. When state-transfer completes, the application end-point may be able to process a whole bunch of operations collected in the ops queues right away, by executing a sequence of do actions.

If the state-transfer protocol is interrupted by a delivery of a new view, the application end-point, as before, processes all of the operations in its ops queues according to the total order, and then decides whether to re-start a state-transfer protocol or to switch back to normal mode.

Tasks

Figure 10.7 does not specify the tasks partition of the application end-point automaton. We define the following tasks:

- $\mathtt{C_{p,1}} = \{\mathtt{response_p}, \mathtt{refresh_p}\}$;
- $\mathtt{C_{p,2}} = \{\mathtt{gcs.send_p}(\langle\text{‘}\textbf{op\_msg}\text{’}, \mathtt{o}, \mathtt{ts}\rangle) \ : \ \mathtt{o} \in \mathtt{O} \ \wedge \ \mathtt{ts} \in \mathtt{Int}\}$;
- $\mathtt{C_{p,3}} = \{\mathtt{gcs.send_p}(\langle\text{‘}\textbf{lt\_msg}\text{’}, \mathtt{ts}\rangle) \ : \ \mathtt{ts} \in \mathtt{Int}\}$;
- $\mathtt{C_{p,4}} = \{\mathtt{gcs.send_p}(\langle\text{‘}\textbf{st\_msg}\text{’}, \mathtt{set}, \mathtt{x}\rangle) \ : \ \mathtt{set} \in \mathtt{SetOf(Proc)} \ \wedge \ \mathtt{x} \in \mathtt{S}\}$;
- $\mathtt{C_{p,5}} = \{\mathtt{gcs.block\_ok_p}\}$;
- $\mathtt{C_{p,6}} = \{\mathtt{do_p}\}$.

[4]Notice that the state-transfer message does not need to be sent to the entire view, but only to the members of $\mathtt{v.set} - \mathtt{SS}$. As is, our GCS system supports sending messages to the entire view memberships only. However, in general, it is straightforward to extend GCS to support group multicasts to subsets of view memberships. The underlying communication service CO_RFIFO already provides this functionality.

This means that, in a fair execution, application end-point p must keep responding to its client, keep multicasting its client's requests to other application end-points using GCS, keep informing other application end-points of its logical time, and keep participating in state-transfer protocols, as well as keep processing requests.

10.2.2 Correctness

In this subsection we prove that the composition of all the application end-point automata IADS_p and the GCS system satisfies the specification of IADS given in Section 10.1. We first establish key properties of our algorithm, and then use them to derive the Stabilization and Interim Atomicity properties; showing the one-to-one correspondence between requests and responses and the FIFO properties is straightforward.

Key properties of the algorithm

The properties and their proofs are similar to the correctness proof in the TR version of [5].

First, we define the following helpful notation:

Notation 10.2.1 *Given an event* $\text{GCS.view}_p(v, T)$ *in an execution of* IADS*,*

- pre_p^v *and* post_p^v *are respectively the pre-state and post-state of* IADS *when* $\text{GCS.view}_p(v, T)$ *occurs.*
- $T_p^v = T$.
- $SS_p^v = \text{post}_p^v[p].SS$*, that is the value of set* SS *that* p *has after it receives view* v*. According to the code,* $SS_p^v = T_p^v$ *if* $\text{pre}_p^v[p].\text{mode} = \text{normal}$*;*

  $SS_p^v = SS_p^{v_p^-} \cap T_p^v$ *if* $\text{pre}_p^v[p].\text{mode} = \text{st}$*, where* v_p^- *is* p*'s view prior to* v*.*
- $\text{obj}_p^v = \text{post}_p^v[p].\text{obj}$*, that is, the state of object replica* obj *that* p *has after it receives view* v *(i.e., after it applies all the operations communicated in the view* v_p^- *that precedes* v*).*
- base_obj_p^v *is defined if* p *ever operates in* normal *mode while in* v *and denotes the state of* p*'s object replica at the time* p *starts* normal *mode of operation in* v*.*

  base_obj_p^v *is equal to* obj_p^v *if* p *does not participate in state-transfer. Otherwise,* base_obj_p^v *is the state of* p*'s object replica at the time* p *completes state-transfer protocol in* v *and switches to normal mode of operation.*

From the fact that GCS satisfies the Transitional Set property (Property 5.1.1, page 64), it follows that, if two application end-points p and q receive the same view v and if q is in p's transitional set for view v, then p and q transition into v from the same view v^- and their transitional sets for view v are the same.

Property 10.2.1 *If both* $\mathtt{p}$ *and* $\mathtt{q}$ *receive view* $\mathtt{v}$*, and* $\mathtt{q} \in \mathtt{T}_\mathtt{p}^\mathtt{v}$*, then* $\mathtt{v}_\mathtt{q}^- = \mathtt{v}_\mathtt{p}^-$ *and* $\mathtt{T}_\mathtt{q}^\mathtt{v} = \mathtt{T}_\mathtt{p}^\mathtt{v}$*.*

The following Lemma states that if two application end-points $\mathtt{p}$ and $\mathtt{q}$ initiate state-transfer while in view $\mathtt{v}^-$ and then later transition together from $\mathtt{v}^-$ into view $\mathtt{v}$, then they either both complete the state-transfer protocol while in $\mathtt{v}^-$ or both do not.

Lemma 10.2.1 *If both* $\mathtt{p}$ *and* $\mathtt{q}$ *receive view* $\mathtt{v}$*, and* $\mathtt{q} \in \mathtt{T}_\mathtt{p}^\mathtt{v}$*, then if* $\mathtt{post}_\mathtt{p}^{\mathtt{v}^-}[\mathtt{p}].\mathtt{mode} = \mathtt{post}_\mathtt{q}^{\mathtt{v}^-}[\mathtt{q}].\mathtt{mode} = \mathtt{st}$*, then* $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{pre}_\mathtt{q}^\mathtt{v}[\mathtt{q}].\mathtt{mode}$*, where* $\mathtt{v}^-$ *denotes* $\mathtt{v}_\mathtt{p}^- = \mathtt{v}_\mathtt{q}^-$ *(Property 10.2.1).*

Proof : Upon receiving view $\mathtt{v}^-$, both $\mathtt{p}$ and $\mathtt{q}$ set their $\mathtt{State\_Await}$ variables to $\mathtt{v}^-.\mathtt{set}$. Because GCS satisfies Virtually Synchronous Delivery and because $\mathtt{p}$ and $\mathtt{q}$ transition together from view $\mathtt{v}^-$ to view $\mathtt{v}$, $\mathtt{p}$ and $\mathtt{q}$ receive the same sets of messages while in view $\mathtt{v}^-$. Therefore, either they both receive all the state-transfer messages and switch to $\mathtt{normal}$ mode while in view $\mathtt{v}^-$, or they both do not. ■

Correctness of the state-transfer protocol relies heavily on the soundness of the $\mathtt{SS}$ sets. The next two lemmas and the corollary that follows establish soundness of the $\mathtt{SS}$ sets and the key properties of the algorithm that this soundness implies.

Lemma 10.2.2 *If both* $\mathtt{p}$ *and* $\mathtt{q}$ *receive view* $\mathtt{v}$*, and* $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^\mathtt{v}$*, then*

1. $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{pre}_\mathtt{q}^\mathtt{v}[\mathtt{q}].\mathtt{mode}$, and
2. $\mathtt{SS}_\mathtt{q}^\mathtt{v} = \mathtt{SS}_\mathtt{p}^\mathtt{v}$.

