Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations

by

Brian A. Coan

B. S. E., Princeton University
(1977)

M. S., Stanford University
(1979)

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

Doctor of Philosophy

at the

Massachusetts Institute of Technology

June 1987

Signature of Author ______________________ 4/17/87
Department of Electrical Engineering and Computer Science

Certified by [illegible] 4/17/87
Nancy A. Lynch, ~~Thesis~~ Supervisor

Accepted by [illegible]
Arthur C. Smith, Chair
Departmental Committee on Graduate Students

Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations

by

Brian A. Coan

Submitted to the Department of
Electrical Engineering and Computer Science on April 17, 1987,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract: We give several new results in the area of fault-tolerant distributed computing. Our specific interest is consensus protocols, that is, protocols that enable correct processors to reach agreement in the presence of disruptive behavior by faulty processors. The results presented in this thesis are as follows: two new efficient agreement protocols, one randomized and one deterministic; a method for efficiently transforming a protocol that reaches agreement on a single bit into a protocol that reaches agreement on values chosen form a larger set; a general method for compiling a protocol that tolerates relatively benign processor faults into one that tolerates more serious processor faults; and a strengthening of the known lower bound on the number of rounds of communication required by consensus protocols.

Keywords: Agreement protocols, approximate agreement protocols, Byzantine generals, consensus problems, distributed systems, fault tolerance, lower bounds, performance analysis, protocol transformations, randomized protocols, time bounds.

Thesis supervisor: Nancy A. Lynch
Title: Ellen Swallow Richards Professor of Computer Science and Engineering

Contents

To my mother and my late father,
Margaret R. Coan and Edward A. Coan

Preface

It is a pleasure to acknowledge the many debts that I incurred in writing this thesis. I owe most to my thesis supervisor, Nancy Lynch. Her guidance has made this work possible. She has been tireless in suggesting ways to improve difficult early drafts. Her high standards contributed much to the quality of this project.

The two readers for this thesis, Cynthia Dwork and Bill Weihl, contributed much to its successful completion. Cynthia Dwork is the coauthor of Chapter 6. The remainder of the thesis has benefited from her thoroughness. She generously made many sacrifices to supervise this work from California. Bill Weihl took an interest in this project from its inception. He provided numerous insightful comments that improved the presentation.

Five other people made substantial contributions to portions of this thesis. Benny Chor is the coauthor of Chapter 5. Alan Fekete and Jennifer Welch provided assistance with the model used in Chapter 4. Michael Fischer suggested the method of attack that led to the result presented in Chapter 2, and he made many suggestions that improved the presentation of that result. Russell Turpin is the coauthor of the first two sections of Chapter 3, and he played a helpful role in introducing me to research in fault-tolerant distributed computing.

The Theory of Distributed Computing group, established by Nancy Lynch, has been an exciting and productive place to work. Its members have made it that way. During the past five years the members have been Baruch Awerbuch, Bard Bloom.

Jim Burns, Cynthia Dwork, Alan Fekete, Ken Goldman, Paris Kanellakis, Nancy Lynch, Michael Merritt, Yoram Moses, Betty Pothier, Liuba Shrira, Gene Stark, Mark Tuttle, Jennifer Welch, and Shmuel Zaks.

On arrival at MIT., I received wise counsel from Ben Hyde, Pete Kennedy, and Albert Meyer. Support and encouragement in trying times was provided by Randy Forgaard, David Gifford, Maurice Herlihy, Elliot Kolodner, Julie Lancaster, Bruce Lindsay, Barbara Liskov, Silvio Micali, Brian Oki, David Reed, Jim Restivo, Joe Ricchio, Jim Stamos, Kathy Yelick, Joe Zachary, and Judy Zachary.

Financial support for this work was provided by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014–83–K–0125, by the National Science Foundation under Grants DCR–83–02391 and CCR–86–11442, by the Office of Army Research under Contract DAAG29–84–K–0058, and by the Office of Naval Research under Contract N00014–85–K–0168.

Achieving Consensus in Fault-Tolerant Distributed Computer Systems

Protocols, Lower Bounds, and Simulations

1 Introduction

The recent proliferation of distributed computing has provided computer-system designers with the opportunity to create computer systems that continue to function correctly despite the failure of some components. Exploiting this opportunity has proved to be a difficult, but important, problem.

In this thesis we consider one particular problem that arises in the design of fault-tolerant distributed computer systems. Specifically, we consider the design of consensus protocols. A consensus protocol enables correct processors to reach agreement in the presence of disruptive behavior by failed processors. We give several new results for consensus protocols. Specifically, we give two new protocols, a strengthened lower bound, and three new simulation results. Our method is theoretical because, as the founding researchers in the field of consensus protocols (Wensley *et al.* [48]) say:

> [Protocols] of this type often contain very subtle errors, and extremely rigorous proofs are needed to ensure their correctness.

1. Background

In 1978 the final report of the SIFT (Software Implemented Fault Tolerance) project [48] documented an attempt to design a fault-tolerant computer system for aircraft control. The intended application of the SIFT system is one in which a failure of a computer system could render the aircraft that it controls unflyable.

Naturally, the reliability requirements for such a system are extremely high. The SIFT report proposed to achieve high reliability through the replication of components. In order to ensure the independence of hardware failures, the architecture proposed was a collection of communicating processors—a distributed system.

Taking advantage of replication to increase the fault-tolerance of a computer system is a difficult problem. One would like the correct processors to act in concert despite the disruptive behavior of any failed processors. An important insight from the SIFT project is that in a distributed system in which some processors may fail, it is necessary for the correct processors to run sophisticated consensus protocols if the system is to guarantee that the correct processors will always be able to act in concert. Since the publication of the SIFT final report, fault-tolerant consensus protocols have been an active area of research.

2. Fault Models

The computer systems that we consider consist of a collection of processors that communicate by sending messages over links. Each link connects a pair of processors. We refer to the collection of links as the communication network. It is known [27] that there are no consensus protocols unless the connectivity of the communication network is high. Throughout this thesis we assume that the communication network is fully connected.

The first step in the development of a consensus protocol is to characterize the faults that it will tolerate. There are two components that could potentially fail, processors and links. We make the assumption that the communication network is reliable and that all failures are processor failures. The principal justification for making this assumption is that links are simpler than processors and therefore potentially more reliable.

A correct processor sends messages according to its protocol. A failed processor can deviate from its protocol. One could imagine that a processor simply stops sending messages when it fails. Alternatively, one could imagine that the message-sending behavior of a failed processor is totally unconstrained. More generally, there are a variety of possible fault models ranging from relatively benign faults to relatively nasty faults. The four most commonly studied fault models are as follows:

- *Crash fault model:* A failed processor follows its protocol correctly for some time. Then, it stops completely, sending no more messages.

- *Failure-by-omission fault model:* A failed processor follows its protocol correctly except that some of its outgoing messages may be lost.

- *Authenticated Byzantine fault model:* Correct processors can sign the messages that they send. A failed processor can send arbitrary messages subject to the constraint that it cannot forge the signature of a correct processor.

- *Byzantine fault model:* There are no constraints on the messages that can be sent by a failed processor.

Throughout this thesis we develop protocols for the Byzantine fault model and we prove impossibility results for the crash fault model. This makes our results as strong as possible because any protocol that works in the Byzantine fault model also works *a fortiori* in more benign fault models and any impossibility result that holds in the crash fault model also holds *a fortiori* in nastier fault models.

3. Timing Models

A timing model embodies the set of assumptions that we make regarding bounds on the relative rates of processors and bounds on message delivery time. The solution to any consensus problem depends on the particular timing model. So, for each consensus problem that we consider in this thesis, we settle on a timing model. If we assume a high amount of synchronization, then it is relatively easy to solve a consensus problem; if we assume a low amount of synchronization, then it is relatively hard (or in some cases impossible). The two most widely studied timing models are the synchronous model and the asynchronous model.

We say that a system is *synchronous* if all processors act in unison. That is, communication takes place in a series of *rounds.* In each round each processor sends messages, receives all of the messages sent to it, and acts on the messages received. The synchronous model is the most optimistic timing model that we know. Dolev, Dwork, and Stockmeyer [17] showed that an equivalent formulation of the synchronous model is to assume a fixed, known upper bound on message delivery time and on the difference in rate between each pair of processors.

We say that a system is *asynchronous* if there is no bound on the relative rates of processors and if there is no bound on the message delay. The asynchronous model is the most pessimistic timing model that we know. A famous result by Fischer, Lynch, and Paterson [28] showed that in the asynchronous model it is impossible to solve consensus problems if we require exact solutions and if we require that protocols always terminate. Subsequent results have shown that it is possible to solve certain consensus problems in asynchronous systems. The problems that can be solved either admit approximate solutions or allow protocols that terminate with high probability, but not with certainty.

With the exception of Chapter 4, we restrict our attention to synchronous systems. In Chapter 4 we restrict our attention to asynchronous systems. We do not consider any timing models intermediate between synchronous and asynchronous. Identifying interesting intermediate timing models is a potentially productive, but largely unexplored, area for research. Attempts in this direction have been made by Coan and Lundelius [14] and by Dwork, Lynch, and Stockmeyer [23].

4. The Agreement Problem

The most commonly studied consensus problem is the agreement problem. A protocol for the agreement problem is run by a distributed system of processors. Each processor starts the protocol with an input value v from a fixed set V of legal inputs. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on an element of V as its answer. There are three conditions that the correct processors must satisfy.

- *Agreement condition:* All correct processors that decide reach the same decision.
- *Validity condition:* If all correct processors start the protocol with input v, then v is the decision of all of the correct processors that decide.
- *Termination condition:* All correct processors eventually decide.

5. An Application of an Agreement Protocol

It may not be apparent how one would use a solution to the agreement problem to construct a reliable distributed system. For instance, the validity condition may seem too weak because it only imposes a constraint on the answer when all processors have the same input. In this section we give an example that shows how a solution to the agreement problem could help in the construction of a reliable system for aircraft control.

Suppose that on an aircraft there are four units, each consisting of a processor and an altimeter. Each processor has an output line over which it can issue requests to either raise or lower a flap. The altimeters of the correctly operating units give approximately equal readings. The processors communicate over a network that is fully connected and reliable. Suppose we have a solution to the agreement problem for four processors with at most one Byzantine fault. We give a protocol $\mathcal{P}$ that tolerates the Byzantine failure of one unit and allows the correct units to reach exact agreement on an altitude within the range of the correct altimeter readings. Then we explain how the protocol $\mathcal{P}$ can be used as part of a reliable system to control the aircraft.

The protocol $\mathcal{P}$ is as follows. Each processor sends its altimeter reading to all of the processors in the system. The system runs four instances of the agreement protocol in parallel. Each processor uses as its input for instance i of the agreement protocol the altimeter reading that it received from processor i. Let v_i be the answer at processor p from instance i of the agreement protocol. Processor p forms the vector $\langle v_1, v_2, v_3, v_4 \rangle$ and answers the average of the second largest and the third largest elements in the vector.

We argue informally that this protocol enables the correct processors to agree on an altitude within the range of the correct altimeter readings. By the agreement condition satisfied by the agreement protocol, all correct processors form the same vector and thus (because they all use the same averaging function) produce the same altitude as an answer. By the validity condition satisfied by the agreement protocol, the altimeter readings of the three correctly operating units are correctly entered in the vector formed by each processor. At most one altimeter is faulty. There are two cases. Either the agreed value of the faulty altimeter is an extreme value or it is not. In either case it is easy to see that the averaging function used by all of the correct processors must produce an altitude in the range of the correct altimeter readings.

We now explain how the protocol $\mathcal{P}$ can be used as a part of a reliable system for aircraft control. We will ensure that the aircraft is completely under the control of the correct units. Recall that more than a majority of the units are assumed to be correct and recall that the correct units have used protocol $\mathcal{P}$ to agree on an altitude in the range of the correct readings. If processors are programmed so that their outputs (to the flap) are a function of their inputs (altimeter readings) and nothing else, then all correct units will produce the same output. It is possible to construct the aircraft so that its flap is not controlled directly by any one processor; rather, the flap is controlled by a physical voting device that will respond to the correct processors (a majority) and ignore the failed processors (a minority). Thus the aircraft goes in a direction that is calculated by a correct processor based on an altitude in the range of the correct altitude readings.

6. Glossary of Consensus Problems

In this thesis we discuss a large number of consensus problems. We define most of these problems in this section. For the four problems that we do not define here, we give references to the definitions. The consensus problems that we discuss fall into two broad categories: variants of the agreement problem and other consensus problems. We define all of the variants of the agreement problem here and, in Table 1, we give the locations of the definitions of the other consensus problems.

Problem	Chapter	Section
Approximate Agreement	4	2
Avalanche Agreement	2	4
Distributed Firing Squad	6	3.1

Table 1: Location of Problem Definitions

We define 48 variants of the agreement problem. They arise from one three-valued choice and four two-valued choices that are all made independently. We enumerate the choices with the default choice in **bold.** The input is either **multivalued** or binary. The agreement requirement is either for **exact** agreement or for crusader agreement. The validity requirement is either **strong** or weak. The decision of the correct processors is either **eventual** or simultaneous. Termination is either **deterministic,** randomized, or lazy.

A protocol is either randomized or deterministic. If it is randomized then each processor can make local random choices; otherwise, this capability is unavailable. It is common to require that correct processors in deterministic protocols decide in all executions. In randomized protocols this requirement is usually relaxed; processors are merely required to terminate with probability one. We distinguish these alternative termination requirements by formulating two variants of a consensus problem, one deterministic and one randomized. These problem variants differ only in the termination requirement. A third variant without any termination requirement is possible. We call this variant lazy.

In a protocol $\mathcal{P}$ that solves an agreement problem, each processor starts with an input value from some fixed set V. We refer to the elements of the set V as *values.* We require that $* \notin V$ and that $|V| \geq 2$. An agreement problem is *binary* if the input set is restricted to be of size two; otherwise the problem is *multivalued.* Each correct processor may, at some point during the execution of $\mathcal{P}$, irrevocably decide on an element of a set A as its answer. If $\mathcal{P}$ is a crusader agreement protocol then $A = V \cup \{*\}$; otherwise, $A = V$. Protocol $\mathcal{P}$ solves a particular agreement problem if it satisfies the corresponding subset of the following conditions. The agreement condition for $\mathcal{P}$ is one of the following:

- *Agreement condition for exact agreement:* All correct processors that decide reach the same decision.
- *Agreement condition for crusader agreement:* All correct processors that decide on an element of V reach the same decision.

The validity condition for $\mathcal{P}$ is one of the following:

- *Validity condition for strong agreement:* If all correct processors start the protocol with input v, then v is the decision of all of the correct processors that decide.
- *Validity condition for weak agreement:* If all processors are correct and if all processors start the protocol with input v, then v is the decision of all of the processors that decide.

The simultaneity condition for $\mathcal{P}$ is one of the following:

- *Simultaneity condition for eventual agreement:* No condition is imposed.
- *Simultaneity condition for simultaneous agreement:* If any correct processor decides then all correct processors decide in the same round. (This condition is applicable only in synchronous systems.)

The termination condition for $\mathcal{P}$ is one of the following:

- *Termination condition for deterministic agreement:* All correct processors eventually decide.
- *Termination condition for randomized agreement:* The probability that all correct processors decide by round r tends to 1 as r tends to infinity. (This formulation of the condition is applicable only in synchronous systems. An alternative formulation would be required in an asynchronous system.)
- *Termination condition for lazy agreement:* No condition is imposed.

7. Roadmap

We now outline the results presented in this thesis. All of the chapters are independent and self-contained except that Chapter 3 depends on some material presented in Chapter 2. In the discussion in this section we let n be the number of processors and we let t be an upper bound on the number of faults that a protocol needs to tolerate.

In Chapter 2 we give an upper bound on the communication requirements for fault-tolerant consensus protocols. We do this by showing that an arbitrary consensus protocol can be simulated by a communication-efficient protocol (*i.e.,* a protocol with communication cost polynomial in the number of processors). As a corollary to this result we obtain a major new result concerning the communication cost of the agreement problem in the Byzantine fault model. This new result is a polynomial-message agreement protocol that uses about half the rounds of communication used by the best previously known communication-efficient protocols. As part of our simulation we found it necessary to formulate and solve a new consensus problem that we call the avalanche agreement problem.

In Chapter 3 we present two applications of the new avalanche agreement protocol developed in Chapter 2. Specifically, we give a general technique for efficiently transforming a binary agreement protocol into a multivalued agreement protocol and we give a simple crusader agreement protocol.

In Chapter 4, extending a technique developed by Bracha [5], we give a general method for compiling a protocol that tolerates relatively benign processor faults (crash) into one that tolerates more serious processor faults (Byzantine). Our method works in asynchronous systems only. Using our compiler we find it easy to develop a new asynchronous approximate agreement protocol that is resilient to Byzantine faults. For $n = 3t + 1$, our approximate agreement protocol terminates in a number of "asynchronous rounds" that is independent of the behavior of the faulty processors. This is an improvement on a result of Dolev, Lynch, Pinter, Stark, and Weihl [19]. Their asynchronous protocol requires that $n \geq 5t + 1$ and it permits the faulty processors to determine the number of "asynchronous rounds" required for termination. In some systems—depending on the size of the system and the number of faults tolerated—our protocol has a faster convergence rate than the protocol of Dolev *et al.*

In Chapter 5 we present a new randomized eventual agreement protocol that reaches agreement in $O(t/\log n)$ expected rounds and $O(n^2t/\log n)$ expected message bits independent of the distribution of processor failures. This performance is further improved to a constant expected number of rounds and $O(n^2)$ message bits if the distribution of processor failures is assumed to be uniform. In either event, the protocol beats the lower bound on rounds for deterministic agreement protocols. Our protocol operates in the Byzantine fault model. Bracha [6] and, more recently, Dwork, Shmoys, and Stockmeyer [24] have devised new randomized eventual agreement protocols that are much faster than ours, but their protocols run in a slightly more benign fault model.

It is known that $t + 1$ is a lower bound on the number of rounds required to solve certain consensus problems. In Chapter 6 we extend the lower bound of $t + 1$ rounds to a larger class of consensus problems. Specifically, we show that the bound holds for the simultaneous randomized agreement problem and for the randomized distributed firing squad problem. Our lower bound contrasts with known fast protocols for the randomized eventual agreement problem, like the protocol presented in Chapter 5.

2

A Communication-Efficient Canonical Form for Fault-Tolerant Distributed Protocols

The task of achieving consensus among the correct processors in a fault-tolerant distributed computer system has been recognized as a fundamental problem in distributed computing. Many consensus protocols are known. Some of these require an amount of communication that is exponential in the number of processors. We present a general simulation of any synchronous consensus protocol by a communication-efficient protocol (*i.e.*, a protocol with communication cost polynomial in the number of processors). An important corollary of the simulation technique is a new communication-efficient protocol for a particular consensus problem called the agreement problem. This new protocol uses about half the number of rounds required by the best previously-known communication-efficient agreement protocol. Our new protocol approaches the known lower bound for rounds to within a small factor arbitrarily close to one. The only known protocols that achieve the lower bound for rounds use an amount of communication that is exponential in the number of processors.