Proof : End-point $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^\mathtt{v}$ implies $\mathtt{q} \in \mathtt{T}_\mathtt{p}^\mathtt{v}$, and hence $\mathtt{T}_\mathtt{q}^\mathtt{v} = \mathtt{T}_\mathtt{p}^\mathtt{v}$ and $\mathtt{v}_\mathtt{p}^- = \mathtt{v}_\mathtt{q}^-$ (Property 10.2.1); let $\mathtt{v}^-$ denote $\mathtt{v}_\mathtt{p}^- = \mathtt{v}_\mathtt{q}^-$. We now consider the cases of $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{normal}$ and $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{st}$ separately:

- $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{normal}$:

    1. There are two cases:

       $\mathtt{post}_\mathtt{p}^{\mathtt{v}^-}[\mathtt{p}].\mathtt{mode} = \mathtt{normal}$: From the IADS code we have, $\mathtt{SS}_\mathtt{p}^{\mathtt{v}^-} = \mathtt{v}^-.\mathtt{set}$. Since $\mathtt{q} \in \mathtt{v}^-.\mathtt{set}$, it follows that $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^{\mathtt{v}^-}$. Hence, inductively, $\mathtt{SS}_\mathtt{q}^{\mathtt{v}^-} = \mathtt{SS}_\mathtt{p}^{\mathtt{v}^-}$. Thus, $\mathtt{post}_\mathtt{q}^{\mathtt{v}^-}[\mathtt{q}].\mathtt{mode} = \mathtt{normal}$, which means that $\mathtt{pre}_\mathtt{q}^\mathtt{v}[\mathtt{q}].\mathtt{mode} = \mathtt{normal}$.

       $\mathtt{post}_\mathtt{p}^{\mathtt{v}^-}[\mathtt{p}].\mathtt{mode} = \mathtt{st}$: If $\mathtt{post}_\mathtt{q}^{\mathtt{v}^-}[\mathtt{q}].\mathtt{mode} = \mathtt{st}$, then by Lemma 10.2.1 $\mathtt{pre}_\mathtt{q}^\mathtt{v}[\mathtt{q}].\mathtt{mode} = \mathtt{normal}$. The case of $\mathtt{post}_\mathtt{q}^{\mathtt{v}^-}[\mathtt{q}].\mathtt{mode} = \mathtt{normal}$ is not possible, but even if it were, then $\mathtt{pre}_\mathtt{q}^\mathtt{v}[\mathtt{q}].\mathtt{mode}$ would be $\mathtt{normal}$.

    2. From the code: $\mathtt{SS}_\mathtt{p}^\mathtt{v} = \mathtt{T}_\mathtt{p}^\mathtt{v}$. By Property 10.2.1), $\mathtt{T}_\mathtt{p}^\mathtt{v} = \mathtt{T}_\mathtt{q}^\mathtt{v}$. By part 1, $\mathtt{T}_\mathtt{q}^\mathtt{v} = \mathtt{SS}_\mathtt{q}^\mathtt{v}$. Thus, $\mathtt{SS}_\mathtt{p}^\mathtt{v} = \mathtt{SS}_\mathtt{q}^\mathtt{v}$.

- $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{st}$:

    1. We have $\mathtt{post}_\mathtt{p}^{\mathtt{v}^-}[\mathtt{p}].\mathtt{mode} = \mathtt{st}$, and then by Lemma 10.2.1 $\mathtt{pre}_\mathtt{q}^\mathtt{v}[\mathtt{q}].\mathtt{mode} = \mathtt{st}$.

2. From the IADS code, we have $\mathtt{SS}_\mathtt{p}^\mathtt{v} = \mathtt{SS}_\mathtt{p}^{\mathtt{v}^-} \cap \mathtt{T}_\mathtt{p}^\mathtt{v}$. Since, $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^\mathtt{v}$, it follows that $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^{\mathtt{v}^-}$. Hence, inductively, $\mathtt{SS}_\mathtt{p}^{\mathtt{v}^-} = \mathtt{SS}_\mathtt{q}^{\mathtt{v}^-}$. By part 1, $\mathtt{pre}_\mathtt{q}^\mathtt{v}[\mathtt{q}].\mathtt{mode} = \mathtt{st}$, which means that $\mathtt{SS}_\mathtt{q}^\mathtt{v} = \mathtt{SS}_\mathtt{q}^{\mathtt{v}^-} \cap \mathtt{T}_\mathtt{q}^\mathtt{v}$; and, by Property 10.2.1, $\mathtt{T}_\mathtt{q}^\mathtt{v} = \mathtt{T}_\mathtt{p}^\mathtt{v}$. Thus, $\mathtt{SS}_\mathtt{q}^\mathtt{v} = \mathtt{SS}_\mathtt{q}^{\mathtt{v}^-} \cap \mathtt{T}_\mathtt{q}^\mathtt{v} = \mathtt{SS}_\mathtt{p}^{\mathtt{v}^-} \cap \mathtt{T}_\mathtt{p}^\mathtt{v} = \mathtt{SS}_\mathtt{p}^\mathtt{v}$. ■

Lemma 10.2.3 *If both* p *and* q *receive view* v*, and* $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^\mathtt{v}$*, then* $\mathtt{obj}_\mathtt{q}^\mathtt{v} = \mathtt{obj}_\mathtt{p}^\mathtt{v}$.

Proof : End-point $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^\mathtt{v}$ implies $\mathtt{q} \in \mathtt{T}_\mathtt{p}^\mathtt{v}$, and hence $\mathtt{T}_\mathtt{q}^\mathtt{v} = \mathtt{T}_\mathtt{p}^\mathtt{v}$ and $\mathtt{v}_\mathtt{p}^- = \mathtt{v}_\mathtt{q}^-$ (Property 10.2.1); let $\mathtt{v}^-$ denote $\mathtt{v}_\mathtt{p}^- = \mathtt{v}_\mathtt{q}^-$. We now consider the cases of $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{normal}$ and $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{st}$ separately:

- $\mathtt{pre}_\mathtt{p}^\mathtt{v}[\mathtt{p}].\mathtt{mode} = \mathtt{normal}$**:** There are two cases:

  $\mathtt{post}_\mathtt{p}^{\mathtt{v}^-}[\mathtt{p}].\mathtt{mode} = \mathtt{normal}$**:** From the IADS code we have, $\mathtt{SS}_\mathtt{p}^{\mathtt{v}^-} = \mathtt{v}^-.\mathtt{set}$. Since $\mathtt{q} \in \mathtt{v}^-.\mathtt{set}$, it follows that $\mathtt{q} \in \mathtt{SS}_\mathtt{p}^{\mathtt{v}^-}$. Hence, inductively, $\mathtt{obj}_\mathtt{q}^{\mathtt{v}^-} = \mathtt{obj}_\mathtt{p}^{\mathtt{v}^-}$, that is the object replicas of p and q have the same state respectively after p and q receive view $\mathtt{v}^-$.

  We now argue that p and q apply the same set of operations in the same order to their object replicas after they receive view $\mathtt{v}^-$. Consider the following three facts: a) Because p and q transition together from view $\mathtt{v}^-$ to view v, p and q receive the same sets of messages while in view $\mathtt{v}^-$ (Virtually Synchronous Deliver), and in particular the same sets of operation requests. b) p and q apply the operation received in view $\mathtt{v}^-$ according to the total order defined by logical timestamps. c) p and q process all of the operations that they receive from GCS in view $\mathtt{v}^-$. Hence, application end-points p and q apply the same set of operations in the same order to their object replicas.

  Since the replicas are initially in the same states $\mathtt{obj}_\mathtt{q}^{\mathtt{v}^-} = \mathtt{obj}_\mathtt{p}^{\mathtt{v}^-}$, the resulting states are also the same: $\mathtt{obj}_\mathtt{q}^\mathtt{v} = \mathtt{obj}_\mathtt{p}^\mathtt{v}$.

  $\mathtt{post}_\mathtt{p}^{\mathtt{v}^-}[\mathtt{p}].\mathtt{mode} = \mathtt{st}$**:** First, we argue that $\mathtt{post}_\mathtt{q}^{\mathtt{v}^-}[\mathtt{q}].\mathtt{mode} = \mathtt{st}$. Assume otherwise: $\mathtt{post}_\mathtt{q}^{\mathtt{v}^-}[\mathtt{q}].\mathtt{mode} = \mathtt{normal}$. Then, it follows from the code that $\mathtt{SS}_\mathtt{q}^{\mathtt{v}^-} = \mathtt{v}^-.\mathtt{set}$, and thus $\mathtt{p} \in \mathtt{SS}_\mathtt{q}^{\mathtt{v}^-}$. By Lemma 10.2.2, $\mathtt{SS}_\mathtt{q}^{\mathtt{v}^-} = \mathtt{SS}_\mathtt{p}^{\mathtt{v}^-}$, which implies that $\mathtt{post}_\mathtt{p}^{\mathtt{v}^-}[\mathtt{p}].\mathtt{mode}$ should be $\mathtt{normal}$, not st — a contradiction. Thus, $\mathtt{post}_\mathtt{q}^{\mathtt{v}^-}[\mathtt{q}].\mathtt{mode} = \mathtt{st}$; so both p and q enter the state-transfer protocol in view $\mathtt{v}^-$, and both of them complete it (Lemma 10.2.1).

  Upon receiving view $\mathtt{v}^-$, both p and q set their `State_Await` variables to $\mathtt{v}^-.\mathtt{set}$ and their `StatesV` variable to `empty`. End-points p and q receive the same sets of messages while in view $\mathtt{v}^-$ (Virtually Synchronous Deliver).

  In particular, they receive the same sets of state-transfer messages and, therefore, apply function `merge` to the same `StatesV` vector. Thus, the states of their object replicas are the same after each of them finishes the state-transfer protocol in view $\mathtt{v}^-$: $\mathtt{base\_obj}_\mathtt{p}^{\mathtt{v}^-} = \mathtt{base\_obj}_\mathtt{q}^{\mathtt{v}^-}$.

  Just as in the previous case, we can argue that p and q apply the same sets of operations in the same order to their object replicas in view $\mathtt{v}^-$. Thus, $\mathtt{obj}_\mathtt{q}^\mathtt{v} = \mathtt{obj}_\mathtt{p}^\mathtt{v}$.

- $\mathtt{pre}^{\mathtt{v}}_{\mathtt{p}}[\mathtt{p}].\mathtt{mode} = \mathtt{st}$**:** From the IADS code, we have $\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{SS}^{\mathtt{v}^-}_{\mathtt{p}} \cap \mathtt{T}^{\mathtt{v}}_{\mathtt{p}}$. Since, $\mathtt{q} \in \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}}$, it follows that $\mathtt{q} \in \mathtt{SS}^{\mathtt{v}^-}_{\mathtt{p}}$. Hence, inductively, $\mathtt{obj}^{\mathtt{v}^-}_{\mathtt{p}} = \mathtt{obj}^{\mathtt{v}^-}_{\mathtt{q}}$.

  As before, we can argue that, since p and q receive and the same sets of operation requests while in view $\mathtt{v}^-$ and since p and q process all of these requests in the same order. Hence, $\mathtt{obj}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{obj}^{\mathtt{v}}_{\mathtt{q}}$. ■

Corollary 10.2.1 *Given a view* v*, if every member* $\mathtt{p} \in \mathtt{v}$ *receives* v*, then*

1. $\cup_{\mathtt{p}\in\mathtt{v.set}} \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{v.set}$;
2. $(\forall\ \mathtt{p} \in \mathtt{v.set})\ (\forall\ \mathtt{q} \in \mathtt{v.set})\ (\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} \cap \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}} \neq 0 \Rightarrow \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}})$;
3. $(\forall\ \mathtt{p} \in \mathtt{v.set})\ (\forall\ \mathtt{q} \in \mathtt{v.set})\ (\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}} \Rightarrow \mathtt{obj}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{obj}^{\mathtt{v}}_{\mathtt{q}})$;

Proof :

1. First, observe that for any $\mathtt{p} \in \mathtt{v.set}$, $\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} \subseteq \mathtt{v.set}$ since $\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} \subseteq \mathtt{T}^{\mathtt{v}}_{\mathtt{p}} \subseteq \mathtt{v.set}$. Thus, $\cup_{\mathtt{p}\in\mathtt{v.set}} \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} \subseteq \mathtt{v.set}$. Second, since every transitional set that p receives includes p itself, $\mathtt{p} \in \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}}$. This means that, for every $\mathtt{p} \in \mathtt{v.set}$, $\mathtt{p} \in \cup_{\mathtt{p}\in\mathtt{v.set}} \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}}$. Thus, $\cup_{\mathtt{p}\in\mathtt{v.set}} \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} \supseteq \mathtt{v.set}$.
2. If $\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} \cap \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}} \neq 0$, then there exists some $\mathtt{t} \in \mathtt{v.set}$ such that $\mathtt{t} \in \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}}$ and $\mathtt{t} \in \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}}$. By Lemma 10.2.2, it follows that $\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{SS}^{\mathtt{v}}_{\mathtt{t}} = \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}}$.
3. Since every transitional set that p receives includes p itself, $\mathtt{p} \in \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}}$. From $\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}}$, it follows that $\mathtt{p} \in \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}}$ and, by Lemma 10.2.3, $\mathtt{obj}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{obj}^{\mathtt{v}}_{\mathtt{q}}$. ■

The following lemma establishes that, when application end-points complete the state-transfer protocol in some view, the new object states that the application end-points compute are the merge of the states that the members had when they entered the view.

Lemma 10.2.4 *Given a view* v *such that all* $\mathtt{q} \in \mathtt{v.set}$ *receive view* v*, if an end-point* p *ever operates in* `normal` *mode while in view* v*, then* $\mathtt{base\_obj}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{merge}(\{\langle \mathtt{q}, \mathtt{obj}^{\mathtt{v}}_{\mathtt{q}}\rangle\ :\ \mathtt{q} \in \mathtt{v.set}\})$.

Proof : First, consider the case when p avoids state-transfer in v, that is, when $\mathtt{SS}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{v.set}$. By Corollary 10.2.1, for every $\mathtt{q} \in \mathtt{v.set}$, $\mathtt{SS}^{\mathtt{v}}_{\mathtt{q}} = \mathtt{SS}^{\mathtt{v}}_{\mathtt{p}}$ (part 2), and $\mathtt{obj}^{\mathtt{v}}_{\mathtt{q}} = \mathtt{obj}^{\mathtt{v}}_{\mathtt{p}}$ (part 3). Thus, $\mathtt{merge}(\{\langle \mathtt{q}, \mathtt{obj}^{\mathtt{v}}_{\mathtt{q}}\rangle\ :\ \mathtt{q} \in \mathtt{v.set}\}) = \mathtt{merge}(\{\langle \mathtt{q}, \mathtt{obj}^{\mathtt{v}}_{\mathtt{p}}\rangle\ :\ \mathtt{q} \in \mathtt{v.set}\})$, which by the identity property of the `merge` function is equal to $\mathtt{obj}^{\mathtt{v}}_{\mathtt{p}}$. $\mathtt{obj}^{\mathtt{v}}_{\mathtt{p}} = \mathtt{base\_obj}^{\mathtt{v}}_{\mathtt{p}}$ by definition of $\mathtt{base\_obj}^{\mathtt{v}}_{\mathtt{p}}$ and the fact that p avoids state-transfer in v.