1. Introduction

In a fault-tolerant distributed computer system, the consensus problem is the problem of ensuring that correct processors reach some consistent decision despite the erroneous behavior of those processors that fail. Protocols have been designed to

A preliminary version of this chapter appeared in the *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* [11].

solve many variants of the consensus problem including the agreement problem [40], the approximate agreement problem [19], the crusader agreement problem [16], and the weak agreement problem [32]. In the Byzantine fault model some of these protocols require a large (exponential in the number of processors) amount of communication. Examples are the agreement protocol of Lamport *et al.* [34] and the approximate agreement protocol of Fekete [25].

We demonstrate general upper bounds on the amount of communication needed for any consensus protocol by showing that an arbitrary consensus protocol can be simulated by a particular communication-efficient canonical-form protocol, which we call the *compact full-information* protocol. The compact full-information protocol solves the same problem as the consensus protocol that it simulates, but it uses an amount of communication that is polynomial in the number of processors and the number of rounds of message exchange.

The compact full-information protocol is a "universal" consensus protocol in the sense that all of the message exchange among the processors is independent of the particular consensus protocol being simulated. The only thing that does depend on the protocol being simulated is the decision function that is used by each processor to compute its answer (output) as a function of its current state.

There is a certain overhead in our simulations. To achieve its small communication cost, the compact full-information protocol incurs an increase in running time (*i.e.*, rounds of message exchange). It takes more than one round for the compact full-information protocol to simulate one round of a consensus protocol. There is a tradeoff between the number of rounds and the degree of the polynomial bounding the communication. The value of this tradeoff is determined by a numerical parameter to the compact full-information protocol. For any $\epsilon > 0$ there is a choice for the parameter that increases the number of rounds by a factor of $1 + \epsilon$ and that uses $O(r \cdot n^{\lceil 2/\epsilon \rceil + 3} \cdot \log |V|)$ bits of communication where n is the number of processors, r is the number of rounds of the simulated protocol, and V is the set of possible inputs to the simulated protocol.

We develop the compact full-information protocol for use in the Byzantine fault model. In more benign fault models like failure-by-omission and crash there is a simple modification of our protocol that can simulate an arbitrary consensus protocol with no increase in the number of rounds. Because the most interesting applications of our results are in the Byzantine fault model, we will restrict our attention to that model in the remainder of this chapter.

As a corollary to the results in the Byzantine fault model, we obtain a major new result about the communication requirements of agreement. Let t be an upper bound on the number of processor faults that a protocol need tolerate. The earliest agreement protocols [34] used exponential communication and $t+1$ rounds; $t+1$ is the known lower bound on rounds [26]. Subsequently, improved protocols yielded polynomial communication using about $2t$ rounds (see [18], [21], [35], and [46]). An open question among researchers in this area for the past few years has been whether there are any protocols that simultaneously use fewer than $2t$ rounds and polynomial communication. We obtain an interesting answer to this question. For any $\epsilon > 0$ there is a protocol that uses $(1+\epsilon)(t+1)$ rounds and polynomial communication where the degree of the polynomial is linear in $1/\epsilon$. We obtain this result by simulating the communication-inefficient $(t+1)$-round protocol of Lamport *et al.* with our new communication-efficient canonical-form protocol.

Another use for our technique is improving the communication complexity of a new approximate agreement protocol of Fekete [25]. His protocol has the optimal convergence rate for any multi-round approximate agreement protocol, but requires exponential communication. We can simulate his protocol with a polynomial-communication protocol that has a near optimal convergence rate.

We develop our simulation of an arbitrary consensus protocol, say $\mathcal{P}''$, by a compact full-information protocol in two stages. First we show that $\mathcal{P}''$ can be simulated by a full-information protocol; then we show that a full-information protocol can be simulated by a compact full-information protocol. A *full-information* protocol is a well-known [26] communication-inefficient canonical-form protocol in which each processor, at each round, broadcasts its entire state, receives one message from each processor, and forms its new state as the ordered collection of all messages received. The simulation of $\mathcal{P}''$ by a full-information protocol yields a canonical form protocol $\mathcal{P}'$ that solves the same problem as $\mathcal{P}''$ and that uses the same number of rounds. Unfortunately, $\mathcal{P}'$ uses an amount of communication that is exponential in system size. We complete the result by showing that $\mathcal{P}'$ can be simulated by a compact full-information protocol $\mathcal{P}$. Protocol $\mathcal{P}$ solves the same problem as protocols $\mathcal{P}''$ and $\mathcal{P}'$, and it uses an amount of communication that is polynomial in the system size.

The heart of our result is our new method for efficiently simulating a full-information protocol with a compact full-information protocol. We do this by using data compression techniques to condense the information being distributed. Messages are compressed by the sender and expanded by the recipient. In parallel with the rest of the compact full-information protocol, each processor at each round computes an expansion function that it can apply to incoming compressed messages to obtain the full message text. Our technique requires that all correct processors be able to consistently expand any message sent by a correct processor. This consistency requirement seems difficult to achieve in the presence of faults, because it requires that the correct processors agree on how to carry out the compression and expansion. Such agreement might, at first, seem to require agreement or some other time-costly protocol. We overcome this difficulty by using a new, different form of agreement that we call avalanche agreement. The overhead in rounds of our simulation is accounted for by the cost of periodically performing avalanche agreement. The difference between avalanche agreement and agreement is explained in Section 4. Using an avalanche agreement protocol to agree on their expansion function enables the correct processors to achieve a sufficient level of agreement at a cost that we can afford.

One limitation of our technique is that it uses a large amount of local computing resources. A complete reconstruction of the local state of processors in a full-information protocol requires an amount of space and time that is exponential in the system size. It is straightforward to devise an efficient data representation that permits each processor to evaluate its decision function using only a polynomial amount of space; however, the question of how much local processing time is required to reach a decision remains open.

In Section 2 we review the definition of the agreement problem. In Section 3 we give our definition of simulation, we characterize those properties of a protocol that are preserved under simulation, and we prove that the full-information protocol can simulate an arbitrary consensus protocol. In Section 4 we define the avalanche agreement problem and give a protocol that solves the problem. This avalanche agreement protocol is called as a subprotocol by the compact full-information protocol. In Section 5 we present the compact full-information protocol and prove that it simulates the full-information protocol.

2. The Agreement Problem

An agreement protocol is run by a distributed system of n processors, at most t of which may fail. Communication is over a network that is fully connected and reliable. The computation takes place in a series of rounds. In each round each correct processor first sends messages, then receives messages, and finally makes a local state change. Correct processors send messages according to their programs. Failed processors can send arbitrary messages.

Each processor starts the protocol with an input value v from a fixed set V of legal inputs. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on an element of V as its answer. There are three conditions that the correct processors must satisfy.

- *Agreement condition:* All correct processors that decide reach the same decision.
- *Validity condition:* If all correct processors start the protocol with input v, then v is the decision of all of the correct processors that decide.
- *Termination condition:* All correct processors eventually decide.

3. Protocol Simulations

In this section we give our definition of one protocol simulating another, and we characterize some important properties of protocol behavior that are preserved by our simulations. We take the first step toward showing that any consensus protocol can be simulated by a communication-efficient protocol, that is, we show that the full-information protocol can simulate any consensus protocol. After this section, the remainder of the chapter is devoted to showing how to simulate the full-information protocol using a particular communication-efficient protocol, which we call the compact full-information protocol.

3.1 Definitions

Following Lynch, Fischer, and Fowler [35], we model a consensus protocol as a synchronous system of automata. We find it convenient to introduce this formalism in order to discuss simulations. Later, when we give our protocols we will use a higher-level language. The mapping from the higher-level language to automata is straightforward.

Throughout this chapter we let n be the number of processors in the system, we let $N = \{1, \ldots, n\}$, and we let t be an upper bound on the number of processor faults that a protocol need tolerate. A protocol $\mathcal{P}$ is described by the following.

- Q is the set of processor states.
- $V \subset Q$ is the set of initial states. We say that a processor has *input* v if it starts in the initial state v.

- M is the set of messages.
- $\mu_{p,q} : Q \rightarrow M$, for $(p,q) \in N^2$, is the message generation function for messages sent from processor p to processor q.
- $\delta_p : M^n \rightarrow Q$, for $p \in N$, is the state transition function for processor p. (The prior state of processor p is omitted from the domain of δ_p because it would be redundant. Processor p can send any required information in a message to itself.)
- $\gamma_p : Q \rightarrow V \cup \{\perp\}$, for $p \in N$, is the decision function for processor p.

For any set S a 0-*dimensional array* of S is any $s \in S$. An i-*dimensional array* of S is any vector $\langle m_1, \ldots, m_n \rangle$ where, for all j, m_j is an $(i-1)$-dimensional array of S. We say that a is an *array* of S if a is an i-dimensional array of S for some i. Our definition of array is standard except that the size along each dimension is always n.

An execution of protocol $\mathcal{P}$ consists of a series of rounds. Each round consists of sending messages, receiving messages, and making a local state change. Each processor starts in the initial state corresponding to its input value. In any execution of protocol $\mathcal{P}$, a correct processor sends messages according to its message generation function and a faulty processor sends arbitrary messages from M. Formally, an *execution* of protocol $\mathcal{P}$ is a triple (F, I, H) where $F \subset N$, where $|F| \leq t$, where I is a 1-dimensional array of V, and where H is a function from $F \times \{1, 2, \ldots\} \times (N - F)$ to M. F is the set of faulty processors. I is the vector of inputs to all of the processors. For all $r \geq 1$, for all correct processors p and for all faulty processors q the value of $H(q, r, p)$ is the round r message from faulty processor q to correct processor p.

We can now give an inductive definition of the round r state of processor p in execution E of protocol $\mathcal{P}$, denoted state(p, r, E). Assume that $E = (F, I, H)$ where $I = \langle i_1, \ldots, i_n \rangle$. We define state$(p, 0, E) = i_p$. For all $r \geq 1$ we define

$$\text{state}(p,r,E) = \begin{cases} \perp & \text{if } p \in F; \\ \delta_p(m_1, \ldots, m_n) & \text{otherwise,} \end{cases}$$

where

$$m_q = \begin{cases} H(q,r,p) & \text{if } q \in F; \\ \mu_{q,p}(\text{state}(q, r-1, E)) & \text{otherwise.} \end{cases}$$

Correct processor p *decides* v in round r of execution E if $\gamma_p(\text{state}(p,r,E)) = v$ and $\gamma_p(\text{state}(p,r',E)) = \perp$ for all $r' < r$.

Execution E of protocol $\mathcal{P}$ is a *deciding execution* if there is some r such that all processors that are correct in E have decided by round r. Protocol $\mathcal{P}$ *terminates* if all executions of $\mathcal{P}$ are deciding executions. Protocol $\mathcal{P}$ is *l-bounded* if all correct processors have decided by round l in any execution of $\mathcal{P}$. Protocol $\mathcal{P}$ is *bounded* if it is l-bounded for some l. Protocol $\mathcal{P}$ is *simultaneous* if for any deciding execution E all correct processors decide in the same round in execution E.

We remark that boundedness is a stronger requirement than termination. Boundedness always implies termination; termination sometimes implies boundedness. If a protocol $\mathcal{P}$ terminates and has a finite message set and a finite input set then it follows by König's infinity lemma that protocol $\mathcal{P}$ is l-bounded for some l. In contrast, if either the message set or the input set is infinite, then there are protocols that terminate but are not bounded.

If E is a deciding execution of $\mathcal{P}$ then ans(E) is defined to be $\langle a_1, \ldots, a_n \rangle$ where a_p is the decision of processor p if processor p is correct and $a_p = \perp$ otherwise.

Predicate $\mathcal{C}$ is a *correctness predicate* if its domain is $V^n \times (V \cup \{\perp\})^n$. Protocol $\mathcal{P}$ satisfies correctness predicate $\mathcal{C}$ if for any deciding execution $E = (F, I, H)$ the value of $\mathcal{C}(I, \text{ans}(E))$ is true. Correctness predicates furnish a convenient way of formalizing the correctness requirements for a consensus protocol. Specifically, the correctness conditions for a protocol $\mathcal{P}$ that solves the agreement problem, the approximate agreement problem, the crusader agreement problem, or the weak agreement problem can be formulated as a requirement that protocol $\mathcal{P}$ terminates and that it satisfies some correctness predicate $\mathcal{C}$. For example, protocol $\mathcal{P}$ solves the agreement problem if it terminates and it satisfies the correctness predicate $\mathcal{C}$ that is defined below. Let

$$\mathcal{A}(I, A) = \bigwedge_{j,k \in N} ((a_j = a_k) \vee (a_j = \perp) \vee (a_k = \perp)),$$

and let

$$\mathcal{V}(I, A) = \left(\bigwedge_{j,k \in N} ((i_j = i_k) \vee (a_j = \perp) \vee (a_k = \perp)) \Rightarrow \bigwedge_{j \in N} ((a_j = i_j) \vee (a_j = \perp)) \right).$$

where $I = \langle i_1, \ldots, i_n \rangle$ and $A = \langle a_1, \ldots, a_n \rangle$. Now let $\mathcal{C}(I, A) = \mathcal{A}(I, A) \wedge \mathcal{V}(I, A)$. The correctness predicate $\mathcal{A}$ formalizes the agreement condition and the correctness predicate $\mathcal{V}$ formalizes the validity condition.

3.2 Definition of Simulation

Let $\mathcal{P}$ and $\mathcal{P}'$ be protocols with the same set of possible inputs. Protocol $\mathcal{P}$ *simulates* protocol $\mathcal{P}'$ if there is a non-decreasing function c from the natural numbers onto the natural numbers and a set of functions f_p for $p \in N$ from the processor states of $\mathcal{P}$ to the processor states of $\mathcal{P}'$ such that for any execution $E = (F, I, H)$ of $\mathcal{P}$ there is an execution $E' = (F, I, H')$ of $\mathcal{P}'$ with $f_p(\text{state}(p, r, E)) = \text{state}(p, c(r), E')$ for any correct processor p and for any $r \geq 0$. We say that execution E' *corresponds* to execution E, that f_p is the *simulation function* for processor p, and that c is the *scaling function.*

3.3 Protocol Properties Invariant under Simulation

In this subsection we characterize the properties of a protocol that are preserved by our notion of simulation. For the remainder of this subsection we let $\mathcal{P}$ and $\mathcal{P}'$ be arbitrary protocol that satisfy the following three assumptions.

- Protocol $\mathcal{P}$ simulates protocol $\mathcal{P}'$ with scaling function c and simulation functions f_p for all $p \in N$.
- Protocol $\mathcal{P}'$ has decision functions γ'_p for all $p \in N$.
- Protocol $\mathcal{P}$ has decision functions $\gamma_p = \gamma'_p \circ f_p$ for all $p \in N$.

Lemma 1: *If E is an arbitrary execution of $\mathcal{P}$ and E' is any execution of $\mathcal{P}'$ that corresponds to E, then $\gamma_p(\text{state}(p, r, E)) = \gamma'_p(\text{state}(p, c(r), E'))$ for any correct processor p and for any $r \geq 0$.*

Proof: Recall that $\gamma_p = \gamma'_p \circ f_p$ by assumption and that $f_p(\text{state}(p, r, E)) = \text{state}(p, c(r), E')$ by the definition of simulation. Therefore

$$\begin{aligned}\gamma_p(\text{state}(p, r, E)) &= \gamma'_p \circ f_p(\text{state}(p, r, E)) \\ &= \gamma'_p(\text{state}(p, c(r), E')).\end{aligned}$$ □

Lemma 2: *Let E be an arbitrary execution of $\mathcal{P}$ and let E' be any execution of $\mathcal{P}'$ that corresponds to E. If a correct processor q decides v in round r' of execution E' then it decides v in round r of execution E where $r = \min\{i \mid c(i) = r'\}$.*

Proof: Say processor q decides v in round r' of execution E'. For any $i < r$ processor q does not decide in round $c(i)$ of execution E' because $c(i) < r'$. By Lemma 1, processor q does not decide in round i of execution E. Thus processor q

does not decide before round r in execution E. By Lemma 1, $\gamma_q(\text{state}(q,r,E)) = \gamma'_q(\text{state}(q,r',E')$ and so processor q decides v in round r of execution E. □

Theorem 3: *The following four conditions hold.*

- *Correctness condition: If protocol $\mathcal{P}'$ satisfies some correctness predicate $\mathcal{C}$ then so does protocol $\mathcal{P}$.*
- *Termination condition: If protocol $\mathcal{P}'$ terminates then so does protocol $\mathcal{P}$.*
- *Boundedness condition: If protocol $\mathcal{P}'$ is l'-bounded for some l' then protocol $\mathcal{P}$ is l-bounded where $l = \min\{i \mid c(i) = l'\}$.*
- *Simultaneity condition: If protocol $\mathcal{P}'$ is simultaneous then so is protocol $\mathcal{P}$.*

Proof: We verify that the four conditions are satisfied.

Correctness condition: Suppose protocol $\mathcal{P}'$ satisfies correctness predicate $\mathcal{C}$. Let $E = (F, I, H)$ be an arbitrary deciding execution of $\mathcal{P}$. Let $E' = (F', I', H')$ be any execution of $\mathcal{P}'$ that corresponds to E. By the definition of simulation, $F = F'$ and $I = I'$. By Lemma 1, E' is a deciding execution and $\text{ans}(E) = \text{ans}(E')$. Therefore $\mathcal{C}(I, \text{ans}(E)) = \mathcal{C}(I', \text{ans}(E'))$ and $\mathcal{P}$ also satisfies correctness predicate $\mathcal{C}$.

Termination condition: We prove the contrapositive of the claim. Suppose that protocol $\mathcal{P}$ does not terminate. By the definition of termination there is some non-deciding execution E of protocol $\mathcal{P}$. Let E' be any execution of $\mathcal{P}'$ that corresponds to E. By Lemma 2, execution E' is a non-deciding execution. Thus, protocol $\mathcal{P}'$ does not terminate.

Boundedness condition: We show that in an arbitrary execution E of protocol $\mathcal{P}$ all correct processors decide by round l. Let E' be any execution of $\mathcal{P}'$ that corresponds to execution E. Because protocol $\mathcal{P}'$ is l'-bounded, execution E' must be a deciding execution in which the correct processors decide by round l'. By Lemma 2 execution E is a deciding execution in which the correct processors decide by round l. Thus protocol $\mathcal{P}$ is l-bounded.