Now, consider the case when p does participate in state-transfer in v. In order for p to switch to `normal` mode, its `StatesV` set must contain state entries for every member $\mathtt{q} \in \mathtt{v.set}$; we must prove that these entries are correct, i.e., are indeed $\mathtt{obj}^{\mathtt{v}}_{\mathtt{q}}$. An application end-point t multicasts a state-transfer message containing $\mathtt{obj}^{\mathtt{v}}_{\mathtt{t}}$ on behalf of q only if $\mathtt{q} \in \mathtt{SS}^{\mathtt{v}}_{\mathtt{t}}$. But then, by Corollary 10.2.1, $\mathtt{SS}^{\mathtt{v}}_{\mathtt{t}} = \mathtt{SS}^{\mathtt{v}}_{\mathtt{q}}$ (part 2), and $\mathtt{obj}^{\mathtt{v}}_{\mathtt{t}} = \mathtt{obj}^{\mathtt{v}}_{\mathtt{q}}$ (part 3). ■

Correspondence between requests and responses and FIFO ordering

It is straightforward to prove that clients receive no spurious responses, that is, that for every response there is a unique, corresponding request. The fact that, for a given client, the order of responses is the same as the order of requests is implied by the use of FIFO queues in the algorithm.

In fact, we can define a refinement $\mathtt{RM_{fifo}}$ between IADS and EO_FIFO_DS: $\mathtt{RM_{fifo}}$ maps a state s of IADS into a state of EO_FIFO_DS in which ops[p] is equal to the concatenation of s[p].inp, s[p].ops[p].o, and s[p].out[p].o. The proof of $\mathtt{RM_{fifo}}$ being a refinement is straightforward.

We can also prove that, under certain conditions, IADS keeps processing requests. There are two conditions: First, is fairness of the execution, i.e., that IADS keeps taking locally-enabled steps. Second, is that GCS does not block a given application end-point from some point forever on. (Note that we do not need to assume stabilization of the underlying network components.)

Theorem 10.2.5 (Eventual Response) *In a fair execution of* IADS *in which an application end-point* p *is not blocked by* GCS *from some point forever on, every* $\mathtt{request_p}$ *event has a unique, corresponding* $\mathtt{response_p}$ *event.*

Proof : Consider a fair execution of IADS and application end-point p. Provided application end-point p is not blocked forever from some point on, p eventually places every request received from its client into queue ops[p] at the time it multicasts the request using GCS. There are two cases: If p's view does not change, this means that p's view is stable (i.e., the members are connected and are able to communicate). Thus, p will eventually learn that everyone's time has passed the timestamp of the request, and will be able to process the request. Otherwise, if p's view changes, then p processes the request when it receives the next view. Whenever p processes requests, it places them on the out queue, and eventually delivers them to its client. ■

Stabilization

We now prove that the IADS algorithm satisfies the IADS Stabilization property (Property 10.1.1).

First, we argue that stabilization of a set of clients leads to the stabilization of GCS. Consider an execution α of IADS in which a set Ω of clients becomes and remains stable from some point on (Definition 10.1.1). Recall that we assume that the underlying model is an asynchronous system enriched with an eventually perfect failure detector ($\diamond P$), as defined by Chockler et al., in [27]. In [27], Chockler et al., also prove that an eventually perfect failure detector is equivalent to a precise group membership service. Thus, it is valid to assume that the MBRSHP service component of GCS behaves like a precise group membership service; in other words, that execution α satisfies View Stability (Property 5.2.1): the MBRSHP service component of GCS delivers a stable view v with the membership set Ω to the GCS end-points of Ω. Since α satisfies View Stability, GCS satisfies Liveness Property 5.2.2: GCS eventually delivers view v to every application end-point in Ω, and does not deliver any subsequent views; moreover, GCS eventually delivers to every application end-point in Ω every message sent in view v. It remains to prove that, after every application end-point p of Ω receives the final view v, the end-point eventually outputs the final $\texttt{refresh}_\texttt{p}(\Omega, \textit{base\_state})$ event, where *base_state* is the same at all the members.

We now prove that the stabilization of GCS leads to the stabilization of IADS.

Theorem 10.2.6 (IADS Stabilization) *In a fair execution in which all application end-points of Ω receive the same view* v*, with* v.set $= \Omega$*, and no subsequent views afterwards, every application end-point* p $\in \Omega$ *eventually starts operating in* normal *mode in* v*, eventually outputs* $\texttt{refresh}_\texttt{p}(\Omega, \textit{base\_state})$*, and produces no* $\texttt{refresh}_\texttt{p}$ *events afterwards, where base_state is the same for all application end-points in* v.set*.*

Proof : If we show that everyone in v.set eventually places the correct refresh information in its out queue, we are done, because, by fairness, everything placed in the out queues gets eventually delivered to the clients. Since p does not receive any other views after v, it does not place any other refresh information on the out queue.

- If $\texttt{SS}_\texttt{p}^\texttt{v} = \texttt{v.set}$, then p sets mode to normal and places the refresh information, $\langle \texttt{v.set}, \texttt{base\_obj}_\texttt{p}^\texttt{v} \rangle$, on the out queue when it receives view v. By Lemma 10.2.4 all p $\in$ v.set have the same $\texttt{base\_obj}_\texttt{p}^\texttt{v}$. Thus, when the final refresh events eventually occur at every p $\in$ v.set, the refreshed object state *base_state* $= \texttt{base\_obj}_\texttt{p}^\texttt{v}$ is the same at all these events.
- If $\texttt{SS}_\texttt{p}^\texttt{v} \neq \texttt{v.set}$, then by Corollary 10.2.1 part 2, there is no application end-point q $\in$ v.set such that $\texttt{SS}_\texttt{q}^\texttt{v} = \texttt{v.set}$. Thus, every q $\in$ v.set participates in state-transfer protocol in v. We now argue that eventually the state-transfer protocol completes at every p, at which time the application end-point switches to normal mode and places the refresh information, $\langle \texttt{v.set}, \texttt{base\_obj}_\texttt{p}^\texttt{v} \rangle$, on its out queue.

  From the code, we see that state-transfer protocol completes when application end-point p collects state-transfer messages on behalf of everyone in v.set. We need to show that, for every q $\in$ v.set, there is some application end-point t $\in$ v.set that sends a state-transfer message on q's behalf. Fix q and let t be $\texttt{min}(\texttt{SS}_\texttt{q}^\texttt{v})$. By Lemma 10.2.2, we have $\texttt{SS}_\texttt{t}^\texttt{v} = \texttt{SS}_\texttt{q}^\texttt{v}$, and thus $\texttt{t} = \texttt{min}(\texttt{SS}_\texttt{t}^\texttt{v})$; so, t sets its send_state flag to true and eventually multicasts a state-transfer message, $\langle \texttt{SS}_\texttt{t}^\texttt{v}, \texttt{obj}_\texttt{t}^\texttt{v} \rangle$ to everyone in view v. Since view v is stable, GCS eventually delivers this message to every p $\in$ v.set, at which time p subtracts $\texttt{SS}_\texttt{t}^\texttt{v}$ (and in particular q) from its States_Await set and adds $\langle \texttt{q}, \texttt{obj}_\texttt{t}^\texttt{v} \rangle$ to its StatesV set. Hence p eventually collects state-transfer messages on behalf of everyone in v.set and completes the state-transfer protocol: computes $\texttt{base\_obj}_\texttt{p}^\texttt{v}$ as a merge of the object states collected in StatesV, switches to normal mode, and places $\langle \texttt{v.set}, \texttt{base\_obj}_\texttt{p}^\texttt{v} \rangle$ on its out queue.

  Again, by Lemma 10.2.4, all p $\in$ v.set have the same $\texttt{base\_obj}_\texttt{p}^\texttt{v}$. Thus, when the final refresh events eventually occur at every p $\in$ v.set, the refreshed object state *base_state* $= \texttt{base\_obj}_\texttt{p}^\texttt{v}$ is the same at all these events. ■

Interim Atomicity

Consider a fair execution in which all application end-points of Ω receive the same view $\mathtt{v}$, with $\mathtt{v.set} = \Omega$, and no subsequent views afterwards. Let *base_state* be $\mathtt{base\_obj}_{\mathtt{p}}^{\mathtt{v}}$ for any $\mathtt{p} \in \mathtt{v.set}$, since $\mathtt{base\_obj}_{\mathtt{p}}^{\mathtt{v}}$ are the same for all $\mathtt{p} \in \mathtt{v.set}$. Let *init_ops*($\mathtt{p}$) be $\mathtt{post}_{\mathtt{p}}^{\mathtt{v}}[\mathtt{p}].\mathtt{inp}$.

We need to prove that the subsequence of a trace of IADS comprised of the events involving the clients of Ω after the final `refresh` events is indistinguishable from a trace of AO$[\Omega, \mathit{base_state}, \mathit{init_ops}]$. The proof of this property relies on the correctness of the active replication algorithm that uses logical timestamps to implement global ordering of operations. This algorithm is well-known and has been proved correct before.

10.2.3 Performance

We consider two performance characteristics of IADS. The first one is how quickly IADS reconfigures and delivers `refresh` inputs when instabilities occur. The second one is how quickly IADS processes requests and delivers responses to its clients. There are other interesting performance characteristics that we could have considered in addition to these ones: for example, message complexity of reconfiguration.

Bound on stabilization

We consider executions of IADS in which a set Ω of clients stabilizes from some point on and all application end-points of Ω receive the same final view $\mathtt{v}$, with $\mathtt{v.set} = \Omega$. Theorem 10.2.6 established that every application end-point $\mathtt{p} \in \Omega$ eventually outputs a final $\mathtt{refresh}_{\mathtt{p}}(\Omega, \mathit{base\_state})$ event.

There are two possible scenarios depending on whether or not the application end-points of Ω execute a state-transfer protocol while in view $\mathtt{v}$.

1. If not, then the final $\mathtt{refresh}_{\mathtt{p}}$ event occurs immediately after the final $\mathtt{view}_{\mathtt{p}}(\mathtt{v}, \mathtt{T}_{\mathtt{p}}^{\mathtt{v}})$; as in Chapter 9 we assume that local actions, except for the $\mathtt{block\_ok}_{\mathtt{p}}$ responses and for the heartbeat messages $\mathtt{gcs.send}_{\mathtt{p}}(\langle\text{'}\mathbf{lt\_msg}\text{'}, ...\rangle)$, occur immediately after being enabled and take zero time. That is, $\mathtt{T}[\mathtt{refresh}_{\mathtt{p}}] = \mathtt{T}[\mathtt{gcview}_{\mathtt{p}}]$, where $\mathtt{refresh}_{\mathtt{p}}$ denotes the final refresh event at $\mathtt{p}$, and $\mathtt{gcview}_{\mathtt{p}}$ refers to the $\mathtt{view}_{\mathtt{p}}(\mathtt{v}, \mathtt{T}_{\mathtt{p}}^{\mathtt{v}})$ event (Definition 9.2.4, page 113).

2. Otherwise, if application end-points of Ω execute a state-transfer protocol while in view $\mathtt{v}$, the final `refresh` events occur once the protocol completes. Let $\delta_{\mathtt{st}}$ denote the latency of state-transfer messages in view $\mathtt{v}$. Then, $\mathtt{T}[\mathtt{refresh}_{\mathtt{p}}] \leq \mathtt{last\text{-}gcview} + \delta_{\mathtt{st}}$, where `last-gcview` refers to the last event among the $\mathtt{gcview}_{\mathtt{p}}$ events for all $\mathtt{p} \in \Omega$ (Definition 9.2.4, page 113).

For simplicity, we assume that $\delta_{\mathtt{st}}$ is the same as the underlying message latency δ of CO_RFIFO (Constraint 9.2.3, page 115). In general, $\delta_{\mathtt{st}}$ could be larger than δ because the size of state-transfer messages is typically much larger than that of regular messages.

Now, we can use the performance results about the GCS system that we derived in Chapter 9 to express an upper-bound on $\mathtt{T}[\mathtt{refresh_p}]$ in terms of the time of the final group event `last-ge` and in terms of various low-level timing assumptions, such as detection latency $\delta_{\mathtt{detection}}$ and message latency δ. In particular, we use the upper-bound for $\mathtt{T}[\mathtt{gcview_p}]$ established by Corollary 9.5.1, page 126.

Roughly speaking, the following Theorem 10.2.7 states that every application end-point delivers the final refresh event within about two latencies – one detection latency and one message latency – away from the time the final group event occurs, provided the application end-point is able to rely on Virtual Synchrony to avoid participating in a state-transfer protocol. Otherwise, if state-transfer is necessary, the final refresh event occurs within about three latencies – one detection latency and two message latencies – away from the time the final group event occurs.

Theorem 10.2.7 *Let α be a timed execution of* IADS *in which view* `v` *becomes stable, as defined by Property 5.2.1. Assuming that* `last-ge` *is the final group event that results in the group's membership stabilizing to* `v.set`*, the following bound holds for each* $\mathtt{p} \in \mathtt{v.set}$*:*

$$\mathtt{T}[\mathtt{refresh_p}] \leq \begin{cases} \mathtt{T}[\mathtt{last\text{-}ge}] + \delta_{\mathtt{detection}} + \delta + c & \textit{if } \mathtt{SS}_{\mathtt{p}}^{\mathtt{v}} = \mathtt{v.set} \\ \mathtt{T}[\mathtt{last\text{-}ge}] + \delta_{\mathtt{detection}} + 2 \times \delta + c & \textit{if } \mathtt{SS}_{\mathtt{p}}^{\mathtt{v}} \neq \mathtt{v.set} \end{cases},$$

provided Assumptions 9.3.1, 9.4.1, and 9.4.2 hold, and provided the suffix of α following the `last-mstart` *event satisfies Constraints 9.2.1, 9.2.2, and 9.2.3 with the given set of application end-points* `v.set` *and time index* `T[last-mstart]`*.*

Proof : Follows immediately from Corollary 9.5.1 and the code for the IADS application end-points. ■

Notice that the condition for whether or not the application end-point avoids a state-transfer in view `v` is internal to the IADS algorithm: The theorem states this condition in terms of the value of the `p.SS` variable after view `v` is delivered to `p`. The condition depends on the specific behavior of the GCS system.

Theorem 10.2.7 captures one type of benefit that is enabled by the GCSs that provide Virtual Synchrony: In some situations the application is able to avoid performing a state-transfer protocol after instabilities occur. An example of a typical situation that leads to this optimization is when a member end-point disconnects or fails in temporal isolation from other group events affecting the membership. In such situations, GCS would (typically) transition all the surviving members together into the same new view. More complicated network changes may also lead to the optimization, especially because our GCS avoids reporting obsolete views to the application. Future research work is necessary in order to classify such situations more precisely.