Simultaneity condition: Suppose protocol $\mathcal{P}'$ is simultaneous. Consider an arbitrary deciding execution E of $\mathcal{P}$. Let E' be any execution of $\mathcal{P}'$ that corresponds to E. By Lemma 1, execution E' is deciding. Because $\mathcal{P}'$ is simultaneous there is some r' such that all correct processors decide in round r' of execution E'. By Lemma 2 there is some r such that all correct processors decide in round r of execution E. This shows that protocol $\mathcal{P}$ is simultaneous. □

3.4 A Simple Simulation

In this subsection we review the well-known result that an arbitrary consensus protocol can be simulated by the full-information protocol. In the full-information protocol (shown as Protocol 1) each processor at each round broadcasts its entire state, receives one message from each processor, and forms its new state as the ordered collection of all messages received. To be precise, when we speak of the full-information protocol, we are speaking of a class of protocols, one for each possible input set. The actual code shown as Protocol 1 works for any choice of input set. Let $\mathcal{P}_V$ denote the full-information protocol with input set V. In the remainder of this subsection we prove that an arbitrary consensus protocol $\mathcal{P}'$ with input set V is simulated by $\mathcal{P}_V$.

```
    Initialization for processor $p$:
        STATE $\leftarrow$ the initial value of processor $p$
    Code for processor $p$ in round $r$:
1.      broadcast STATE
2.      receive MSG$_q$ from processor $q$ for $1 \le q \le n$
3.      STATE $\leftarrow \langle$MSG$_1, \ldots,$ MSG$_n\rangle$
```

Protocol 1: The Full-Information Protocol

Let $\mathcal{P}'$ be an arbitrary consensus protocol with input set V, state set Q', message generating functions $\mu_{p,q}$ for $(p,q) \in N^2$, and state transition functions δ_p for $p \in N$. Let Q be the set of all arrays of V. Observe that Q is the state set of $\mathcal{P}_V$. Define the functions $f_p : Q \to Q'$ for $p \in N$ as follows

$$f_p(s) = \begin{cases} s & \text{if } s \in V; \\ \delta_p(\mu_{1,p}(f_1(s_1)), \ldots, \mu_{n,p}(f_n(s_n))) & \text{if } s = \langle s_1, \ldots, s_n \rangle. \end{cases}$$

Let c be the identity function on the natural numbers.

Theorem 4: *$\mathcal{P}_V$, the full-information protocol with input set V, simulates protocol $\mathcal{P}'$ with simulation functions f_p for $p \in N$ and with scaling function c.*

Proof: Because c is the identity function it is sufficient to show that for any execution $E = (F, I, H)$ of the full-information protocol there is an execution $E' = (F, I, H')$ of $\mathcal{P}'$ with $f_p(\text{state}(p, r, E)) = \text{state}(p, r, E')$ for any correct processor p and for any $r \ge 0$. The proof is in two stages. First, we construct the execution E'. Second, we show that the execution E' has the desired property.

Construction of the execution E': The execution $E' = (F, I, H')$ is completely specified except for H'. We now specify H', the history of messages sent by faulty processors in execution E'. For all $r \geq 0$, for all correct processors p where $\text{state}(p, r, E) = \langle s_1, \ldots, s_n \rangle$, and for all faulty processors q we specify that $H'(q, r, p) = \mu_{q,p}(f_q(s_q))$.

Verification that the execution E' has the desired property: We show that $f_p(\text{state}(p, r, E)) = \text{state}(p, r, E')$ for any correct processor p and for any $r \geq 0$. The proof is by induction on r.

Basis: ($r = 0$) This follows immediately because correct processors have the same initial states in executions E and E' and because f_p is the identity function when applied to the initial states of the full-information protocol.

Induction: Let M_q be the round r message from processor q to processor p in execution E and let M'_q be the round r message from processor q to processor p in execution E'. In the full-information protocol $\text{state}(p, r, E) = \langle M_1, \ldots, M_n \rangle$ because of step 3 of the code.

We claim that $M'_q = \mu_{q,p}(f_q(M_q))$ for all $q \in N$. If processor q is faulty the claim follows immediately from the choice of execution E'. If processor q is correct then

$$\begin{aligned} M'_q &= \mu_{q,p}(\text{state}(q, r-1, E')) && \text{By the definitions of } M'_q \text{ and } \mu_{q,p}. \\ &= \mu_{q,p}(f_q(\text{state}(q, r-1, E))) && \text{By the induction hypothesis.} \\ &= \mu_{q,p}(f_q(M_q)) && \text{From step 1 of the code.} \end{aligned}$$

This proves the claim that $M'_q = \mu_{q,p}(f_q(M_q))$ for all $q \in N$. Now we conclude the induction by calculating that

$$\begin{aligned} f_p(\text{state}(p, r, E)) &= f_p(\langle M_1, \ldots, M_n \rangle) && \text{From step 3 of the code.} \\ &= \delta_p(\mu_{1,p}(f_1(M_1)), \ldots, \mu_{n,p}(f_n(M_n))) && \text{By the definition of } f_p. \\ &= \delta_p(M'_1, \ldots, M'_n) && \text{By the claim.} \\ &= \text{state}(p, r, E'). && \text{By the definition of } \delta_p. \square \end{aligned}$$

4. Avalanche Agreement

We formulate and solve the avalanche agreement problem as a building block for use in our compact full-information protocol. At various points in the compact full-information protocol it is convenient to achieve some measure of agreement among the correct processors. We might try using a standard agreement protocol for this purpose. Unfortunately, we find that we cannot afford the cost (in rounds) of standard agreement. By using an avalanche agreement protocol instead, we are able to achieve a sufficient level of agreement among the correct processors at a cost that we can afford.

A protocol that solves the avalanche agreement problem operates under the same failure and communication assumptions as an agreement protocol. Each processor begins the protocol with an input value from some fixed set V. We refer to the elements of the set V as *values*. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on a value (element of V) as its answer. There are three conditions that the correct processors must satisfy.

- *Avalanche condition:* If any correct processor decides v in round r then all correct processors decide v by round $r + 1$.
- *Consensus condition:* If all correct processors start the protocol with input v then v is the decision of all of the correct processors by round 2.
- *Plausibility condition:* If any correct processor decides v then v must have been the input to some correct processor.

There are four ways in which the avalanche agreement problem differs from the standard agreement problem. First, there is no requirement that all executions (of an avalanche agreement protocol) terminate. Second, certain executions (those in which all correct processors have the same input) are required to terminate very fast (in two rounds). Third, in any execution that terminates, all of the correct processors are required to make their decisions within some window of two rounds. Fourth, no correct processor is permitted to produce as an answer any value that was not the input to at least one correct processor. The first of these differences tends to make the avalanche agreement problem easier to solve than the agreement problem. The remainder of the differences tend to make the avalanche agreement problem harder to solve than the agreement problem. The combined effect of all of the differences is to make the two problems incomparable.

It is straightforward to use standard techniques like those of Fischer, Lynch, and Merritt [27] to show that no avalanche agreement protocol tolerates t processor faults unless the total number of processors, n, is at least $3t + 1$. Protocol 2 solves the avalanche agreement problem for $n = 3t + 1$. It is a new deterministic protocol designed to solve this new problem; however, it incorporates many ideas from previously known randomized protocols for the standard agreement problem. Among these are the protocols of Ben-Or [2], of Chor and Coan [9], and of Rabin [43].

Consider the variant of the avalanche agreement problem in which the consensus condition has been strengthened to require agreement in one round rather than two. It is straightforward to use the proof technique of Fischer and Lynch [26] to show that if $n \leq 4t$ there is no solution to this variant. If $n \geq 4t + 1$ then it is easy to solve the problem using a simple variant of Protocol 2. We omit the details here.

```
    Initialization for processor $p$:
        VAL $\leftarrow$ the initial value of processor $p$
    Code for processor $p$ in round $r$:
1.      broadcast VAL
2.      receive $\text{MSG}_q$ from processor $q$ for $1 \leq q \leq n$
3.      let ANS be the most frequent non-$\perp$ message among the $\text{MSG}_q$
            (break ties arbitrarily)
4.      let NUM be the number of occurrences of ANS
5.      if $r = 1$ then
6.          if NUM $\geq 2t + 1$ then VAL $\leftarrow$ ANS else VAL $\leftarrow \perp$
7.      if $r > 1$ then
8.          if NUM $\geq t + 1$ then VAL $\leftarrow$ ANS
9.          if NUM $\geq 2t + 1$ and have not decided yet then decide VAL
```

Protocol 2: The Avalanche Agreement Protocol

In the following discussion and proof of Protocol 2 we append two subscripts to each variable from the protocol. The first subscript, say r, is a positive integer and the second subscript, say p, is in N. By this notation we mean the value of the subscripted variable at processor p at the *end* of round r. For example, $\text{VAL}_{r,p}$ is the value of variable VAL at processor p at the end of round r.

In any execution of Protocol 2, value v is *persistent* if there is some correct processor p such that $\text{VAL}_{1,p} = v$. Processor p *votes* for value v in round r if it sends any round r messages containing only v. In every round each correct processor broadcasts a message containing at most one value. So, a correct processor votes for at most one value in each round. A faulty processor may vote for many values by sending conflicting votes to different recipients.

We give an informal description of the avalanche agreement protocol before proving it correct. For convenience we describe the protocol from the point of view of an arbitrary correct processor p. (All processors run the same code.) At the end of round r, the variable $\text{VAL}_{r,p}$ holds the value, if any, that processor p currently prefers as its answer. In round $r+1$ processor p votes for $\text{VAL}_{r,p}$ and then updates its preference based on the votes it receives. The first round plays a special role in the protocol. In round 1, the number of values favored by correct processors is reduced to at most one—the persistent value. The protocol ensures that after round 1 no correct processor votes for any value other than the persistent value. In the second and subsequent rounds processor p uses the number of votes to predict when there will be an "avalanche" of correct processors favoring some value v, which must be the unique persistent value. As soon as processor p gets enough $(2t+1)$ votes to predict an avalanche it decides v. Processor p continues to participate in the protocol (send and receive messages) after it has decided.

Lemma 5: *There is at most one persistent value.*

Proof: Assume not. Then, there are values v and v' and correct processors p and q such that $\text{VAL}_{1,p} = v \neq v' = \text{VAL}_{1,q}$. In round 1 processor p must have received at least $2t+1$ votes for value v and processor q must have received at least $2t+1$ votes for value v'. The total number of processors is $3t+1$; therefore, at least $t+1$ processors, including at least one correct processor, voted for both v and v'. This is impossible behavior for a correct processor. We have a contradiction. □

Lemma 6: *For all correct processors p and for all rounds $r \geq 1$, either $\text{VAL}_{r,p}$ is the persistent value or $\text{VAL}_{r,p} = \perp$.*

Proof: The claim for $r=1$ follows immediately from Lemma 5, so assume that $r \geq 2$ is the first round in which the claim fails. There is some correct processor p and some non-persistent value v such that $\text{VAL}_{r,p} = v$. In round r processor p must have received at least $t+1$ votes for v; at least one is from some correct processor q. So, $\text{VAL}_{r-1,q} = v$. This contradicts the assumption that r is the first round in which the claim fails. □

Theorem 7: *Protocol 2 solves the avalanche agreement problem.*

Proof: We show that the avalanche, consensus, and plausibility conditions are satisfied.

Avalanche condition: Say that correct processor p decides v in round r. By Lemma 6, any correct processor that decides must pick the unique persistent value.

Thus, v is the decision of all of the correct processors that decide. We conclude the proof by showing that all correct processors decide v by round $r+1$. In round r processor p gets at least $2t+1$ votes for v; at least $t+1$ are from correct processors. So, all processors get at least $t+1$ votes for v in round r. By Lemma 6, any correct processor gets at most t votes for any value $v' \neq v$. Therefore, an arbitrary correct processor q sets $\text{VAL}_{r,q}$ to v in round r, broadcasts v in round $r+1$, gets at least $2t+1$ votes for v in round $r+1$, and decides v by round $r+1$.

Consensus condition: Let value v be the input to all of the correct processors. There are at least $2t+1$ correct processors that all broadcast v in round 1. All correct processors receive at least $2t+1$ votes for v in round 1 and therefore broadcast v in round 2. All correct processors receive at least $2t+1$ votes for v in round 2 and therefore decide v in round 2.

Plausibility condition: Let value v be the decision of a correct processor p. By Lemma 6, v is the persistent value. So, at least $2t+1$ processors (at least $t+1$ of which are correct) voted for v in round 1. Value v is the input to all of these correct processors. □

The communication cost of Protocol 2 is high because processors send messages for an unbounded number of rounds. This cost can be limited in two ways. In many applications (including Section 5) we are only interested in the results of an avalanche agreement protocol for a small fixed number of rounds. We can limit the communication cost by halting the protocol in the first round in which we are uninterested in its results. Alternatively, a simple coding convention for messages allows us to implement Protocol 2 so that at most $O(n^2 \cdot \log|V|)$ message bits are used in any execution. In Protocol 2 each correct processor broadcasts a non-null message each round. The convention gives a meaning to null messages. A processor that wishes to send the same message that it sent in the previous round instead sends the null message (at a cost of 0 bits). It is easy to show that using this convention each correct processor sends at most 3 non-null messages in any execution.

5. Compact Full-Information Protocols

In Section 3 we outlined the two-step process by which an arbitrary consensus protocol $\mathcal{P}$ is simulated by a communication-efficient canonical form protocol. The protocol $\mathcal{P}$ is first simulated by a full-information protocol that is, in turn, simulated by a compact full-information protocol. In Subsection 3.4 we showed in detail that the full-information protocol can simulate protocol $\mathcal{P}$. In this section we complete the description of the process. That is, we define the compact full-information protocol and prove that it simulates the full-information protocol.

5.1 Definitions

For any array a we make the following definition of the operation compress(a), which yields the set of all elements of a:

$$\text{compress}(a) = \begin{cases} \{a\} & \text{if } a \text{ is a 0-dimensional array;} \\ \bigcup_{i=1}^{n} \text{compress}(a_i) & \text{if } a = \langle a_1, \ldots, a_n \rangle. \end{cases}$$

An *index array* is an array of N. A *value array* is an array of V where V is the set of possible inputs to $\mathcal{P}$. In a full-information protocol, all messages sent by correct processors are value arrays and at each round the state of each correct processor is a value array.

A *partial* function may be undefined (denoted $\perp$) on some elements of its domain. We adopt the convention that any partial function used in this chapter is undefined whenever any of its arguments is undefined and that any array used in this chapter is undefined (equals $\perp$) whenever any of its elements is undefined. Partial function f is an *extension* of partial function g if for all x either $f(x) = g(x)$ or $g(x) = \perp$. A function f defined on arrays is *substitutive* if for all $a_1, \ldots, a_n$ the following holds: $f(\langle a_1, \ldots, a_n \rangle) = \langle f(a_1), \ldots, f(a_n) \rangle$.

When we simulate an arbitrary protocol $\mathcal{P}'$ by a compact full-information protocol the tradeoff between time and communication is determined by a parameter k. For any integer $k > 0$, there is a compact full-information protocol $\mathcal{P}$ that is structured as a series of blocks of $k + 2$ rounds. In each of the first k rounds of a block, $\mathcal{P}$ makes one round of progress in its simulation of $\mathcal{P}'$. The last two rounds are overhead—no progress is made.

We define some functions that relate various ways of numbering rounds. Let $r \geq 1$ be a round in a compact full-information protocol.

- block(r) $= \lceil r/(k+2) \rceil$ is the block of which round r is a part.
- prior(r) $=$ (block(r) $- 1) \cdot (k + 2)$ is 0 if round r is in the first block; otherwise, it is the last round prior to the current block.
- phase(r) $= r -$ prior(r) is the number of rounds since the start of the current block.

- simul(r) $= k \cdot (\text{block}(r) - 1) + \min(\text{phase}(r), k)$ is the number of rounds of progress that have currently been made in the simulation of the full-information protocol.

In Table 1, we illustrate the relationship among these quantities for 14 actual and 8 simulated rounds of a compact full-information protocol with parameter 2.

r	1	2	3	4	5	6	7	8	9	10	11	12	13	14
block(r)	1	1	1	1	2	2	2	2	3	3	3	3	4	4
prior(r)	0	0	0	0	4	4	4	4	8	8	8	8	12	12
phase(r)	1	2	3	4	1	2	3	4	1	2	3	4	1	2
simul(r)	1	2	2	2	3	4	4	4	5	6	6	6	7	8

Table 1: An Execution of 8 Simulated Rounds with $k = 2$

5.2 Subprotocols

Ordinary sequential programming problems are frequently decomposed into simpler subproblems using subroutines. In a similar way we simplify our compact full-information protocol by using avalanche agreement as a subprotocol. Subprotocols are similar to subroutines in that they help us decompose problems; however, they have different semantics. For example, our subprotocols run in parallel with the main protocol and their results take at least one round to become available. In this subsection we define the syntax and semantics that we will use for calls to subprotocols.

Recall that each round of any protocol $\mathcal{P}$ consists of three components that are performed in order: sending messages, receiving messages, and local state change. In the language we use to write our protocols there is no specific mechanism that ensures that this structure is followed; however, a quick inspection is generally sufficient to verify that it is. We will only write protocols that conform to this round structure.

We adopt the convention that if a call to the subprotocol SUB appears in round r of protocol $\mathcal{P}$ then the first round of SUB coincides with round r of $\mathcal{P}$. This implies that the call to SUB should appear in the text of $\mathcal{P}$ before any round r messages are sent and that all inputs to SUB must be available at the start of round r (*i.e.*, computed at round $r - 1$ at the latest). If processor p decides in SUB in the round that coincides with round r' of $\mathcal{P}$ then we make this answer available to processor p

in round r' of $\mathcal{P}$ before it computes its local state change. This simply means that in the local-state-change portion of each round of protocol $\mathcal{P}$ we perform all of the local state changes of the subprotocols before we perform the local state change of $\mathcal{P}$. Relative order among the subprotocols does not matter because they cannot interact—we have provided no mechanism for one subprotocol to name or access the internal variables of another subprotocol.

We require that all processors in protocol $\mathcal{P}$ initiate precisely the same subprotocols at precisely the same rounds. If in round r of protocol $\mathcal{P}$ there are x active subprotocols running then all round r messages are $(x+1)$-tuples—one component for each subprotocol and one component for $\mathcal{P}$.

Within a protocol $\mathcal{P}$, at an arbitrary correct processor p a call to subprotocol SUB is written

call SUB(input: IN, result: OUT, rounds: l)

where IN and OUT are local variables of processor p in $\mathcal{P}$ and where l is either an integer or ∞. (We require that all processors in $\mathcal{P}$ use the same value for l.) Let r be the round in which processor p executes the call statement. The variable IN must be defined by the end of round $r-1$. Subprotocol SUB is started with input IN in round r and is run forever if $l=\infty$; otherwise it is run for l rounds. The variable OUT initially has the value $\perp$. If processor p in subprotocol SUB decides v in round r' of $\mathcal{P}$ then the instance of variable OUT at processor p is set to v at the start of the local-state-change portion of round r' of $\mathcal{P}$. There is no requirement that processors in SUB eventually decide.