There is also another type of benefit that is enabled by Virtual Synchrony: A reduction in the number of state messages communicated during state-transfer (when state-transfer can not be avoided). As the IADS algorithm demonstrates, because Virtual Synchrony synchronizes the application end-points that transition together into the new view, it is enough for only one representative from the sets of synchronized end-points to send its state during state-transfer.

Processing of requests

First consider how quickly client's requests are processed in a stable component when all the end-points comprising the component operate in normal mode. The IADS algorithm is able to process a given request as soon as it determines the request's position in the totally-ordered sequence of all the requests communicated in the current view and as soon as it receives and processes all the preceding requests. This time depends on the specific algorithm used for totally ordering requests.

For the sake of concreteness, we presented a simple, symmetric algorithm based on logical time. Recall that, in this algorithm, an application end-point determines the order of an operation o as soon as the end-point learns that all the other end-points have reached the logical time assigned to o. In the worst case, this time is bounded by approximately two message latencies (a precise bound must involve the frequency of heart-beat messages). However, if the end-points' logical clocks are kept closely synchronized, then the bound is approximately one message latency (as explained in [71, p. 609]).

Instead of the total-order algorithm based on logical time, we could have used other total-order algorithms, such as based on a sequencer process or a token (e.g., [24, 55, 20, 78], or on other, more sophisticated schemes [25, 80]. An algorithm that uses a dedicated "leader" end-point to order requests and inform other end-points of the ordering requires two message latencies: one from the original end-point to the leader and the other from the leader to the end-points. Token-based solution achieve similar performance. In general, the performance of state-of-the-art total order algorithms, in situations when the underlying network is well-behaved, is close to a single message latency.

An alternative approach to using Group Communication for building replicated data services is to use Consensus (e.g., [67, 83]). In this approach, during normal mode of operation, the servers hosting object replicas run Consensus to agree on the order in which to process clients' requests. Different replicated services differ in the particular data consistency semantics that they guarantee. For the services that provide weaker consistency semantics, such as the IADS application, using Consensus is an overkill. Our algorithm for IADS is able to process requests in the time it takes to totally-order them; in the case of a leader-based scheme — one round-trip time to the leader. In contrast, *optimized* solutions based on Consensus require two round-trip messages: the original message to the leader, a "query" message from the leader to the end-points and back, and then the "decision" message from the leader to the end-points [67]. In addition, the replication approaches based on Group Communication provide convenient mechanisms to support partitionable semantics and dynamic sets of clients; Consensus offers no such mechanisms.

Now, we consider different factors that affect processing of requests when the clients' component undergoes changes or when the IADS algorithm executes state-transfer protocol.

When instabilities occur, the total-order algorithm may take longer than "normal-case" latency to finalize the position of a request, and in some cases it may not succeed. For example, the end-point may never receive the information from disconnected end-points that is necessary for determining the position of the request. Our algorithm guarantees that, once the end-point starts processing a request (i.e., multicasts the request to other end-points using GCS), it will apply the request to the object replica and respond to the client at most by the time the end-point receives a next view.

An addition factor that must be considered is the time it takes the end-point to *start* processing a request after the request is submitted by the client. This time depends on whether or not at the time the request is submitted the end-point is blocked by GCS. If it is not blocked, then the end-point starts processing the request immediately after it is submitted. Otherwise, the end-point keeps the request in its `inp` queue and then starts processing the request only at the time it receives the next view from GCS.[5]

The final factor that affects how quickly the end-point processes requests is the mode of operation, which can be either normal or state-transfer. Recall that the end-point may apply clients' operations to its object replica only when its mode is `normal`, i.e., it is not engaged in state-transfer. Thus, when the end-point receives a view from GCS and starts processing operation requests in this new view, the final application of the operations to the object replica is delayed until the end-point completes the state-transfer protocol. In situations, when the end-point is able to rely on Virtual Synchrony to avoid participating in a state-transfer protocol, this potential delay is also avoided.

[5]An extension of Virtual Synchrony, called *Optimistic Virtual Synchrony*, supports transmission of application messages in parallel with formation of new views (Sussman et al., in [90, 89]). With Optimistic Virtual Synchrony, the IADS application can be modified to avoid the processing delay caused by blocking.

Chapter 11

Conclusions

We have developed a formal design of a novel group communication service targeted for WANs. The design implements a variant of the Virtual Synchrony semantics that includes a collection of properties that have been shown useful for many distributed applications (see [27]). Many GCSs, for example [93, 82, 12, 8, 34], support these and similar properties. The design provides theoretical underpinnings for a practical group communication system for WANs: our Virtual Synchrony algorithm was implemented (in C++) [92] as an extension of the Xpand system [10] under development at the Hebrew University of Jerusalem.

The motivation for this dissertation was twofold: First, the specifications and descriptions of the existing GCSs had been shown to be highly imprecise and ambiguous, and some, as we discovered, had serious algorithmic flaws. Due to the complexity of GCSs, prior efforts formalizing these systems had only limited success. Second, the existing systems had been designed for local-area networks, and did not perform well in wide-area networks. The dissertation addresses both of these challenges.

The contributions made by this dissertation include a complete formalization of a large-scale Virtually Synchronous GCS, a superior WAN-oriented architecture, and a superior algorithm for implementing Virtual Synchrony.

The formalization of the Virtually Synchronous GCS includes: precise specifications of the GC service and its execution environment; a precise and modular description of the algorithm implementing the service specification; a formal correctness proof showing that the algorithm satisfies the specified properties; and a formal analysis of the algorithm's performance.

The new architecture decouples two key components of GCSs — Virtual Synchrony and Membership — allowing them to execute asynchronously and in parallel. Such decoupling has been regarded as critical for providing scalable and efficient group communication services in wide-area networks: It allows for the utilization of a scalable architecture for group *membership* in which a small set of membership servers maintains group membership of a large set of clients [9]. The architecture suggested here has been adopted by the Moshe system developed at UCSD and MIT [58].

The new algorithm for implementing Virtual Synchrony involves just a single message-exchange round among the participating processes. This constitutes a significant improvement from the existing solutions (e.g., [43, 8, 47, 6, 12]), which require at least two communication rounds. Moreover, unlike existing solutions (e.g., [8, 47, 12, 82]), the new algorithm is able to respond dynamically to cascading connectivity changes, without wasting resources on handling obsolete network situations. These innovations to the algorithm are critical for WANs, which typically have high and unpredictable message latency, and frequent connectivity changes.

At the heart of the decoupled architecture and the one-round Virtual Synchrony algorithm lies a simple yet powerful idea: start-change identifiers. When a membership server decides to start forming a new view, it notifies its clients (the GCS end-points) about this; the notification message includes a start-change identifier that corresponds to the current view formation attempt of the server. Upon receiving these notifications, the GCS end-points start a synchronization protocol by sending their synchronization messages tagged with the start-change identifiers. The protocol proceeds in parallel with the membership service forming a new view; when formed and delivered to the GCS end-points, the view will include information about which start-change identifiers were given to which member. This information communicates to the end-points which synchronization messages they need to consider from other end-points. Since no pre-agreement upon a common identifier takes place, there is nothing that inhibits the membership service and the Virtual Synchrony algorithm from handling changes in the membership when a view formation and synchronization protocols are already in progress: the membership service has to notify the affected end-points of the change, and the end-points just have to forward their synchronization messages to the joining members.