5.3 The Compact Full-Information Protocol

The code for the compact full-information protocol is given as Protocol 3. In the following discussion and proof we append two subscripts to each variable from Protocol 3. The first subscript, say r, is a natural number and the second subscript, say p, is in N. By this notation we mean the following. If $r \geq 1$ we mean the value of the subscripted variable at processor p at the *end* of round r. If $r=0$ we mean the value of the subscripted variable at processor p at the *start* of round 1. For example, $\text{OUT}_{q,b,r,p}$ is the value of variable $\text{OUT}_{q,b}$ at processor p at the end of round r. We let $\mathcal{A}_{i,b}$ denote the instance of the subprotocol AVALANCHE that is initiated in block b and for which each processor uses as input the current value of the variable VAL_i. In Protocol 3 the state of each correct processor consists of the

```
Initialization for processor $p$:
    CORE $\leftarrow$ the initial value of processor $p$
Code for processor $p$ in round $r$ where $b$ = block($r$):
1.      if phase($r$) $\leq k + 1$ then
2.          if $b = 1$ or phase($r$) $> 1$ then broadcast CORE else broadcast $p$
3.          receive MSG$_q$ from processor $q$ for $1 \leq q \leq n$
4.          for $i \leftarrow 1$ to $n$ do
5.              if $b = 1$ or compress(MSG$_i$) $\subset \{u \mid$ OUT$_{u,b} \neq \perp\}$
6.                  then VAL$_i \leftarrow$ MSG$_i$
7.                  else VAL$_i \leftarrow$ MSG$_p$
8.          if phase($r$) $\leq k$ then CORE $\leftarrow \langle$VAL$_1, \ldots,$ VAL$_n\rangle$
9.      if phase($r$) $= k + 2$ then
10.         for $i \leftarrow 1$ to $n$ do
11.             call AVALANCHE(input: VAL$_i$, result: OUT$_{i,b+1}$, rounds: $k + 3$)
```

Protocol 3: The Compact Full-Information Protocol

following variables: r, CORE, MSG$_i$ for $i \in N$, VAL$_i$ for $i \in N$, and OUT$_{i,b}$ for $i \in N$ and for $b \in \{2, \ldots, \text{block}(r) + 1\}$.

In the discussion of Protocol 3 we call MSG$_{q,r,p}$ the round r *message* from processor q to processor p, and we call VAL$_{q,r,p}$ the round r *corrected message* from processor q to processor p.

Protocol 3 is divided into blocks of $k + 2$ rounds. In each of the first k rounds of a block, one round of progress is made in the simulation of the full-information protocol. The last two rounds of each block are overhead—no progress is made in the simulation of the full-information protocol.

In most rounds—specifically in rounds 1 through k of block 1 and in rounds 2 through k of the other blocks—the compact full-information protocol closely resembles the full-information protocol. In parallel with any subprotocol calls that may be in progress, each correct processor broadcasts CORE (step 2), receives (step 3) and possibly corrects (steps 5–7) a message from each processor, and forms (step 8) its new CORE as the ordered collection of corrected messages received. The correction operation in steps 5–7 only changes messages sent by faulty processors; this fact follows from Lemmas 13 and 16, which we will prove.

The last two rounds of an arbitrary block b are overhead. Say round r is the last round of block b. In round r the correct processors initiate n instances of the avalanche agreement subprotocol. Specifically, for all $q \in N$, the correct processors initiate avalanche agreement on the value of $\text{CORE}_{r-2,q}$. This is done as follows. In phase $k+1$ of block b processor q broadcasts $\text{CORE}_{r-2,q}$. An arbitrary correct processor p receives the message, stores it in $\text{MSG}_{q,r-1,p}$, stores a corrected copy of the message in $\text{VAL}_{q,r-1,p}$, and then, in phase $k+2$, initiates $\mathcal{A}_{q,b}$ with $\text{VAL}_{q,r-1,p}$ as input. The answer produced by $\mathcal{A}_{q,b}$ is stored in the variable $\text{OUT}_{q,b+1}$. Thus, for all rounds r', $\text{OUT}_{q,b+1,r',p}$ is the round r' estimate that processor p makes of the value of $\text{CORE}_{r-2,q}$. Round $r+1$ is the first round of the next block after block b. If processor q is correct, then (as we will show in Lemma 17) $\text{OUT}_{q,b+1,r+1,p}$ is equal to $\text{CORE}_{r-2,q}$. So, at the start of the next block after block b, the estimate that processor p makes of the value of $\text{CORE}_{r-2,q}$ is equal to $\text{CORE}_{r-2,q}$.

We can now describe the first round of arbitrary block b where $b \geq 2$. In parallel with the subprotocol calls that are in progress, an arbitrary correct processor p broadcasts p (step 2), receives (step 3) and possibly corrects (steps 5–7) a message from each processor, and forms (step 8) its new CORE as the ordered collection of corrected messages received. The message p is intended to be a reference to the answer produced by $\mathcal{A}_{p,b-1}$. We can now see why the compact full-information protocol is communication efficient. The saving in message size results from using references to the output of various avalanche agreement protocols instead of using the entire text of the answer itself.

Let r be any round and let p be any correct processor. Intuitively speaking, the value of $\text{CORE}_{r,p}$, together with all of the current results of avalanche agreement available at processor p at round r, constitute a compact representation of the round simul(r) state of processor p in the simulated execution of the full-information protocol. The full state of processor p in the simulated execution can be reconstructed using the expansion functions which we will define in the next subsection. Intuitively, the expansion functions operate by replacing references by the arrays to which they refer.

5.4 Introduction to the Proof of Simulation

We define several *expansion functions* based on the states of the processors—in particular, based on the results of the various avalanche agreement subprotocols that have been run. The round r expansion functions of processor p are denoted $\phi_{b,r,p}$, for $b \in \{1, \ldots, \text{block}(r)\}$. If $b > 1$ then $\phi_{b,r,p}$ is a substitutive partial function from index arrays to value arrays. If $b = 1$ then $\phi_{b,r,p}$ is the identity function on value arrays, which is also substitutive. Because expansion functions are substitutive, it is sufficient to define them on indices and values. We let $\phi_{b,r,p}(x) = \bot$ if either $b = 1$ and $x \notin V$ or $b > 1$ and $x \notin N$; otherwise, $\phi_{b,r,p}$ is defined as follows:

$$\phi_{b,r,p}(x) = \begin{cases} x & \text{if } b = 1; \\ \phi_{b-1,r,p}(\textsc{out}_{x,b,r,p}) & \text{otherwise.} \end{cases}$$

We introduce some terminology that we will use when we discuss the expansion functions. If $\textsc{var}$ is a variable of processor p then $\textsc{var}_{r,p}$ is *b-expandable* if $\phi_{b,r,p}(\textsc{var}_{r,p}) \neq \bot$ and $\textsc{var}_{r,p}$ is *expandable* if it is b-expandable for $b = \text{block}(r)$.

For all $p \in N$ and for any processor state s define the function $f_p(s)$ to be s if s is an initial state; otherwise, define it to be $\phi_{\text{block}(r),r,p}(\textsc{core})$ where r and $\textsc{core}$ are contained in s. In Subsection 5.6 we will prove that the compact full-information protocol simulates the full-information protocol with simulation functions f_p for $p \in N$ and scaling function simul.

5.5 Technical Lemmas

In this subsection we prove some technical lemmas that we will use in the proof of our main simulation result in Subsection 5.6. We begin with a definition. For all correct processors p, for all rounds r, and for all $b \geq 2$ we let

$$\text{known}(b, r, p) = \{i \mid \textsc{out}_{i,b,r,p} \neq \bot\}.$$

In order to explain the importance of this definition we introduce some terminology. If $i \in N$ then we say that i is a *reference*. Reference i is *b-known* in round r at correct processor p if $i \in \text{known}(b, r, p)$. Reference i is *known* in round r at correct processor p if it is b-known for $b = \text{block}(r)$. We will show that if any variable $\textsc{var}_{r,p}$ contains only references that are b-known in round r at processor p then $\textsc{var}_{r,p}$ is b-expandable.

In Lemma 8 we prove that any reference that is b-known at correct processor p at round r is also b-known at processor p at round $r + 1$.

Lemma 8: *For all rounds r, for all blocks $b \geq 2$, and for all correct processors p, it holds that* $\text{known}(b, r, p) \subset \text{known}(b, r + 1, p)$.

Proof: Consider an arbitrary $i \in \text{known}(b, r, p)$. We have $\textsc{out}_{i,b,r,p} \neq \bot$ by the definition of $\text{known}(b, r, p)$. If $b = \text{block}(r)$ and $\text{phase}(r) \neq k+2$ then $\textsc{out}_{i,b,r+1,p} = \textsc{out}_{i,b,r,p}$ by the avalanche condition satisfied by $\mathcal{A}_{i,b-1}$; otherwise, $\textsc{out}_{i,b,r+1,p} = \textsc{out}_{i,b,r,p}$ because $\mathcal{A}_{i,b-1}$ is no longer running and the variable $\textsc{out}_{i,b}$ is therefore not being changed. Thus $\textsc{out}_{i,b,r+1,p} \neq \bot$ and so $i \in \text{known}(b, r + 1, p)$. $\Box$

In Lemma 9 we prove that any reference that is known at correct processor p at round r is also known at all correct processors at round $r+1$ as long as rounds r and $r+1$ are in the same block.

Lemma 9: *For all rounds r where $b = \text{block}(r)$ and for all correct processors p and q, if $\text{phase}(r) \neq k+2$ then $\text{known}(b,r,p) \subset \text{known}(b,r+1,q)$.*

Proof: Consider an arbitrary $i \in \text{known}(b,r,p)$. We have $\text{OUT}_{i,b,r,p} \neq \perp$ by the definition of $\text{known}(b,r,p)$. By the avalanche condition satisfied by $\mathcal{A}_{i,b-1}$ we have that $\text{OUT}_{i,b,r+1,q} = \text{OUT}_{i,b,r,p}$. Thus $\text{OUT}_{i,b,r+1,q} \neq \perp$ and so $i \in \text{known}(b,r+1,q)$. □

In Lemma 10 we prove that all of the references in all of the round r corrected messages at correct processor p are known at processor p in round r as long as the reference p was known at processor p in the first round of the current block.

Lemma 10: *For all rounds r where $b = \text{block}(r)$, for all correct processors p, and for all $q \in N$, if $b \geq 2$ and $p \in \text{known}(b, \text{prior}(r)+1, p)$ then $\text{compress}(\text{VAL}_{q,r,p}) \subset \text{known}(b,r,p)$.*

Proof: The proof is by induction on $\text{phase}(r)$.

Basis: ($\text{phase}(r) = 1$) From step 2 of the code $\text{MSG}_{p,r,p} = p$. Thus we have that $\text{compress}(\text{MSG}_{p,r,p}) \subset \text{known}(b,r,p)$. The effect of steps 5–7 of the code is that, if $\text{compress}(\text{MSG}_{q,r,p}) \subset \text{known}(b,r,p)$ then $\text{VAL}_{q,r,p} = \text{MSG}_{q,r,p}$; otherwise, $\text{VAL}_{q,r,p} = \text{MSG}_{p,r,p}$. In either event we have that $\text{compress}(\text{VAL}_{q,r,p}) \subset \text{known}(b,r,p)$.

Induction: $\text{MSG}_{p,r,p} = \langle \text{VAL}_{1,r-1,p}, \ldots, \text{VAL}_{n,r-1,p} \rangle$ because $\text{phase}(r) > 1$. So, we can calculate that

$$\begin{aligned} \text{compress}(\text{MSG}_{p,r,p}) &\subset \text{known}(b,r-1,p) && \text{By the induction hypothesis.} \\ &\subset \text{known}(b,r,p) && \text{By Lemma 8.} \end{aligned}$$

The effect of steps 5–7 of the code is that, if $\text{compress}(\text{MSG}_{q,r,p}) \subset \text{known}(b,r,p)$ then $\text{VAL}_{q,r,p} = \text{MSG}_{q,r,p}$; otherwise, $\text{VAL}_{q,r,p} = \text{MSG}_{p,r,p}$. In either event we have that $\text{compress}(\text{VAL}_{q,r,p}) \subset \text{known}(b,r,p)$. □

In Lemma 11 we prove that all of the references in all of the round r corrected messages at correct processor p are known at processor p in round r.

Lemma 11: *For all rounds r where $b = \text{block}(r)$, for all correct processors p, and for all $q \in N$, if $b \geq 2$ then $\text{compress}(\text{VAL}_{q,r,p}) \subset \text{known}(b,r,p)$.*

Proof: The proof is by induction on b.

Basis: ($b = 2$) Let $r' = \text{prior}(r) + 1$. From step 2 of the code $\text{MSG}_{p,r'-2,s} = \text{CORE}_{r'-3,p}$ for all correct processors s. By the test at step 5 of the code $\text{VAL}_{p,r'-2,s} = \text{CORE}_{r'-3,p}$ for all correct processors s because $\text{block}(r'-2) = 1$. Thus $\text{OUT}_{p,b,r',p} = \text{CORE}_{r'-3,p}$ by the consensus condition satisfied by $\mathcal{A}_{p,b-1}$ and so $p \in \text{known}(b, r', p)$. The claim follows by Lemma 10.

Induction: Let $r' = \text{prior}(r) + 1$. $\text{CORE}_{r'-3,p} = \langle \text{VAL}_{1,r'-3,p}, \ldots, \text{VAL}_{n,r'-3,p} \rangle$ by step 8 of the code because $\text{phase}(r'-3) = k$. Let processor s be an arbitrary correct processor. We can calculate that

$$\begin{aligned} \text{compress}(\text{CORE}_{r'-3,p}) &\subset \text{known}(b-1, r'-3, p) && \text{By the induction hypothesis.} \\ &\subset \text{known}(b-1, r'-2, s) && \text{By Lemma 9.} \end{aligned}$$

From step 2 of the code $\text{MSG}_{p,r'-2,s} = \text{CORE}_{r'-3,p}$. By the test at step 5 of the code $\text{VAL}_{p,r'-2,s} = \text{MSG}_{p,r'-2,s}$ because $\text{compress}(\text{MSG}_{p,r'-2,s}) \subset \text{known}(b-1, r'-2, s)$. Thus $\text{OUT}_{p,b,r',p} = \text{CORE}_{r'-3,p}$ by the consensus condition satisfied by $\mathcal{A}_{p,b-1}$ and so $p \in \text{known}(b, r', p)$. The the claim follows by Lemma 10. □

In Lemma 12 we prove that any index array is expandable if it contains only known references.

Lemma 12: *For all index arrays a, for all rounds r, for all $b \geq 2$, and for all correct processors p, if* $\text{compress}(a) \subset \text{known}(b, r, p)$ *then* $\phi_{b,r,p}(a) \neq \perp$.

Proof: The proof is by induction on b.

Basis: ($b = 2$) Consider an arbitrary $x \in \text{compress}(a)$. We have that $\phi_{b,r,p}(x) = \phi_{1,r,p}(\text{OUT}_{x,2,r,p}) = \text{OUT}_{x,2,r,p}$ by the definition of $\phi_{b,r,p}$. $\text{OUT}_{x,2,r,p} \neq \perp$ because $x \in \text{known}(2, r, p)$. Thus $\phi_{b,r,p}(x) \neq \perp$. The claim follows by the substitutivity of $\phi_{b,r,p}$.

Induction: Consider an arbitrary $x \in \text{compress}(a)$. We have that $\phi_{b,r,p}(x) = \phi_{b-1,r,p}(\text{OUT}_{x,b,r,p})$ by the definition of $\phi_{b,r,p}$. $\text{OUT}_{x,b,r,p} \neq \perp$ because $x \in \text{known}(b, r, p)$. By the plausibility condition satisfied by $\mathcal{A}_{x,b-1}$, $\text{VAL}_{x,\text{prior}(r)-1,q} = \text{OUT}_{x,b,r,p}$ for some correct processor q. We now calculate that

$$\begin{aligned} \text{compress}(\text{OUT}_{x,b,r,p}) &\subset \text{known}(b-1, \text{prior}(r)-1, q) && \text{By Lemma 11.} \\ &\subset \text{known}(b-1, \text{prior}(r), p) && \text{By Lemma 9.} \\ &\subset \text{known}(b-1, r, p) && \text{By Lemma 8.} \end{aligned}$$

By the induction hypothesis $\phi_{b-1,r,p}(\text{OUT}_{x,b,r,p}) \neq \perp$. Thus $\phi_{b,r,p}(x) \neq \perp$. The claim follows by the substitutivity of $\phi_{b,r,p}$. □

In Lemma 13 we prove that all corrected messages at all of the correct processors are expandable. This is a key lemma that we will use often in the rest of the chapter.

Lemma 13: *For all rounds r where $b =$ block(r), for all correct processors p, and for all $q \in N$, it holds that $\phi_{b,r,p}(\text{VAL}_{q,r,p}) \neq \perp$.*

Proof: If $b = 1$ then $\phi_{b,r,p}(\text{VAL}_{q,r,p}) \neq \perp$ because $\phi_{b,r,p}$ is the identity function. If $b \geq 2$ then compress$(\text{VAL}_{q,r,p}) \subset$ known(b, r, p) by Lemma 11, and, by Lemma 12, $\phi_{b,r,p}(\text{VAL}_{q,r,p}) \neq \perp$. □

In the next two lemmas we show some important properties of the expansion functions.

Lemma 14: *For all rounds r, for all $b \geq 1$, and for all correct processors p, it holds that $\phi_{b,r+1,p}$ is an extension of $\phi_{b,r,p}$.*

Proof: The proof is by induction on b.

Basis: ($b = 1$) The result follows immediately because $\phi_{b,r+1,p}$ and $\phi_{b,r,p}$ are both the identity function.

Induction: Consider an arbitrary $i \in N$. If $\phi_{b,r,p}(i) = \perp$ then the claim is trivially true, so assume that $\phi_{b,r,p}(i) \neq \perp$. We wish to show that $\phi_{b,r+1,p}(i) = \phi_{b,r,p}(i)$.

By assumption $\phi_{b,r,p}(i) \neq \perp$. $\phi_{b,r,p}(i) = \phi_{b-1,r,p}(\text{OUT}_{i,b,r,p})$ by the definition of $\phi_{b,r,p}$. Thus, $\text{OUT}_{i,b,r,p} \neq \perp$. If $b =$ block(r) and phase$(r) \leq k+1$ then $\text{OUT}_{i,b,r+1,p} = \text{OUT}_{i,b,r,p}$ by the avalanche condition satisfied by $\mathcal{A}_{i,b-1}$; otherwise, $\text{OUT}_{i,b,r+1,p} = \text{OUT}_{i,b,r,p}$ because $\mathcal{A}_{i,b-1}$ is no longer running and the variable $\text{OUT}_{i,b}$ is therefore not being changed. Using the fact that $\phi_{b,r,p}(i) \neq \perp$, we now calculate that

$$\begin{aligned}
\phi_{b,r,p}(i) &= \phi_{b-1,r,p}(\text{OUT}_{i,b,r,p}) && \text{By the definition of } \phi_{b,r,p}. \\
&= \phi_{b-1,r,p}(\text{OUT}_{i,b,r+1,p}) && \text{Because } \text{OUT}_{i,b,r+1,p} = \text{OUT}_{i,b,r,p}. \\
&= \phi_{b-1,r+1,p}(\text{OUT}_{i,b,r+1,p}) && \text{By the induction hypothesis.} \\
&= \phi_{b,r+1,p}(i) && \text{By the definition of } \phi_{b,r+1,p}.
\end{aligned}$$
□

Lemma 15: *For all rounds r where $b =$ block(r) and for all correct processors p and q, if* phase$(r) \neq k+2$ *then $\phi_{b,r+1,p}$ is an extension of $\phi_{b,r,q}$.*

Proof: The proof is by induction on b.

Basis: ($b = 1$) The result follows immediately because $\phi_{b,r+1,p}$ and $\phi_{b,r,q}$ are both the identity function.

Induction: Consider an arbitrary $i \in N$. If $\phi_{b,r,q}(i) = \bot$ then the claim is trivially true, so assume that $\phi_{b,r,q}(i) \neq \bot$. We wish to show that $\phi_{b,r+1,p}(i) = \phi_{b,r,q}(i)$.

By assumption $\phi_{b,r,q}(i) \neq \bot$. $\phi_{b,r,q}(i) = \phi_{b-1,r,q}(\text{OUT}_{i,b,r,q})$ by the definition of $\phi_{b,r,q}$. Thus, $\text{OUT}_{i,b,r,q} \neq \bot$ and $\text{OUT}_{i,b,r+1,p} = \text{OUT}_{i,b,r,q}$ by the avalanche condition satisfied by $\mathcal{A}_{i,b-1}$. Let $r' = \text{prior}(r)$. By the plausibility condition satisfied by $\mathcal{A}_{i,b-1}$ there is some correct processor s such that $\text{VAL}_{i,r'-1,s} = \text{OUT}_{i,b,r,q}$. Let $x = \phi_{b-1,r'-1,s}(\text{VAL}_{i,r'-1,s})$. By Lemma 13, $x \neq \bot$. We can now calculate that

$$\begin{aligned} x &= \phi_{b-1,r'-1,s}(\text{OUT}_{i,b,r,q}) && \text{Because } \text{OUT}_{i,b,r,q} = \text{VAL}_{i,r'-1,s}. \\ &= \phi_{b-1,r',q}(\text{OUT}_{i,b,r,q}) && \text{By the induction hypothesis.} \\ &= \phi_{b-1,r,q}(\text{OUT}_{i,b,r,q}) && \text{By Lemma 14.} \\ &= \phi_{b,r,q}(i) && \text{By the definition of } \phi_{b,r,q}. \end{aligned}$$

Similarly we calculate that

$$\begin{aligned} x &= \phi_{b-1,r'-1,s}(\text{OUT}_{i,b,r+1,p}) && \text{Because } \text{OUT}_{i,b,r+1,p} = \text{VAL}_{i,r'-1,s}. \\ &= \phi_{b-1,r',p}(\text{OUT}_{i,b,r+1,p}) && \text{By the induction hypothesis.} \\ &= \phi_{b-1,r+1,p}(\text{OUT}_{i,b,r+1,p}) && \text{By Lemma 14.} \\ &= \phi_{b,r+1,p}(i) && \text{By the definition of } \phi_{b,r+1,p}. \end{aligned}$$

Thus $\phi_{b,r+1,p}(i) = \phi_{b,r,q}(i)$, which is what we sought to show. □

In Lemma 16 we prove that if the round r message from processor q to correct processor p is expandable then the round r corrected message from processor q to processor p is equal to the round r message from processor q to processor p.

Lemma 16: *For all rounds r where $b = \text{block}(r)$, for all correct processors p, and for all $q \in N$, if $\phi_{b,r,p}(\text{MSG}_{q,r,p}) \neq \bot$ then $\text{VAL}_{q,r,p} = \text{MSG}_{q,r,p}$.*

Proof: If $b = 1$ then the condition in step 5 is satisfied and $\text{VAL}_{q,r,p}$ is set equal to $\text{MSG}_{q,r,p}$ in step 6. Suppose instead that $b \geq 2$. Because $\phi_{b,r,p}(\text{MSG}_{q,r,p}) \neq \bot$ it is immediate that $\text{compress}(\text{MSG}_{q,r,p}) \subset \text{known}(b, r, p)$. Thus the condition in step 5 is satisfied and $\text{VAL}_{q,r,p}$ is set equal to $\text{MSG}_{q,r,p}$ in step 6. □

In Lemma 17 we prove an important property about the expansion of references in the first round of each block after the first. This result explains why, in certain rounds, it is sensible for a processor to broadcast its own index in step 2 of the code.

Lemma 17: *For all rounds r where $b = \text{block}(r)$ and for all correct processors p and q, if* $\text{phase}(r) = 1$ *and* $b \geq 2$ *then* $\text{OUT}_{q,b,r,p} = \text{CORE}_{r-3,q}$.

Proof: By Lemma 13, $\phi_{b-1,r-3,q}(\text{VAL}_{u,r-3,q}) \neq \perp$ for all $u \in N$. By the substitutivity of $\phi_{b-1,r-3,q}$ and by step 8 of the code, $\phi_{b-1,r-3,q}(\text{CORE}_{r-3,q}) \neq \perp$. Let processor s be an arbitrary correct processor. $\text{MSG}_{q,r-2,s} = \text{CORE}_{r-3,q}$ from step 2 of the code. By Lemma 15, $\phi_{b-1,r-2,s}(\text{MSG}_{q,r-2,s}) \neq \perp$. So, $\text{VAL}_{q,r-2,s} = \text{CORE}_{r-3,q}$ by Lemma 16. Thus each correct processor uses $\text{CORE}_{r-3,q}$ as its input to $\mathcal{A}_{q,b-1}$. By the consensus condition $\text{OUT}_{q,b,r,p} = \text{CORE}_{r-3,q}$, which is what we sought to show. □

5.6 Proof of Simulation

Theorem 18: *The compact full-information protocol simulates the full-information protocol with simulation functions f_p for $p \in N$ and scaling function* simul.

Proof: We must show that for any execution $E = (F, I, H)$ of the compact full-information protocol, there is an execution $E' = (F, I, H')$ of the full-information protocol with $f_p(\text{state}(p, r, E)) = \text{state}(p, \text{simul}(r), E')$ for any correct processor p and for any $r \geq 0$.

The proof is in two stages. First we construct the execution E'. Second we show that it has the desired property.

Construction of the execution E': The execution $E' = (F, I, H')$ is completely specified except for H'. We now specify H', the history of messages sent by faulty processors in the execution E'. For all $r \geq 1$ where $\text{phase}(r) \leq k$, for all correct processors p, and for all faulty processors q we specify that $H'(q, \text{simul}(r), p) = \phi_{\text{block}(r),r,p}(\text{VAL}_{q,r,p})$. This message is a value array by Lemma 13. Thus the execution E' is well defined.

Verification that the execution E' has the desired property: We show that $f_p(\text{state}(p, r, E)) = \text{state}(p, \text{simul}(r), E')$ for any correct processor p and for any $r \geq 0$. The proof is by induction on r.

Basis: ($r = 0$) This follows immediately because correct processors have the same initial states in executions E and E' and because f_p is the identity function when applied to the initial states of the compact full-information protocol.

Induction: If phase$(r) > k$ then simul(r) = simul$(r-1)$ and the result follows immediately from Lemma 14 and the induction hypothesis. Suppose instead that phase$(r) \leq k$. Let b = block(r) and let $\langle v_1, \ldots, v_n \rangle$ = state$(p, \text{simul}(r), E')$. The calculations in the rest of the proof rely on the fact that $v_q \neq \perp$ for all $q \in N$. This fact follows because the execution E' is well defined.

We claim $v_q = \phi_{b,r,p}(\text{VAL}_{q,r,p})$ for all $q \in N$. The proof of the claim has four cases.

Case 1: (Processor q is faulty.) $v_q = \phi_{b,r,p}(\text{VAL}_{q,r,p})$ by the specification of execution E'.

Case 2: (Processor q is correct and $r = 1$.) We calculate the value of v_q as follows:

$v_q = \text{state}(q, \text{simul}(r) - 1, E')$	Because processors p and q are correct.
$= f_q(\text{state}(q, r-1, E))$	By the induction hypothesis.
$= \text{CORE}_{r-1,q}$	By the definition of f_q.
$= \phi_{b,r,p}(\text{CORE}_{r-1,q})$	Because $\phi_{b,r,p}$ is the identity function.
$= \phi_{b,r,p}(\text{MSG}_{q,r,p})$	$\text{MSG}_{q,r,p} = \text{CORE}_{r-1,q}$ from step 2 of the code.
$= \phi_{b,r,p}(\text{VAL}_{q,r,p})$	By Lemma 16.

Case 3: (Processor q is correct and phase$(r) > 1$.) We calculate the value of v_q as follows:

$v_q = \text{state}(q, \text{simul}(r) - 1, E')$	Because processors p and q are correct.
$= f_q(\text{state}(q, r-1, E))$	By the induction hypothesis.
$= \phi_{b,r-1,q}(\text{CORE}_{r-1,q})$	By the definition of f_q.
$= \phi_{b,r,p}(\text{CORE}_{r-1,q})$	By Lemma 15.
$= \phi_{b,r,p}(\text{MSG}_{q,r,p})$	$\text{MSG}_{q,r,p} = \text{CORE}_{r-1,q}$ from step 2 of the code.
$= \phi_{b,r,p}(\text{VAL}_{q,r,p})$	By Lemma 16.

Case 4: (Processor q is correct and phase$(r) = 1$ and $r > 1$.) We calculate the value of v_q as follows:

$v_q = \text{state}(q, \text{simul}(r) - 1, E')$	Because processors p and q are correct.
$= f_q(\text{state}(q, r-3, E))$	By the induction hypothesis.
$= \phi_{b-1,r-3,q}(\text{CORE}_{r-3,q})$	By the definition of f_q.

$= \phi_{b-1,r-2,p}(\text{CORE}_{r-3,q})$	By Lemma 15.
$= \phi_{b-1,r,p}(\text{CORE}_{r-3,q})$	By Lemma 14.
$= \phi_{b-1,r,p}(\text{OUT}_{q,b,r,p})$	By Lemma 17.
$= \phi_{b,r,p}(q)$	By the definition of $\phi_{b,r,p}$.
$= \phi_{b,r,p}(\text{MSG}_{q,r,p})$	$\text{MSG}_{q,r,p} = q$ from step 2 of the code.
$= \phi_{b,r,p}(\text{VAL}_{q,r,p})$	By Lemma 16.

Having proved the claim that $v_q = \phi_{b,r,p}(\text{VAL}_{q,r,p})$ for all $q \in N$, we now conclude the induction step of the proof by calculating that

$f_p(\text{state}(p, r, E))$

$= \phi_{b,r,p}(\text{CORE}_{r,p})$	By the definition of f_p.
$= \phi_{b,r,p}\langle \text{VAL}_{1,r,p}, \ldots, \text{VAL}_{n,r,p}\rangle$	From step 8 of the code.
$= \langle \phi_{b,r,p}(\text{VAL}_{1,r,p}), \ldots, \phi_{b,r,p}(\text{VAL}_{n,r,p})\rangle$	By the substitutivity of $\phi_{b,r,p}$.
$= \langle v_1, \ldots, v_n\rangle$	By the claim.
$= \text{state}(p, \text{simul}(r), E')$	By assumption. $\square$

5.7 Performance Analysis

Corollary 19: *For any $\epsilon > 0$, the agreement problem can be solved in $(1+\epsilon)(t+1)$ rounds using $O(t \cdot n^{\lceil 2/\epsilon \rceil + 3} \cdot \log|V|)$ message bits.*

Proof: Let protocol $\mathcal{P}'$ be the agreement protocol of Lamport *et al.* [34]. Protocol $\mathcal{P}'$ is a $(t+1)$-round exponential-message agreement protocol. Let γ'_p, for $p \in N$, be the decision functions used in protocol $\mathcal{P}'$. By Theorem 4, the full-information protocol simulates protocol $\mathcal{P}'$ with some simulation functions, say f_p, for $p \in N$. By Theorem 18, the compact full-information protocol simulates the full-information protocol with some simulation functions, say f'_p, for $p \in N$. Let protocol $\mathcal{P}$ be the compact full-information protocol with arbitrary correct processor p using decision function $\gamma_p = \gamma'_p \circ f_p \circ f'_p$. By Theorem 3, protocol $\mathcal{P}$ is an agreement protocol. We now analyze the performance of protocol $\mathcal{P}$.

In each of the first k rounds of a block of protocol $\mathcal{P}$, one round of progress is made in the simulation of protocol $\mathcal{P}'$. In the last two rounds, no progress is made. Therefore, in x actual rounds, protocol $\mathcal{P}$ has simulated at least $\frac{k}{k+2}x$ rounds of protocol $\mathcal{P}'$, for all x. In order for our agreement protocol to terminate within $(1+\epsilon)(t+1)$ actual rounds, we require that $(k+2)/k \leq 1+\epsilon$. Solving for the minimum integer k we get $k = \lceil 2/\epsilon \rceil$. Therefore, to achieve agreement in

$(1+\epsilon)(t+1)$ rounds, we run protocol $\mathcal{P}$ with parameter $k = \lceil 2/\epsilon \rceil$ and with arbitrary processor p using decision function γ_p.

The communication cost of protocol $\mathcal{P}$ consists of the cost of avalanche agreement and the cost of the remainder of the protocol. In the non-avalanche portion of the protocol, in each of $O(t)$ rounds each processor broadcasts a message of size $O(n^k \cdot \log |V|)$ for a total cost of $O(t \cdot n^{k+2} \cdot \log |V|)$ bits. This cost is dominated by the cost of avalanche agreement. In the avalanche agreement portion of the protocol, in each of $O(t)$ rounds, each processor broadcasts at most n messages of size $O(n^k \cdot \log |V|)$ for a total of $O(t \cdot n^{k+3} \cdot \log |V|)$. Expressed in terms of ϵ, this communication complexity is $O(t \cdot n^{\lceil 2/\epsilon \rceil + 3} \cdot \log |V|)$ message bits. In this analysis we made use of the fact (which we have not proved) that it is possible to adopt a representation for the index arrays used in the compact full-information protocol that allows each index to be stored in a constant number of bits. □

If $n \geq 4t + 1$ then a modification of our technique can simulate any $(t+1)$-round consensus protocol by a $(1+\epsilon)(t+1)$-round protocol that uses $O(t \cdot n^{\lceil 1/\epsilon \rceil + 3} \cdot \log |V|)$ message bits. Given that $n \geq 4t + 1$ it is possible to solve a variant of the avalanche agreement problem with a consensus condition modified to require a decision in one round rather than two. Using this variant avalanche agreement protocol, we can reduce the number of rounds in each block of a compact full-information protocol by one. Analyzing the new compact full-information protocol gives the total communication cost of $O(t \cdot n^{\lceil 1/\epsilon \rceil + 3} \cdot \log |V|)$ message bits.

We compare the cost (*i.e.*, rounds and message bits) of our agreement protocol with the cost of the protocol of Srikanth and Toueg [46], which uses the smallest number of rounds of any previously known protocol. The protocol of Srikanth and Toueg uses $2t + 1$ rounds and $O(t \cdot n^2 \cdot \log n \cdot \log |V|)$ message bits. Both protocols require that $n \geq 3t + 1$. The performance of our protocol for various values of ϵ is given in Table 2.

ϵ	Rounds	Message Bits $(n \geq 4t+1)$	Message Bits $(4t \geq n \geq 3t+1)$
1	$2 \cdot (t+1)$	$O(t \cdot n^4 \cdot \log |V|)$	$O(t \cdot n^5 \cdot \log |V|)$
$\frac{1}{2}$	$1\frac{1}{2} \cdot (t+1)$	$O(t \cdot n^5 \cdot \log |V|)$	$O(t \cdot n^7 \cdot \log |V|)$
$\frac{1}{3}$	$1\frac{1}{3} \cdot (t+1)$	$O(t \cdot n^6 \cdot \log |V|)$	$O(t \cdot n^9 \cdot \log |V|)$

Table 2: Communication Cost of Our New Agreement Protocol

We find that when $\epsilon = 1$ our protocol uses about the same number of rounds and more messages bits than the protocol of Srikanth and Toueg; however, for smaller values of ϵ, our protocol uses a number of rounds that is unattainable with the protocol of Srikanth and Toueg. Also, our technique is more general and may therefore have greater applicability (*e.g.,* reducing the communication cost of the approximate agreement protocol of Fekete [25]). A significant limitation of our technique is the large amount of local computation that it requires. By contrast the protocol of Srikanth and Toueg uses a small amount of space and time locally at each processor. In this comparison we ignore a possible optimization due to Dolev *et al.* [18] and another due to Perry [41] and to Turpin and Coan [47] because these optimizations have a similar impact on both protocols.

3

Two Applications of the Avalanche Agreement Protocol

We give two applications of avalanche agreement. They are a general technique for constructing simple and efficient solutions to the multivalued agreement problem and a simple solution to the crusader agreement problem. These two applications, in addition to being interesting in their own right, are intended to show the usefulness of an avalanche agreement protocol in the modular construction of consensus protocols.

1. Introduction

Making use of the avalanche agreement protocol defined in Chapter 2, we give a general technique for constructing simple and efficient solutions to the multivalued agreement problem and we give a simple solution to the crusader agreement problem.

The material in this chapter depends on the following portions of Chapter 2: the definition of l-boundedness, simultaneity, and termination (Subsection 3.1); the definition of the agreement problem (Section 2); the formulation and the solution of the avalanche agreement problem (Section 4); and the syntax and the semantics of subprotocols (Subsection 5.2).

Sections 1 and 2 of this chapter are joint work with Russell Turpin and were published in a different form in *Information Processing Letters* [47].

We make the following two additional definitions. The *binary agreement problem* is the agreement problem with the input set restricted to be of size two. The *multivalued agreement problem* is the agreement problem with no restrictions on the size of the input set.

All of the protocols that we consider in this chapter function correctly in the Byzantine fault model. Thus they also function correctly *a fortiori* in more benign fault models.

We show how to construct a multivalued agreement protocol using an arbitrary binary agreement protocol and an arbitrary avalanche agreement protocol as subprotocols. The resulting protocol is simple, has an easy correctness proof, and can have a low communication cost. Its cost is a function of the particular binary agreement protocol and avalanche agreement protocol used; there are choices that give the best known message complexity for a multivalued agreement protocol. A similar construction was discovered independently by Perry [41].