A distinct and important characteristic of our design is the high level of formality and rigor at which it was carried out. This dissertation provides precise descriptions of the GCS algorithm and its semantics, as well as a formal proof of the algorithm's correctness (both safety and liveness). Previously, formal approaches based on I/O automata were used to specify the semantics of Virtually Synchronous GCSs and to model and verify their applications, for example, in [26, 39, 31, 63, 32, 51]. However, due to their size and complexity, Virtual Synchrony algorithms were not previously modeled using formal methods, nor were they assertionally verified. Our experience has taught us the importance of careful modeling and verification: in the process of proving our algorithm's correctness we often uncovered subtleties and ambiguities that had to be resolved.

Developing a formal and rigorous design was challenging due to the project's scale and intricacy. In order to accomplish this goal, we developed a novel inheritance-based methodology for incremental construction of specifications, algorithms, and, above all, proofs (Chapter 3, [61, 62]). The key contribution of this methodology is a generic framework for reuse of proofs analogous and complementary to the reuse provided by object-oriented software engineering methodologies; this is the first work that addressed reuse of proofs. Proof reuse is critical for improving cost-effectiveness and scalability of formal methods, which itself is crucial for successful efforts in modeling and validation of large-scale software systems.

Using the inheritance-based methodology, we were able to specify, describe, and verify the Virtual Synchrony algorithm incrementally, at each step focusing on a different property.

This way, it was easy to see which part of the algorithm corresponds to which property of the specification. Likewise, the proof was broken up into pieces of manageable sizes, with each piece focusing solely on the part of the code that was being proven. For example, in order to prove that VS_RFIFO+TS simulates VSRFIFO : SPEC we extended the simulation relation from WV_RFIFO to WV_RFIFO : SPEC and reasoned solely about the extension, without repeating the reasoning about the parent components; this-proof reuse was justified by the Proof Extension theorem of Chapter 3. The use of the incremental modeling and verification methodology was the key to our success in formalizing such a complex and sophisticated distributed system. We hope that this methodology shall also be helpful to other researchers working on formal modeling of complex distributed systems.

Previous projects that modeled large-scale complex systems using I/O automata relied on *composition*: complex algorithms were expressed by multiple manageable parts that jointly compose the algorithm (see, for example, [51]). In the context of our project, however, composition could not have been used in lieu of inheritance for two reasons: First, composition does not allow different components to share the same data structures. In contrast, all the parts of our algorithm share common data structures such as message buffers. Using composition, we would have had to duplicate these data structures as well as the book-keeping logic associated with them. This would make the algorithm models more cumbersome.

The second and more important reason is that composition does not allow for proof reuse, since it does not guarantee that one component does not violate the guarantees of another. Consider our project, for example. Had we implemented the Virtually-Synchronous Delivery property as a layer above WV_RFIFO, we would have had no guarantee that the combined system preserved the Within-View Delivery and the Reliable FIFO properties. In fact, we would have had to prove (1) that the WV_RFIFO layer satisfies these properties; (2) that the Virtually-Synchronous Delivery layer does not change message ordering in a way that violates the WV_RFIFO properties; and (3) that the Virtually-Synchronous Delivery layer satisfies the Virtually-Synchronous Delivery property. When introducing a third layer that implements Self Delivery, we would, once again, have had to prove that the new component does not violate the previously established properties. This repetition of reasoning is precisely what our inheritance-based technique allowed us to avoid.

In addition to specifying, modeling, and verifying our design, we have also carried out a theoretical analysis of its performance characteristics. The analysis follows the *conditional approach* that involves statements about system's behavior, under particular assumptions about behavior of the environment and of the network substrate. Specifically, in our analysis, we considered an execution of GCS in which a network component becomes stable from some point on, and investigated how long it takes GCS to deliver the final views to the component members after the final group event affecting the component occurs.

One important feature of the conditional approach to performance analysis is that it focuses on establishing precise performance predictions for particular, interesting types of executions (such as normal and failure cases); this is in contrast to a more common approach that lumps all the cases together and complicates and obscures the situation with probabilities. The analysis we carried out in this dissertation is a first step towards a well-defined methodology. To go one step further, other execution scenarios must be considered. The expected behavior of the system can then be derived from the performance results for different scenarios and from the probabilities of these scenarios occurring.

Another important feature of the conditional approach is that it lends itself naturally to composition: The performance characteristics of the entire system can be expressed as a composition of the performance properties of individual components. Thus, in our analysis, we first analyzed the performance of the Virtual Synchrony algorithm run by GCS end-points in terms of the timings of the relevant MBRSHP events, and under certain low-level timing assumptions. Then, we considered a specific membership service, [58], and expressed reasonable time bounds on the events of this service in terms of the timings of the underlying network events. Finally, we composed the two bounds to yield a performance result for the GCS system as a whole.

The advantage of the compositional approach to performance analysis is that different components can be studied in isolation, and that the analysis of a system can be reused when alternative implementations of system components are plugged in instead of the original ones. Thus, if a different GCS uses our Virtual Synchrony algorithm in the context of a membership service different from [58], it would be straightforward to compose the performance characteristics that we derived for the Virtual Synchrony algorithm with those of the membership service to yield the result about the performance of the GCS.

Finally, in order to illustrate the utility of our GCS system, we presented a simple algorithm that uses the GCS system to implement a replicated data service. The service allows a dynamic group of clients to access and modify a replicated data object. It guarantees a particular type of weak data consistency that preserves atomicity when the set of clients and the underlying network remain stable. We called this type of consistency *Interim Atomicity.*

The algorithm we presented formalizes the ideas of [5, 66, 83]. Our contribution lies in finding a clean, clear way to specify the semantics of Interim Atomicity. In our specification, we define a *generalized atomic object*, and then require the subsequence of a trace of the system involving the clients of a stable component to be a trace of the generalized atomic object. An interesting future step is to use this approach to study and define other types of weak consistency semantics that could be useful for applications that operate in highly dynamic, partitionable environments (e.g., mobile).

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Divyakant Agrawal and Amr El Abbadi. Exploiting logical structures in replicated databases. *Information Processing Letters*, 33(5):255–260, January 1990.

[4] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *3rd International Workshop on Services in Distributed and Networked Environment (SDNE)*, pages 84–91, June 1996.

[5] Y. Amir, G. V. Chockler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version: Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[6] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

[7] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1994.

[8] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4), November 1995.

[9] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In Marios Mavronicolas, Michael Merritt, and Nir Shavit, editors, *Networks in Distributed Computing (DIMACS workshop)*, volume 45 of *DIMACS*, pages 23–42. American Mathematical Society, 1998.

[10] T. Anker, G. Chockler, I. Shnaiderman, and D. Dolev. The design of Xpand: A group communication system for wide area networks. Technical Report 2000-31, Institute of Computer Science, Hebrew University, Jerusalem, Israel, July 2000.

[11] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

[12] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, April 2001. Previous version: University of Bologna Department of Computer Science Technical Report UBLCS98-1.

[13] Ralph-Johan Back, Anna Mikhajlova, and Joakim von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 12:18–40, 2000.

[14] A. Bartoli, B. Kemme, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communications. Technical Report UBLCS-2000-17, Departement of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna (Italy), December 2000.

[15] Philip Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 7-8, pages 217–304. Addison-Wesley, 1987.

[16] Mark Bickford and Jason Hickey. An object-oriented approach to verifying group communication systems. http://www.cs.cornell.edu/jyh/papers/cav99_ooioa/, 1998.

[17] K. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[18] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. In *Multimedia Computing and Networking (MMCN98)*, 1998.

[19] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 123–138. ACM, Nov 1987.