We also show how to construct a crusader agreement protocol using an arbitrary avalanche agreement protocol as a subprotocol. In performance and behavior, our protocol is very close to an earlier crusader agreement protocol by Dolev [16]. We believe that our contribution is not the protocol itself; rather, it is the modular method of constructing the protocol using an avalanche agreement protocol as a subprotocol. Because of its method of construction, our crusader agreement protocol is simple and has an easy correctness proof.

In Section 2 we construct a multivalued agreement protocol and analyze its performance. In Section 3 we define the crusader agreement problem and give a crusader agreement protocol.

2. The Multivalued Agreement Problem

In this section we give a general technique for using an arbitrary binary agreement protocol and an arbitrary avalanche agreement protocol to construct a multivalued agreement protocol. The new multivalued agreement protocol is essentially just a series of two subprotocol calls, one to the avalanche agreement protocol and one to the binary agreement protocol. The communication cost of the multivalued agreement protocol is a function of the cost of the two subprotocols. Specifically, the worst-case number of rounds used by the multivalued agreement protocol is two more than the worst-case number of rounds used by the binary agreement subprotocol and the worst-case number of message bits sent by the multivalued agreement protocol is the sum of the worst-case number of message bits sent by the two sub-protocols.

Let V be some set such that $|V| \geq 2$. Let DEFAULT be an arbitrary element of V. Let BINARY be an arbitrary binary agreement protocol. Without loss of generality we assume that the input set of BINARY is $\{0,1\}$. Let AVALANCHE be an arbitrary avalanche agreement protocol.

The multivalued agreement protocol with input set V is given as Protocol 1. All processors run the same code. For convenience we describe the code run by an arbitrary correct processor p. Processor p stores its input in the variable AA_IN. In round 1 processor p initiates avalanche agreement with input AA_IN. It sets BA_IN to 1 if the avalanche agreement protocol decides in two rounds; otherwise, it sets BA_IN to 0. In round 3 it initiates the binary agreement protocol with input BA_IN. Eventually the binary agreement protocol terminates at processor p. If the answer is 0 then processor p answers DEFAULT; otherwise, processor p decides on the value that is the output from the avalanche agreement protocol. We will prove that such a value exists whenever the decision from the binary agreement protocol is 1.

```
Initialization for processor $p$:
    AA_IN $\leftarrow$ the initial value of processor $p$
Code for processor $p$ in round $r$:
    if $r = 1$ then
        call AVALANCHE(input: AA_IN, result: AA_OUT, rounds: 3)
    if $r = 2$ then
        if AA_OUT $\neq \perp$ then BA_IN $\leftarrow$ 1 else BA_IN $\leftarrow$ 0
    if $r = 3$ then
        call BINARY(input: BA_IN, result: BA_OUT, rounds: $\infty$)
    if $r \geq 3$ then
        if BA_OUT $\neq \perp$ then
            if BA_OUT $= 1$ then decide AA_OUT else decide DEFAULT
```

Protocol 1: The Multivalued Agreement Protocol

In the following proof of Protocol 1 we append two subscripts to each variable from the protocol. The first subscript, say r, is a positive integer and the second subscript, say p, is in N. By this notation we mean the value of the subscripted variable at processor p at the *end* of round r. For example, $\text{AA\_IN}_{r,p}$ is the value of variable AA_IN at processor p at the end of round r.

Theorem 1: *Protocol 1 solves the multivalued agreement problem.*

Proof: We show that the agreement, validity, and termination conditions are satisfied.

Agreement condition: By the agreement and termination conditions satisfied by BINARY, there is some $b \in \{0,1\}$ and some $r \geq 3$ such that $\text{BA\_OUT}_{r,p} = b$ for all correct processors p. There are two cases, either $b = 0$ or $b = 1$. If $b = 0$ then all correct processors decide DEFAULT. The agreement condition is satisfied. If $b = 1$ then, by the validity condition satisfied by BINARY, there is some correct processor q such that $\text{BA\_IN}_{2,q} = 1$. From step 4 of the code, $\text{AA\_OUT}_{2,q} \neq \perp$. Consider arbitrary correct processor p. From step 9 of the code, processor p decides $\text{AA\_OUT}_{r',p}$ for some r' where $3 \leq r' \leq r$. By the avalanche condition satisfied by AVALANCHE, $\text{AA\_OUT}_{3,p} = \text{AA\_OUT}_{2,q}$. Clearly, $\text{AA\_OUT}_{r',p} = \text{AA\_OUT}_{3,p}$. Thus arbitrary correct processor p decides $\text{AA\_OUT}_{2,q}$ and the agreement condition is satisfied.

Validity condition: Let v be the input to all of the correct processors. By the consensus condition satisfied by AVALANCHE, $\text{AA\_OUT}_{2,p} = v$ for all correct processors p. Clearly, for all $r \geq 3$ and for all correct processors p, $\text{AA\_OUT}_{r,p} = \text{AA\_OUT}_{2,p}$. From step 4 of the code, $\text{BA\_IN}_{2,p} = 1$ for all correct processors p. Let processor p be an arbitrary correct processor. By the validity and termination conditions satisfied by BINARY, there is some $r \geq 3$ such that $\text{BA\_OUT}_{r,p} = 1$. From step 9 of the code, processor p decides v.

Termination condition: This follows immediately from the termination condition satisfied by BINARY. □

Theorem 2: *If* BINARY *is l-bounded for some l then Protocol 1 is $(l+2)$-bounded.*

Proof: All correct processors call BINARY in round 3. BINARY is l-bounded for some l. Thus all correct processors decide by round $l+2$ of Protocol 1 and so Protocol 1 is $(l+2)$-bounded. □

Theorem 3: *If* BINARY *is simultaneous then so is Protocol 1.*

Proof: All correct processors call BINARY in round 3. BINARY is simultaneous. Thus all correct processors decide in the same round in Protocol 1 and so Protocol 1 is simultaneous. □

In order to analyze the communication cost of Protocol 1 we make some definitions. For any protocol $\mathcal{P}$ we let bits($\mathcal{P}$) be the worst-case number of message bits sent by the correct processors in any execution of protocol $\mathcal{P}$ and we let rounds($\mathcal{P}$) be the worst-case number of rounds until the last correct processor decides in any execution of protocol $\mathcal{P}$. We extend the standard notation [31] for the asymptotic behavior of functions of one variable to functions of three variables. Let $\omega(f(x,y,z))$ denote the set of all $g(x,y,z)$ such that for all positive constants c there exists positive constants x_0, y_0, and z_0 with $g(x,y,z)/f(x,y,z) \geq c$ as long as $x \geq x_0$, $y \geq y_0$, and $z \geq z_0$. Let $o(f(x,y,z))$ denote the set of all $g(x,y,z)$ such that for all positive constants c there exists positive constants x_0, y_0, and z_0 with $|g(x,y,z)/f(x,y,z)| \leq c$ as long as $x \geq x_0$, $y \geq y_0$, and $z \geq z_0$. This generalization corresponds to making claims about functions when x, y, and z are all large.

The communication cost of Protocol 1 depends on the particular choice of subprotocols and on the input set. For the purpose of this analysis we assume that Protocol 1 uses the avalanche agreement protocol given in Section 4 of Chapter 2. Let $\mathcal{T}(\mathcal{P},V)$ denote the instance of Protocol 1 that has input set V and uses protocol $\mathcal{P}$ as its binary agreement subprotocol. It is easy to see for any binary agreement protocol $\mathcal{P}$ and for any set V that $\text{bits}(\mathcal{T}(\mathcal{P},V)) = \text{bits}(\mathcal{P}) + 3n^2 \cdot \log|V|$ and $\text{rounds}(\mathcal{T}(\mathcal{P},V)) = \text{rounds}(\mathcal{P}) + 2$. Using a variant of the relay-processor optimization due to Dolev and Strong [21] it is possible to improve this to $\text{bits}(\mathcal{T}(\mathcal{P},V)) = \text{bits}(\mathcal{P}) + 3tn \cdot \log|V|$ and $\text{rounds}(\mathcal{T}(\mathcal{P},V)) = \text{rounds}(\mathcal{P}) + 2$.

We can view the results of this section as a general technique for transforming an arbitrary multivalued agreement protocol into one that has an improved message complexity but uses two extra rounds. For example, let $\mathcal{P}_V$ denote the multivalued agreement protocol of Lynch, Fischer, and Fowler [35] with input set V. Then $\text{bits}(\mathcal{P}_V) = (t^3 \cdot \log t + tn) \cdot \log|V|$ and $\text{rounds}(\mathcal{P}_V) = 2t + 5$. In contrast, $\text{bits}(\mathcal{T}(\mathcal{P}_{\{0,1\}},V)) = t^3 \cdot \log t + tn \cdot \log|V|$ and $\text{rounds}(\mathcal{T}(\mathcal{P}_{\{0,1\}},V)) = 2t + 7$. More generally, if $\mathcal{P}_V$ is any agreement protocol with input set V where

$$\text{bits}(\mathcal{P}_V) = \omega(\text{bits}(\mathcal{P}_{\{0,1\}}) + tn \cdot \log|V|)$$

then

$$\text{bits}(\mathcal{T}(\mathcal{P}_{\{0,1\}},V)) = o(\text{bits}(\mathcal{P}_V)).$$

Our claim for the generality of our technique—a claim that we believe defies formal proof—is based on the observation that every known multivalued agreement protocol, $\mathcal{P}_V$, which does not use some variant of our technique or the similar technique developed independently by Perry [41], has the property that $\text{bits}(\mathcal{P}_V) = \omega(\text{bits}(\mathcal{P}_{\{0,1\}}) + tn \cdot \log|V|)$. A limitation of our technique is that there is no improvement in performance if our transformation is applied a second time.

3. The Crusader Agreement Problem

The crusader agreement problem was first formulated and solved by Dolev [16]. For uniformity of style within this thesis, we adopt a formulation that is slightly different from his. In his formulation there is one processor, the "sender," that plays a special role; our formulation is symmetric. There are very easy simulation results between the two models.

A protocol that solves the crusader agreement problem operates under the same failure and communication assumptions as an agreement protocol. Each processor begins the protocol with an input value from some fixed set V. We assume that $* \notin V$. Let $A = V \cup \{*\}$. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on an element of A as its answer. There are three conditions that the correct processors must satisfy.

- *Agreement condition:* All correct processors that decide on an element of V reach the same decision.
- *Validity condition:* If all correct processors start the protocol with input v then v is the decision of all correct processors that decide.
- *Termination condition:* All correct processors eventually decide.

The crusader agreement protocol is given as Protocol 2. All processors run the same code. For convenience we describe the code run by an arbitrary correct processor p. Processor p stores its input in the variable IN. In round 1 processor p initiates avalanche agreement with input IN. In round 2 processor p decides v if v is the round 2 decision of the avalanche agreement subprotocol; otherwise, processor p decides $*$.

```
    Initialization for processor $p$:
        IN $\leftarrow$ the initial value of processor $p$
    Code for processor $p$ in round $r$:
1.      if $r = 1$ then
2.          call AVALANCHE(input: IN, result: OUT, rounds: 2)
3.      if $r = 2$ then
4.          if OUT $\neq \perp$ then decide OUT else decide $*$
```

Protocol 2: The Crusader Agreement Protocol

Theorem 4: *Protocol 2 solves the crusader agreement problem.*

Proof: We show that the agreement, validity, and termination conditions are satisfied.

Agreement condition: This follows immediately from the avalanche condition satisfied by AVALANCHE.

Validity condition: This follows immediately from the consensus condition satisfied by AVALANCHE.

Termination condition: It is apparent from steps 3–4 of the code that all correct processors decide in round 2 of Protocol 2. □

4 A Compiler that Increases the Fault-Tolerance of Asynchronous Protocols

We give a compiler that increases the fault-tolerance of certain asynchronous protocols. Specifically, it transforms a "source protocol" that is resilient to crash faults into an "object protocol" that is resilient to Byzantine faults. Our compiler can simplify the design of protocols for the Byzantine fault model because it enables us to break the design process into two steps. The first step is to design a protocol for the crash fault model. The second step, which is completely mechanical, is to compile the protocol into one for the Byzantine fault model. We use our compiler to produce a new asynchronous approximate agreement protocol that operates in the Byzantine fault model. Specifically, we design a new asynchronous approximate agreement protocol for the crash fault model and we observe that this protocol can be compiled into a protocol for the Byzantine fault-model. In the Byzantine fault model, the new protocol improves in several respects on the performance of the asynchronous approximate agreement protocol of Dolev, Lynch, Pinter, Stark, and Weihl.

1. Introduction

We give a compiler that transforms an arbitrary standard-form asynchronous protocol that tolerates crash faults into an asynchronous protocol that tolerates Byzantine faults and that solves the same problem as the original protocol. Our compiler incorporates communication primitives and a message validation scheme developed by Bracha [5]. Bracha argues informally that his tools restrict the disruptive behavior of a processor that fails with a Byzantine fault. He argues that the restricted behavior is that of a processor subject only to crash failures. He then uses these primitives to construct an interesting new protocol for the Byzantine fault model.

Our goal is similar to Bracha's. It is to simplify the design and proof of asynchronous protocols that are resilient to Byzantine faults. Specifically, our approach is as follows. We incorporate Bracha's communication primitives and message validation scheme in a compiler, which we prove correct. Then, we design and prove protocols correct in the crash fault model. It follows from the correctness of the compiler that the protocols that we design for the crash fault model can be compiled into protocols that operate correctly in the Byzantine fault model.

A limitation of our compiler is that it only works for deterministic protocols. It is an open question to construct and prove correct a compiler for randomized protocols. Because our compiler works only for deterministic protocols, it is not useful in the particular application that Bracha considers.

There seem to be two principal benefits of our approach. First, it is simpler to design and prove protocols in the crash model than it is to do the same in the Byzantine model. Using our method, only the compiler needs to be proved correct for the Byzantine model. Second, our approach is modular. For example, we give two versions of our compiler with slightly different performance tradeoffs. (The two versions of the compiler use slightly different communication primitives.) After we prove a protocol $\mathcal{P}$ correct in the crash fault model, we can use either version of the compiler to transform $\mathcal{P}$ into a protocol that is correct in the Byzantine fault model.

It should be clear that our compiler must change some of the properties of a protocol (like the kind of faults tolerated) and leave other properties unchanged (like the problem solved by the protocol). We use *correctness predicates* to formalize one of the properties that we would like our compiler to preserve. A correctness predicate is any predicate defined on the inputs to and answers of correct processors. We show that our compiler preserves the satisfaction of correctness predicates. We also show that our compiler preserves termination. Thus our compiler preserves the solution to any problem that can be formalized by a correctness predicate and a requirement that all correct processors eventually terminate. Agreement and approximate agreement are such problems.

Asynchronous systems are "harder" than synchronous ones because they can experience a superset of the executions of synchronous systems. If a protocol solves some problem in an asynchronous system, then it follows that the protocol solves the same problem in a synchronous system. Of course, this holds for any protocol that is the output of our compiler. The foregoing argument might lead one to think that our compiler is capable of transforming a synchronous protocol that solves some problem in the crash model into a synchronous protocol that solves the same problem in the Byzantine model. This is wrong. The difficulty is that our compiler relies on the fact that its source protocol operates correctly in asynchronous executions with crash faults. The property of asynchronous executions that our compiler relies on is that some messages between correct processors may be delivered very late.

A limitation of our technique is that we are unable to force a faulty processor to accurately report its input value. We have accommodated this limitation by requiring that correctness predicates not depend on the input to a faulty processor. It seems that this requirement does not substantially restrict the number of problems that can be formalized with correctness predicates.

Informally, we say that a protocol terminates in time r if all correct processors decide by time r in any execution in which the message-delivery time is bounded above by one. Our compiler imposes a certain overhead in the running time. There are two versions of the compiler. The first version increases the time by a factor of two and requires that the number of processors be more than four times the number of faults tolerated. The second version increases the time by a factor of three and requires that the number of processors be more than three times the number of faults tolerated. The first version of the compiler uses new communication primitives; the second version incorporates the communication primitives developed by Bracha. Both versions of the compiler substantially increase the number of message bits sent.

Our compiler requires that its "source protocols" be in a particular standard form. We believe that most asynchronous protocols can be put into this standard form. Nevertheless, it would be good to extend the compiler so that it could compile arbitrary source protocols. We believe that this can be done, but the unrestricted compiler and its proof seem very complicated. Because of their apparent complexity, we leave the construction and proof of the unrestricted compiler as an open problem.

Unfortunately, there are a limited number of known protocols that are potential source protocols for our compiler. The well-known impossibility result of Fischer, Lynch, and Paterson [28] shows that many problems have no deterministic solution in an asynchronous system. The only problems currently defined in the literature that can be solved with deterministic protocols in asynchronous systems are the approximate agreement problem [19] and the inexact agreement problem [36]. Despite the limited number of potential source protocols, we believe that our compiler is interesting because of the method that it embodies.

Using the two versions of our compiler, we produce a pair of new asynchronous protocols. Specifically, we develop two asynchronous approximate agreement protocols that are resilient to Byzantine faults and that improve in several respects on the asynchronous approximate agreement protocol of Dolev, Lynch, Pinter, Stark, and Weihl [19]. Our method is to design a new asynchronous approximate agreement protocol for the crash fault model and observe that this protocol can be compiled into a protocol for the Byzantine fault model (using either version of our compiler). The protocol that we design in the crash fault model uses many ideas developed by Dolev *et al.* for use in their protocol. (The approximate agreement problem is defined in Section 2.)

In the Byzantine fault model our new approximate agreement protocols tolerate a larger proportion of faulty processors than the protocol of Dolev *et al.* Their protocol requires that the number of processors be more than five times the number of faults tolerated. One version of our protocol requires that the number of processors be more than four times the number of faults tolerated; the other requires that the number of processors be more than three times the number of faults tolerated. The second version of our protocol has an optimal amount of redundancy. This follows because Fischer, Lynch, and Merritt [27] have shown that no protocol solves the approximate agreement problem unless the number of processors is more than three times the number of faults tolerated.

Intuitively, the *convergence rate* of an approximate agreement protocol is the factor by which the range of possible answers is reduced each unit of time. Despite the overhead introduced by the compiler, one of our approximate agreement protocols has an improved convergence rate for some (small) system sizes. Our improvement in the convergence rate does not contradict the proved optimality of the convergence rate of the protocol of Dolev *et al.* Their claim of optimality is for protocols of a particular form. The output of our compiler is not of that form.

The running time of either one of our approximate agreement protocols depends only on the inputs to the correct processors and the size of the system. In the protocol of Dolev *et al.* the faulty processors can choose the amount of time that will elapse before the correct processors decide.

We now give an outline of the remainder of the chapter. In Section 2 we define the approximate agreement problem. In Section 3 we give our model for asynchronous protocols that operate in either the crash fault model or the Byzantine fault model. In Section 4 we present our compiler and we prove that it works correctly. In Section 5 we give a new asynchronous approximate agreement protocol for the crash fault model and we prove it correct. We then note that this protocol can be compiled into a protocol for the Byzantine fault model.

2. The Approximate Agreement Problem

The approximate agreement problem is solved in various fault models including crash and Byzantine. The requirements given here apply to both of these fault models. In a protocol for the approximate agreement problem, each processor begins with some real number as its input. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on a real number as its answer. A parameter ϵ specifies the accuracy required in the solution. There are three conditions that the correct processors must satisfy in all executions.

- *Agreement condition:* If v and v' are the decisions of two correct processors, then $|v - v'| \leq \epsilon$.
- *Validity condition:* If v is the decision of some correct processor, then there are correct processors with inputs i and i' such that $i \leq v \leq i'$.
- *Termination condition:* All correct processors eventually decide.

3. The Model

We model processors as state machines that communicate by sending messages. In Subsection 3.1 we begin by defining some parameters that specify the size of the systems that we consider. In Subsection 3.2 we give our model for asynchronous protocols subject to crash faults. In Subsection 3.3 we define a subset of the possible executions in the crash fault model; we call them synchronous executions. In Subsection 3.4 we give our model for asynchronous protocols subject to Byzantine faults. In Subsection 3.5 we define correctness predicates as a way to formalize part of the definition of a problem in the asynchronous model. Finally, in Subsection 3.6 we define our time measure.

3.1 Definition of Parameters

For the remainder of this chapter, let n be the number of processors, let t be an upper bound on the number of processors that fail, and let $N = \{1, \ldots, n\}$. We define the *redundancy* to be $(n-1)/t$. For the remainder of this chapter we assume that the redundancy is at least 3 and we let ϵ be an arbitrary fixed value of the parameter to the approximate agreement problem.

Let $\mathcal{I}^+$ be the set of positive integers; let $\mathcal{N}$ be the set of natural numbers, including 0; and let $\mathcal{R}$ be the set of real numbers.

3.2 The Crash Fault Model

A processor is modeled as an infinite state machine with a message buffer. The message buffer—modeled as a multiset of messages—holds those messages that have been sent to the processor but not yet received. Messages in the message buffer are reliably tagged with the identity of the sending processor. In each step a processor receives a set containing at most one message from its buffer and (based on its transition function) sends a set of messages. The transition function of a processor uses the current state and current set of messages received to compute a new state and a set of messages to be sent. There is a fixed set V of possible inputs to the processors. For each element $v \in V$, each processor has one *initial state* that corresponds to having input v. The processors are indexed by the set N.

A *configuration* C is a vector of n states, one for each processor, and a vector of n multisets of messages, one for each message buffer. An *initial configuration* has all processors in initial states and all buffers equal to the empty multiset.

An *event* is denoted either (*step:* p) or (*receive:* p, q, m). The event (*step:* p) models processor p taking a step without receiving a message. The event $e =$ (*step:* p) is *applicable* to any configuration. The configuration resulting from applying event e to configuration C, denoted $e(C)$, is obtained from C by changing the state of processor p according to the transition function and adding messages from processor p to the appropriate buffers according to the transition function. The event (*receive:* p, q, m) models processor p receiving the message m from processor q. The event $e' =$ (*receive:* p, q, m) is *applicable* to configuration C if the message m (tagged with sender q) is an element of the buffer of processor p in configuration C. The configuration resulting from applying event e' to configuration C, denoted $e'(C)$, is obtained from C by removing the message m from the buffer of processor p, changing the state of processor p according to the transition function, and adding messages from processor p to the appropriate buffers according to the transition function.

A *schedule* is a finite or infinite sequence of events. A finite schedule $\sigma = e_1 e_2 \ldots e_k$ is *applicable* to configuration C if e_1 is applicable to C, e_2 is applicable to $e_1(C)$, *etc.* The resulting configuration is denoted $\sigma(C)$. An infinite schedule is applicable to configuration C if every finite prefix of the schedule is applicable to C.

An *execution* of a protocol is a triple (F, I, σ) where $I = \langle i_1, \ldots, i_n \rangle$ is a 1-dimensional array of V, where σ is applicable to the initial configuration in which an arbitrary processor p begins in the initial state that corresponds to having input i_p, where all processors in $N - F$ take an infinite number of steps in σ, and where every message that is sent to a processor that takes an infinite number of steps in σ is eventually delivered. A processor p is *faulty* in the execution (F, I, σ) if $p \in F$; otherwise, p is *correct.*

Each processor has a *decision function* that maps from processor states to $V \cup \{\perp\}$. In the first step in which the decision function of a processor applied to the current state is $v \in V$, we say that the processor *decides* v. After a processor has decided, it makes no further use of its decision function. An execution is a *deciding execution* if all correct processors eventually decide. A protocol *terminates* if all of its executions are deciding executions.

In our crash fault model, processors have an "atomic broadcast" capability. A processor can send several messages in one (atomic) step. Every message that is sent to a processor that takes an infinite number of steps is delivered. Compared to a crash fault model without atomic broadcast, the failures in our model are more benign. This increases the strength of our compilation result because it potentially makes it easier to write source protocols.

There will be no atomic broadcast capability in our Byzantine fault model.

3.3 Synchronous Executions in the Crash Fault Model

In this subsection we define what it means for an execution in the crash fault model to be synchronous. If $E = (F, I, \sigma)$ is an execution, C is the initial configuration in execution E, σ' is a prefix of σ, and $p \in N$, then we define deliver(E, σ', p) to be the lexicographically least schedule that delivers to processor p all of the messages that are in the buffer of p in the configuration $\sigma'(C)$. An execution $E = (F, I, \sigma)$ is *synchronous* if σ can be expressed as the concatenation of subschedules, $\sigma = \sigma_0\sigma_1\sigma_2\ldots$, where $\sigma_r = \alpha(r,1)\alpha(r,2)\ldots\alpha(r,n)$, and where $\alpha(r,p)$ satisfies one of the following three conditions for all $r \in \mathcal{N}$ and for all $p \in N$:

- *Condition 1:* $p \in F$ and $\alpha(r,p)$ is the empty sequence.

- *Condition 2:* $r = 0$ and $\alpha(r,p) = (\textit{step: } p)$. Recall that $(\textit{step: } p)$ is the event in which processor p takes a step without receiving any messages.

- *Condition 3:* $r \geq 1$ and $\alpha(r,p)$ is any reordering of the events in deliver(E, σ', p) where $\sigma' = \sigma_0\sigma_1 \ldots \sigma_{r-1}$.

In a synchronous execution of a protocol, we say that an event happens in *phase* r if it is in $\alpha(r,p)$ for some p.

3.4 The Byzantine Fault Model

The Byzantine fault model has much in common with the crash fault model. In this subsection we define only those parts of the Byzantine fault model that differ from the crash fault model. The two differences are in the definition of events and in the definition of executions.

An *event* is denoted either $(\textit{step: } p)$, $(\textit{receive: } p,q,m)$, or $(\textit{error: } p,q,m)$. The events $(\textit{step: } p)$ and $(\textit{receive: } p,q,m)$ are defined as they are in the crash fault model. The event $(\textit{error: } p,q,m)$ models processor q erroneously sending the message m to processor p. The event $e = (\textit{error: } p,q,m)$ is *applicable* to any configuration. The configuration resulting from applying event e to configuration C, denoted $e(C)$, is obtained from C by adding the message m (tagged with sender q) to the buffer of processor p.

An *execution* of a protocol is a triple (F,I,σ) where $I = \langle i_1, \ldots, i_n \rangle$ is a 1-dimensional array of V, where σ is applicable to the initial configuration in which an arbitrary processor p begins in the initial state that corresponds to having input i_p, where all processors in $N - F$ take an infinite number of steps in σ, where every message that is sent to a processor in $N - F$ is eventually delivered, where processors in $N - F$ take no error steps, and where processors in F take only error steps. A processor p is *faulty* in the execution (F,I,σ) if $p \in F$; otherwise, p is *correct*.

We remark that the requirement that a faulty processor take only error steps does not restrict the kinds of faults that can be exhibited in an execution. This is because any message sending pattern can be achieved with error steps and because we are never interested in examining the state of a faulty processor.

We could give a definition of "synchronous executions" in the the Byzantine fault model analogous to the definition of synchronous executions in the crash fault model. We omit that definition because we have no need to discuss such executions formally.

3.5 Correctness Predicates

In this subsection we define correctness predicates as a way to formalize part of a problem definition. The definitions in this subsection apply to both the crash fault model and the Byzantine fault model.

Predicate $\mathcal{C}$ is a *correctness predicate* if its domain is $(V \cup \{\perp\})^{2n}$. If $E = (F, \langle i_1, \ldots, i_n\rangle, \sigma)$ is any execution then $\text{inp}(E)$ is defined to be $\langle i'_1, \ldots, i'_n\rangle$ where $i'_p = i_p$ if processor p is correct in E and $i_p = \perp$ otherwise. If E is any deciding execution then $\text{ans}(E)$ is defined to be $\langle a_1, \ldots, a_n\rangle$ where a_p is the decision of processor p in execution E if p is correct in E and $a_p = \perp$ otherwise. Protocol $\mathcal{P}$ satisfies correctness predicate $\mathcal{C}$ if for any deciding execution E the value of $\mathcal{C}(\text{inp}(E), \text{ans}(E))$ is true. Correctness predicates furnish a convenient way of formalizing the correctness requirements for a consensus protocol. For example, protocol $\mathcal{P}$ solves the approximate agreement problem (with parameter ϵ) if it terminates and it satisfies correctness predicate $\mathcal{C}$ that is defined below. Let

$$\mathcal{A}(I, A) = \bigwedge_{j,k \in N} ((a_j = \perp) \vee (a_k = \perp) \vee (|a_j - a_k| \leq \epsilon)),$$

and

$$\mathcal{V}(I, A) = \bigwedge_{j \in N} \left((a_j = \perp) \vee \bigvee_{k,l \in N} ((i_k \neq \perp) \wedge (i_l \neq \perp) \wedge (i_k \leq a_j \leq i_l))\right).$$

where $I = \langle i_1, \ldots, i_n\rangle$ and $A = \langle a_1, \ldots, a_n\rangle$. Now let $\mathcal{C}(I, A) = \mathcal{A}(I, A) \wedge \mathcal{V}(I, A)$. The correctness predicate $\mathcal{A}$ formalizes the agreement condition and the correctness predicate $\mathcal{V}$ formalizes the validity condition.

3.6 Time

In this subsection we define a notion of time in asynchronous executions. We will use this notion when we discuss the performance of our compiler. The definitions in this subsection apply to both the crash fault model and the Byzantine fault model.

We define S to be a *timing* if S is an infinite nondecreasing unbounded sequence of real numbers. Let $E = (F, I, \sigma)$ be an execution. If event e is the i-th element of σ, then the *time* at which event e occurs in timing S of E is r where r is the

i-th element of S. Timing S of execution E is 1-*bounded* if (1) each processor that takes a non-error step in E takes its first step at time 0 and (2) any message that is sent at time x is delivered at or before time $x + 1$. (The time at which a particular message is sent is defined to be the time of the event that causes the message to be inserted in a buffer.) A *timed execution* is an execution with a 1-bounded timing.

4. The Compiler

We give two versions of our compiler. One works in any system where the redundancy is at least four. It increases the running time by a factor of two. The other works in any system where the redundancy is at least three. It increases the running time by a factor of three. We prove the correctness of the first version of the compiler. The correctness proof for the second version is similar and is only sketched.

It would be tedious to write protocols directly in terms of our formal model. In the remainder of this chapter we write protocols in a higher-level language. The mapping from protocols written in the higher-level language to protocols written directly in terms of the formal model is a straightforward exercise that we omit. To accommodate our new higher-level language for expressing protocols, we will redefine the terms "transition function" and "decision function" in the body of this section.

4.1 Standard Protocols

Our compiler works only for protocols in standard form. A protocol is in standard form if it corresponds to an instance of Protocol 1 customized by specifying A, V, $\mathcal{S}$, and $\mathcal{D}$. A is the set of possible values of the variable STATE. $V \subset A$ is the set of possible inputs. $\mathcal{S} : N \times \mathcal{I}^+ \times (A \cup \{\perp\})^n \rightarrow A \cup \{\perp\}$ is the *transition function.* The transition function maps a triple consisting of a processor index, a positive integer (representing an "asynchronous round" number), and a vector of messages ($\perp$ represents the absence of a message) into either a processor state (*i.e.*, element of A) or $\perp$. $\mathcal{D} : N \times \mathcal{I}^+ \times A \rightarrow V \cup \{\perp\}$ is the *decision function.* The decision function maps a triple consisting of a processor index, a positive integer, and a processor state into a possible decision. In the range of the decision function, an element of V represents a decision and $\perp$ represents the absence of a decision. Throughout the rest of this chapter, an instance of Protocol 1 customized with A, V, $\mathcal{S}$, and $\mathcal{D}$ is denoted $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$. For the remainder of this section we choose an arbitrary fixed A, V, $\mathcal{S}$, and $\mathcal{D}$.

```
Initialization for processor $p$:
        STATE $\leftarrow$ the initial value of processor $p$
        MSG$_{l,q} \leftarrow \perp$ for all $(l,q) \in \mathcal{I}^+ \times N$
1.  for $r \leftarrow 1$ to $\infty$ do
2.      broadcast $(r, \text{STATE})$
3.      until MSG$_{r,p} \neq \perp$ and $|\{q \in N \mid \text{MSG}_{r,q} \neq \perp\}| \geq n - t$ do
4.          receive any message $(l, m)$ from any processor $q$
5.          MSG$_{l,q} \leftarrow m$
6.      STATE $\leftarrow \mathcal{S}(p, r, \langle \text{MSG}_{r,1}, \ldots, \text{MSG}_{r,n} \rangle)$
7.      DECISION $\leftarrow \mathcal{D}(p, r, \text{STATE})$
8.      if DECISION $\neq \perp$ then decide DECISION
```

Protocol 1: The Standard Protocol (Crash Faults)

We impose the requirement that $\mathcal{S}(r, p, \langle m_1, \ldots, m_n \rangle) \neq \perp$ if and only if there is a set $G \subset N$ such that $p \in G$, $|G| \geq n - t$, and $m_q \neq \perp$ for all $q \in G$. That is, $\mathcal{S}(r, p, \langle m_1, \ldots, m_n \rangle)$ is defined on exactly those message patterns (*i.e.,* patterns of which elements of $\langle m_1, \ldots, m_n \rangle$ are defined) that would cause a correct processor p to exit the inner loop (steps 3 to 5) and proceed to step 6.

A standard-form protocol operates in a series of *asynchronous rounds.* The r-th execution of the body of the main loop is asynchronous round r. At the start of each asynchronous round a correct processor broadcasts a message containing its state. It then waits to receive messages from a sufficiently large group of processors (including itself). It computes its new state by applying its transition function to a triple consisting of its index, the number of the current asynchronous round, and the vector of messages received. Finally, it (possibly) decides on an answer by applying its decision function to its new state. It may seem unusual that a correct processor sends a copy of its state to itself (and waits to receive it) in each asynchronous round. We adopt this convention because it simplifies our compiler.

We say that M is a *message array* if it is a 2-dimensional array of $A \cup \{\perp\}$ indexed by $\mathcal{I}^+$ and N (asynchronous rounds and processor indices). Message array L is an *extension* of message array M if for all r and p either $L_{r,p} = M_{r,p}$ or $M_{r,p} = \perp$. Let $\mathcal{M}$ be the set of all message arrays. In protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ the variable MSG is always an element of $\mathcal{M}$. If the value at processor p of MSG$_{r,q}$ is ever m, then p received the message (r, m) from processor q. The value at processor p of MSG$_{r,q}$ is $\perp$ if p has received no message (r, m) from processor q for any m. At processor p at any time in an execution, the message array MSG contains all of the messages received by processor p up to the present time. Note that messages that arrive "too late" are stored in MSG but do not affect the state or decision of processor p.

4.2 Filtered Message Arrays

In this subsection we define a filter that we will use in both versions of our compiler. This filter operates on message arrays. It obliterates (replaces by $\perp$) messages that seem "implausible" and it passes all other messages unchanged. It is an adaptation of a "validation" scheme due to Bracha [5]. We now define the filter.

For all $G \subset N$ define pick$(G, \langle v_1, \ldots, v_n\rangle)$ to be $\langle v'_1, \ldots, v'_n\rangle$ where $v'_i = v_i$ if $i \in G$ and $v'_i = \perp$ otherwise. The function pick returns a vector in which some of the elements of $\langle v'_1, \ldots, v'_n\rangle$—those with indices in $N - G$—are replaced by $\perp$. The function filter maps from $\mathcal{M}$ to $\mathcal{M}$. Define filter(M) to be M' where

$$M'_{1,p} = \begin{cases} M_{1,p} & \text{if } M_{1,p} \in V; \\ \perp & \text{otherwise,} \end{cases}$$

and

$$M'_{r,p} = \begin{cases} M_{r,p} & \text{if } \exists_{G \subset N}\ M_{r,p} = \mathcal{S}(p, r-1, \text{pick}(G, \langle M'_{r-1,1}, \ldots, M'_{r-1,n}\rangle)); \\ \perp & \text{otherwise,} \end{cases}$$

for all $r \in \{2, 3, \ldots\}$ and for all $p \in N$. For all p, $M'_{1,p}$ is equal to $M_{1,p}$ if and only if $M_{1,p}$ is an element of V, the set of possible inputs to the protocol. For all p and for all $r \geq 2$, $M'_{r,p}$ is equal to $M_{r,p}$ if and only if there is some set $G \subset N$ such that $M_{r,p}$ is the message that would be sent by a correct processor p that received the messages in pick$(G, \langle M'_{r-1,1}, \ldots, M'_{r-1,n}\rangle)$.

In the next two lemmas we prove basic properties of filtered message arrays.

Lemma 1: Suppose that L is a message array and $M =$ filter(L). If $M_{r,p} = \perp$ then $M_{r+1,p} = \perp$ for all $r \in \mathcal{I}^+$ and $p \in N$.

Proof: Immediate from the definition of filter(L). □

Lemma 2: If L is a message array then L is an extension of filter(L).

Proof: Immediate from the definition of filter(L). □

Lemma 3: If message array L is an extension of message array M, then filter(L) is an extension of filter(M).

Proof: Let $L' = \text{filter}(L)$ and let $M' = \text{filter}(M)$. We prove by induction on r that, for all r and for all p, either $L'_{r,p} = M'_{r,p}$ or $M'_{r,p} = \perp$.

Basis: $(r = 1)$ Consider an arbitrary $p \in N$. If $M'_{1,p} = \perp$ then the claim is trivially true, so assume that $M'_{1,p} \neq \perp$. By Lemma 2, $M'_{1,p} = M_{1,p}$. By the definition of filter(M), $M_{1,p} \in V$. Because L is an extension of M, $L_{1,p} = M_{1,p}$. By the definition of filter(L), $L'_{1,p} = L_{1,p}$. Thus $L'_{1,p} = M'_{1,p}$.