[20] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[21] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[22] T. Budd. *An Introduction to Object-Oriented Programming, 2nd Edition.* Addison Wesley Longman, 1996.

[23] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[24] J. Chang and N. Maxemchunk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3), 1984.

[25] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 237–246, June 1998.

[26] G. V. Chockler. An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions. Master's thesis, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1997.

[27] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, December 2001. Previous version: MIT Technical Report MIT-LCS-TR-790, September 1999.

[28] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994. Preliminary version in Proceedings of OOPSLA'89, ACM SIGPLAN 4th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pages 433–443, New Orleans, Louisiana, October 1989.

[29] F. Cristian and F. Schmuck. Agreeing on process group membership in asynchronous distributed systems. Technical Report CSE95-428, Department of Computer Science and Engineering, University of California, San Diego, 1995.

[30] R. De Prisco. *On building blocks for distributed systems.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, December 1999.

[31] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 227–236, June 1998.

[32] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic primary configuration group communication service. In *13th International Symposium on DIStributed Computing (DISC)*, pages 64–78, Bratislava, Slovak Republic, 1999.

[33] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[34] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[35] D. Dolev, D. Malki, and H. R. Strong. An asynchronous membership protocol that tolerates partitions. Technical Report CS94-6, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1994.

[36] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[37] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 300–309, May 1996.

[38] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science special issue on Distributed Algorithms*, 220, 1999.

[39] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001. Previous version appeared in PODC 1997.

[40] R. Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical Report TR-95-1506, Cornell University, 1995.

[41] R. Friedman and A. Vaysburd. Fast replicated state machines over partitionable networks. In *16th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1997.

[42] R. Friedman and A. Vaysburd. High-performance replicated distributed objects in partitionable environments. Technical Report 97-1639, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, July 1997.

[43] Roy Friedman and Robbert van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University, August 1995.

[44] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, USA, 2000.

[45] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems.* MIT Lab for Computer Science, Cambridge, MA, December 1997. http://sds.lcs.mit.edu/∼garland/ioaLanguage.html.

[46] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, October 1997. IEEE Computer Society Press.

[47] Katherine Guo, Werner Vogels, and Robbert van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *7th ACM SIGOPS European Workshop*, September 1996.

[48] David Harel and Orna Kupferman. On the behavioral inheritance of state-based objects. In *Technology of Object-Oriented Languages (TOOLS)*. IEEE Computer Society Press, Los Alamitos, CA, July 30th – August 3rd 2000.

[49] Andreas V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Proceedings of Theoretical Aspects of Computer Software*, pages 548–568. Springer-Verlag (*LNCS* 526), 1991.

[50] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[51] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for ensemble layers. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer-Verlag, March 1999.

[52] M. Hiltunen and R. Schlichting. Properties of membership services. In *2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, 1995.

[53] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[54] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[55] M. F. Kaashoek and A. S. Tanenbaum. An evaluation of the Amoeba group communication system. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 436–447, May 1996.

[56] Samuel Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Fifteenth Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[57] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[58] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 2002. To appear.

[59] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20th International Conference on Distributed Computing Systems (ICDCS)*, pages 344–355. IEEE Computer Society Press, April 2000. Full version: MIT Lab. for Computer Science Tech. Report MIT-LCS-TR-794.

[60] Idit Keidar and Roger Khazan. A virtually synchronous group multicast algorithm for WANs: Formal approach. *SIAM Journal on Computing*, 2002. To appear.

[61] Idit Keidar, Roger Khazan, Nancy Lynch, and Alex Shvartsman. An inheritance-based technique for building simulation proofs incrementally. In *22nd International Conference on Software Engineering (ICSE)*, pages 478–487. ACM, June 2000.

[62] Idit Keidar, Roger Khazan, Nancy Lynch, and Alex Shvartsman. An inheritance-based technique for building simulation proofs incrementally. *ACM Transactions on Software Engineering and Methodology*, 11(1):1–29, January 2002. Previous version in ICSE 2000, pp. 478–487.

[63] Roger Khazan, Alan Fekete, and Nancy Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on DIStributed Computing (DISC)*, pages 258–272, Andros, Greece, September 1998.

[64] L. Lamport. On interprocess communication, Parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[65] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[66] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.

[67] B. Lampson. How to build a highly available system using consensus. In Babaoğlu and Marzullo, editors, *Distributed Algorithms*, LNCS 1151. Springer-Verlag, 1996.

[68] B. Lampson. Generalizing Abstraction Functions. Massachusetts Institute of Technology, Laboratory for Computer Science, principles of computer systems class, handout 8, 1997. ftp://theory.lcs.mit.edu/pub/classes/6.826/www/6.826-top.html.

[69] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. and Sys.*, 16(1):1811–1841, November 1994.

[70] Barbara Liskov and Jeannette M. Wing. A New Definition of the Subtype Relation. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 118–141, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[71] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[72] N. A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[73] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[74] S. McCanne. A distributed whiteboard for network conferencing, 1992.

[75] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1995.

[76] J. Misra and K.M. Chandy. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[77] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994. Full version: technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[78] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[79] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[80] L. Rodrigues, H. Fonseca, and P. Verissimo. Totally ordered multicast in large-scale systems. In *International Conference on Distributed Computing Systems (ICDCS)*, 1996.

[81] Luis Rodrigues, Katherine Guo, Antonio Sargento, Robbert van Renesse, Brad Glade, Paulo Verissimo, and Ken Birman. A dynamic light-weight group service. In *15th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 23–25, October 1996. also Cornell University Technical Report, TR96-1611, August, 1996.

[82] A. Schiper and A.M. Ricciardi. Virtually synchronous communication based on a weak failure suspector. In *23rd IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 534–543, June 1993.

[83] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[84] G. Shamir. Shared Whiteboard: A Java Application in the Transis Environment. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, October 1996. Available from: `http://www.cs.huji.ac.il/∼transis/publications.html`.

[85] A. P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Information Processing Letters*, 39(1):45–49, July 1991.

[86] Neelam Soundarajan and Stephen Fridella. Inheriting and modifying behavior. In R. Ege, M. Singh, and B. Meyer, editors, *Technology of Object-Oriented Languages (TOOLS-23)*, pages 148–162. IEEE Computer Society Press, 1997.

[87] Neelam Soundarajan and Stephen Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 206–215. IEEE Computer Society Press, 1998.

[88] Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of OOPSLA '95 Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 30 of *ACM SIGPLAN Notices*, pages 200–214. ACM, October 1995.

[89] J. Sussman, I. Keidar, and K. Marzullo. Optimistic virtual synchrony. Technical Report MIT-LCS-TR-792, MIT Lab for Computer Science, November 1999. Also Technical Report CS1999-634 University of California, San Diego, Department of Computer Science and Engineering.

[90] J. Sussman, I. Keidar, and K. Marzullo. Optimistic virtual synchrony. In *19th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 42–51, October 2000.

[91] J. Sussman and K. Marzullo. The *Bancomat* problem: An example of resource allocation in a partitionable asynchronous system. In *12th International Symposium on DIStributed Computing (DISC)*, September 1998. Full version: Tech Report 98-570 University of California, San Diego Department of Computer Science and Engineering.

[92] Igor Tarashchanskiy. Virtual Synchrony Semantics: Client-Server Implementation. Master's thesis, MIT Department of Electrical Engineering and Computer Science, August 2000. Master of Engineering.

[93] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[94] Daniel Yates, Nancy Lynch, Victor Luchangco, and Margo Seltzer. I/O automaton model of operating system primitives. Technical report, Harvard University and Massachusetts Institute of Technology, May 1999. Bachelor's thesis.