Induction: Consider an arbitrary $p \in N$. If $M'_{r,p} = \perp$ then the claim is trivially true, so assume that $M'_{r,p} \neq \perp$. By Lemma 2, $M'_{r,p} = M_{r,p}$. Because L is an extension of M, $L_{r,p} = M_{r,p}$. By the definition of filter(M), there is some set $G \subset N$ such that $M'_{r-1,q} \neq \perp$ for all $q \in G$ and

$$M_{r,p} = \mathcal{S}(p, r-1, \text{pick}(G, \langle M'_{r-1,1}, \ldots, M'_{r-1,n}\rangle)).$$

By the induction hypothesis

$$\text{pick}(G, \langle M'_{r-1,1}, \ldots, M'_{r-1,n}\rangle) = \text{pick}(G, \langle L'_{r-1,1}, \ldots, L'_{r-1,n}\rangle).$$

Thus, by the definition of filter(L), $L'_{r,p} = L_{r,p}$. So we have that $L'_{r,p} = M'_{r,p}$. □

4.3 The Object Protocol

Our compiler operates by translating an instance of Protocol 1 customized by A, V, $\mathcal{S}$, and $\mathcal{D}$ into an instance of Protocol 2 customized by A, V, $\mathcal{S}$, and $\mathcal{D}$. Throughout the rest of this chapter an instance of Protocol 2 customized with A, V, $\mathcal{S}$, and $\mathcal{D}$ is denoted $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$.

The skeleton of Protocol 2 is similar to that of Protocol 1. Three distinguishing features of Protocol 2 should be noted. First, Protocol 2 requires that the redundancy be at least four. Second, there is extra communication in Protocol 2. Each processor uses this extra communication to construct a message array called RAW. Third, in Protocol 2 each processor applies its transition function to the message array MSG, which is obtained by filtering RAW.

As in a standard-form protocol, Protocol 2 operates in a series of *asynchronous rounds*. The r-th execution of the body of the main loop is asynchronous round r. At the start of each asynchronous round a correct processor p broadcasts a message containing the current value of the variable STATE. It then waits until its filtered message array, MSG, contains messages from a sufficiently large group of processors (including itself). It computes its new value for the variable STATE by applying

```
Initialization for processor $p$:

    $\textsc{state} \leftarrow$ the initial value of processor $p$
    $\textsc{vote}_{l,q,i,u} \leftarrow \perp$ for all $(l,q,i,u) \in \mathcal{I}^+ \times N \times \mathcal{N} \times N$
    $\textsc{raw}_{l,q} \leftarrow \perp$ for all $(l,q) \in \mathcal{I}^+ \times N$
    $\textsc{msg}_{l,q} \leftarrow \perp$ for all $(l,q) \in \mathcal{I}^+ \times N$

1.  for $r \leftarrow 1$ to $\infty$ do
2.      broadcast $(r, p, 0, \textsc{state})$

3.      until $\textsc{msg}_{r,p} \neq \perp$ and $|\{q \in N \mid \textsc{msg}_{r,q} \neq \perp\}| \geq n - t$ do
4.          receive any message $(l,q,i,m)$ from any processor $u$
5.          if $\textsc{vote}_{l,q,i,u} = \perp$ then
6.              $\textsc{vote}_{l,q,i,u} \leftarrow m$
7.              $\textsc{num} \leftarrow |\{s \in N \mid \textsc{vote}_{l,q,i,s} = m\}|$
8.              if $i = 0$ and $q = u$ then broadcast $(l,q,1,m)$
9.              if $\textsc{num} = n - 2t$ then broadcast $(l,q,i+1,m)$
10.             if $\textsc{num} = n - t$ then $\textsc{raw}_{l,q} \leftarrow m$
11.             $\textsc{msg} \leftarrow$ filter($\textsc{raw}$)

12.     $\textsc{state} \leftarrow \mathcal{S}(p, r, \langle \textsc{msg}_{r,1}, \ldots, \textsc{msg}_{r,n} \rangle)$
13.     $\textsc{decision} \leftarrow \mathcal{D}(p, r, \textsc{state})$

14.     if $\textsc{decision} \neq \perp$ then decide $\textsc{decision}$
```

Protocol 2: The Object Protocol (Byzantine Faults, $n \geq 4t + 1$)

its transition function to a triple consisting of its index, the number of the current asynchronous round, and the vector of messages in the filtered message array. Finally, it (possibly) decides on an answer by applying its decision function to the new value of STATE.

A processor in Protocol 2 maintains the invariant that MSG = filter(RAW). Messages are placed in the array RAW when they accumulate enough votes. The votes are stored in the array VOTES. If $\textsc{votes}_{l,q,i,u} = m$ then processor u sent a *level* i vote that $\textsc{raw}_{l,q}$ should be m. If a processor receives a level 0 vote from processor q that $\textsc{raw}_{l,q}$ should be m, then it sends a level 1 vote that $\textsc{raw}_{l,q}$ should be m. If a processor receives $n - 2t$ level i votes that $\textsc{raw}_{l,q}$ should be m, then it sends a level $i + 1$ vote that $\textsc{raw}_{l,q}$ should be m. If it receives $n - t$ level i votes that $\textsc{raw}_{l,q}$ should be m, then it sets $\textsc{raw}_{l,q}$ to m.

4.4 Preliminary Lemmas

In this subsection we give five lemmas that will be of use in our main correctness argument in Subsection 4.5. The first four lemmas establish some basic properties of the communication primitives used in Protocol 2 and the last lemma establishes an important liveness property for Protocol 2.

Lemma 4: *If any correct processor ever assigns any value m to its copy of* $\text{RAW}_{r,p}$ *in an execution of protocol* $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$, *then every correct processor eventually assigns the value m to its copy of* $\text{RAW}_{r,p}$ *and no correct processor ever assigns any value $m' \neq m$ to its copy of* $\text{RAW}_{r,p}$.

Proof: Any correct processor that assigns a value to its copy of $\text{RAW}_{r,p}$ does it after receiving an (r, p, i, m') message for some $i \in \mathcal{N}$ and $m' \in A$. We call such a message a *level i* message. Let j be the smallest number such that a level j message causes some correct processor to assign a value to its copy of $\text{RAW}_{r,p}$, let q be such a processor, and let d be the value assigned. Processor q gets $n-t$ messages (r, p, j, d). At least $n - 2t$ are from correct processors. There are at most $2t$ processors that could send an (r, p, j, d') message for any $d' \neq d$. Thus no correct processor assigns any $d' \neq d$ to its copy of $\text{RAW}_{r,p}$ based on a level j message. It is easy to show by induction on j' that for all $j' \geq j + 1$ and for all $d' \neq d$ there are no (r, p, j', d') messages sent by any correct processor. All correct processors eventually receive $n-2t$ messages (r, p, j, d) and broadcast an $(r, p, j+1, d)$ message. Thus each correct processor eventually receives $n - t$ messages $(r, p, j + 1, d)$ and assigns d to its copy of $\text{RAW}_{r,p}$. □

Based on Lemma 4, we define the *eventual value* of $\text{RAW}_{r,p}$ in an execution of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$, denoted $[\text{RAW}_{r,p}]$, to be the common value assigned to $\text{RAW}_{r,p}$ by the correct processors. If the correct processors never assign a value to $\text{RAW}_{r,p}$ then we define $[\text{RAW}_{r,p}]$ to be $\perp$. We define the *eventual value* of RAW, denoted [RAW], in the obvious way. That is, [RAW] is the 2-dimensional array whose elements are the eventual values of the corresponding elements of RAW. Based on Lemma 3, we define the *eventual value* of $\text{MSG}_{r,p}$ and MSG analogously.

Lemma 5: *If a correct processor p ever broadcasts the message $(r, p, 0, m)$ for any r and m, then* $[\text{RAW}_{r,p}] = m$.

Proof: All $n - t$ correct processors eventually receive the message $(r, p, 0, m)$ from processor p. They all broadcast the message $(r, p, 1, m)$. All $n - t$ correct processors eventually receive at least $n-t$ copies of the message $(r, p, 1, m)$ and at most t copies of any message $(r, p, 1, m')$ for any $m' \neq m$. Each correct processor assigns m to its local copy of $\text{RAW}_{r,p}$, and so $[\text{RAW}_{r,p}] = m$. □

Lemma 6: *Let $r \in I^+$. If a correct processor p never broadcasts a message $(r, p, 0, m)$ for any m, then* $[\text{RAW}_{r,p}] = \perp$.

Proof: It is easy to show by induction on i that no correct processor ever sends a message (r, p, i, m) for any m. Therefore no correct processor ever assigns any value to its copy of $\text{RAW}_{r,p}$ and so $[\text{RAW}_{r,p}] = \bot$. □

Lemma 7: *If M is the value of the variable* RAW *at some processor p at some time in an execution and L is the value of the same variable at processor p at some later time in the same execution, then L is an extension of M.*

Proof: Immediate from Lemma 4. □

In the next lemma we prove an important liveness property of Protocol 2.

Lemma 8: $[\text{MSG}_{r,p}] \neq \bot$ *for all* $r \in \mathcal{I}^+$ *and for all* $p \in N - F$.

Proof: The proof is by induction on r.

Basis: $(r = 1)$ Let p be an arbitrary correct processor. Let $v \in V$ be the input to processor p. In its first step processor p sends the message $(1, p, 0, v)$. By Lemma 5, $[\text{RAW}_{1,p}] = v$. By the definition of filter, $[\text{MSG}_{1,p}] = v$.

Induction: Let p be an arbitrary correct processor. By the induction hypothesis $[\text{MSG}_{r-1,p}] \neq \bot$. By Lemma 2, $[\text{RAW}_{r-1,p}] \neq \bot$. By Lemma 6, there is some time when processor p sends the message $(r-1, p, 0, m')$ for some m'. Thus there is some time at which processor p executes the broadcast (step 2) in asynchronous round $r - 1$.

By the induction hypothesis, $[\text{MSG}_{r-1,q}] \neq \bot$ for all $q \in N - F$. By Lemmas 3 and 7, there is some time after which the variable MSG at processor p always satisfies the condition that $\text{MSG}_{r-1,q} \neq \bot$ for all $q \in N - F$. Therefore, processor p eventually sends some message $(r, p, 0, m)$. Let M be the value of the variable MSG at processor p when processor p sends the message $(r, p, 0, m)$. It follows from the code that

$$m = \mathcal{S}(p, r-1, \langle M_{r-1,1}, \ldots, M_{r-1,n} \rangle).$$

By Lemmas 3 and 4, [MSG] is an extension of M. Therefore there is a $G \subset N$ such that

$$m = \mathcal{S}(p, r-1, \text{pick}(G, \langle [\text{MSG}_{r-1,1}], \ldots, [\text{MSG}_{r-1,n}] \rangle)).$$

By Lemma 5, $[\text{RAW}_{r,p}] = m$. By the definition of filter, $[\text{MSG}_{r,p}] = [\text{RAW}_{r,p}]$. Thus $[\text{MSG}_{r,p}] \neq \bot$. □

4.5 Proof of Correctness

In this subsection we show for any A, V, $\mathcal{S}$, and $\mathcal{D}$ that if protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ solves some problem (formalized by a correctness predicate and a termination requirement) then protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ solves the same problem. Our approach is to exhibit for any execution E of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ an execution of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ in which the correct processors "do the same thing" that they did in execution E.

Let v_0 be an arbitrary fixed element of V. In this subsection we will find it convenient to use v_0 as a "default input."

We now construct for any execution E of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ a synchronous execution of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$, denoted crash($E$), in which the correct processors "do the same thing" that they did in E. Because crash(E) is a synchronous execution in the crash fault model, it is completely determined by the following three items: (1) the inputs to the processors; (2) the number, if any, of the last phase in which each processor takes steps, and (3) the order in which each operating processor receives its phase r messages for each r. We will construct crash(E) so that the following three properties hold: (1) the input to processor p is $[\textsc{msg}_{1,p}]$ if $[\textsc{msg}_{1,p}] \neq \perp$ and v_0 otherwise; (2) a processor p sends a phase r message if and only if $[\textsc{msg}_{r,p}] \neq \perp$; and (3) messages are delivered to a processor p in phase r in precisely the order that causes processor p to send the message $(r+1, [\textsc{msg}_{r+1,p}])$.

Define support(p, r, M) to be the lexicographically least set $G \subset N$ such that $M_{r-1,q} \neq \perp$ for all $q \in G$ and

$$M_{r,p} = \mathcal{S}(p, r-1, \text{pick}(G, \langle M_{r-1,1}, \ldots, M_{r-1,n} \rangle)).$$

If there is no such set G, then define support(p, r, M) to be $\emptyset$.

If $M_{1,p} = \perp$ then define $\beta(0, p, M)$ to be the empty sequence of events; otherwise, define $\beta(0, p, M)$ to be the event (*step:* p). For all $r \geq 1$, if $M_{r+1,p} = \perp$ then define $\beta(r, p, M)$ to be the empty sequence of events; otherwise, define $\beta(r, p, M)$ to be the sequence of events that consists of (1) the receipt by processor p of all of the (non-$\perp$) messages in $\langle M_{r,1}, \ldots, M_{r,n} \rangle$ that are from processors in support$(p, r+1, M) - \{p\}$ followed by (2) the receipt by processor p of $M_{r,p}$ followed by (3) the receipt by processor p of all of the remaining (non-$\perp$) messages in $\langle M_{r,1}, \ldots, M_{r,n} \rangle$.

Suppose $E = (F, I, \sigma)$ is an arbitrary execution of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$. We define crash($E$) to be $(F, \langle i'_1, \ldots, i'_n \rangle, \sigma'_0 \sigma'_1 \ldots)$ where

$$i'_p = \begin{cases} [\text{MSG}_{1,p}] & \text{if } [\text{MSG}_{1,p}] \neq \perp; \\ v_0 & \text{otherwise,} \end{cases}$$

and for all r

$$\sigma'_r = \beta(r, 1, [\text{MSG}])\beta(r, 2, [\text{MSG}]) \ldots \beta(r, n, [\text{MSG}]).$$

Lemma 9: *Let $p \in N$, $q \in N$, $r \in \mathcal{N}$, and $m \in A$. Let L be a message array. Let $M =$ filter(L). Let C be an initial configuration for protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$. Let $\sigma' = \sigma_0 \sigma_1 \ldots \sigma_{r-1}$ where (for $0 \leq r' \leq r - 1$) $\sigma_{r'} = \beta(r', 1, M) \ldots \beta(r', n, M)$. Suppose that σ' is a schedule that is applicable to configuration C. Suppose that processor p sends an (r, m) message to processor q in $\sigma'(C)$. Then $m = M_{r,p}$.*

Proof: There are two cases.

Case 1: ($r = 1$) Let v be the input to processor p in configuration C. Clearly, $m = v$. To send the message $(1, m)$, processor p must take at least one step in schedule σ'. It follows from Lemma 1 and the definition of σ' that $M_{1,p} \neq \perp$. By the construction of σ', $v = M_{1,p}$. Thus $m = M_{1,p}$.

Case 2: ($r \geq 2$) Let M' be the value of the variable MSG in protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ after the application of the event that causes the sending of the message (r, m) from processor p to processor q. By Lemmas 3 and 7, message array M is an extension of message array M'. Because the message (r, m) is sent in $\sigma'(C)$, we have that support(p, r, M) $\neq \perp$. We claim that

$$\langle M'_{r-1,1}, \ldots, M'_{r-1,n} \rangle = \text{pick}(\text{support}(p, r, M), \langle M_{r-1,1}, \ldots, M_{r-1,n} \rangle).$$

The claim follows by the definition of $\beta(r - 1, p, M)$. Using the claim, we calculate that

$$\begin{aligned} m &= \mathcal{S}(p, r-1, \langle M'_{r-1,1}, \ldots, M'_{r-1,n} \rangle) && \text{From the code.} \\ &= \mathcal{S}(p, r-1, \text{pick}(\text{support}(p, r, M), \langle M_{r-1,1}, \ldots, M_{r-1,n} \rangle)) && \text{By the claim.} \\ &= M_{r,p}. && \text{By the definition of support.} \end{aligned}$$ □

Lemma 10: *If $E = (F, \langle i_1, \ldots, i_n \rangle, \sigma)$ is an execution of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$, then $E' =$ crash(E) is an execution of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$.*

Proof: Suppose that $E' = (F, I', \sigma')$ where $I' = \langle i'_1, \ldots, i'_n \rangle$. Partition the schedule σ' into subschedules σ'_0, σ'_1, *etc.* with σ'_i defined as it is in the definition of crash.

To show that E' is an execution of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$, we verify the following three properties. (1) The schedule σ' is applicable to the initial configuration C of $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ in which F is the set of faulty processors and in which processor p begins in the initial state that corresponds to input i'_p for all $p \in N$. (2) All processors in $N - F$ take an infinite number of steps. (3) If processor p takes an infinite number of steps, then every message that is sent to processor p is eventually delivered.

Property 1: We prove for all $r \in \mathcal{N}$ that all events in σ'_r are applicable. The proof is by induction on r.

Basis: $(r = 0)$ All events in σ'_0 are of the form (*step:* p) for some $p \in N$. It is immediate that σ'_0 is applicable to configuration C.

Induction: Pick an arbitrary event $e =$ (*receive:* $q, p, (r, [\textsc{msg}_{r,p}])$) from the schedule σ'_r. By the induction hypothesis, the schedule $\sigma'_0\sigma'_1 \ldots \sigma'_{r-1}$ is applicable to configuration C. The event e is in the schedule $\beta(r, q, [\textsc{msg}])$. By the definition of $\beta(r, q, [\textsc{msg}])$, $[\textsc{msg}_{r,p}] \neq \perp$. By the definition of $\beta(r-1, p, [\textsc{msg}])$, processor p sends some message (r, m) to processor q in $\sigma'_0\sigma'_1 \ldots \sigma'_{r-1}(C)$. By Lemma 9, $m = [\textsc{msg}_{r,p}]$. Thus the message $(r, [\textsc{msg}_{r,p}])$ is placed in the buffer of processor q. It is easy to see that the event e is unique in the schedule σ'. Therefore, the message $(r, [\textsc{msg}_{r,p}])$ is in the buffer of processor q in the configuration just before the event e is applied. It follows that the schedule $\sigma'_0\sigma'_1 \ldots \sigma'_r$ is applicable to configuration C.

Property 2: Let p be an arbitrary element of $N - F$. By Lemma 8, $[\textsc{msg}_{r,p}] \neq \perp$ for all $r \in \mathcal{I}^+$. By the construction of σ', each (non-$\perp$) message in $[\textsc{msg}]$ is delivered to processor p in the execution (F, I', σ'). There are an infinite number of such messages. To receive all of these messages, processor p must take an infinite number of steps.

Property 3: Let q be any processor that takes an infinite number of steps in execution E'. Consider an arbitrary message (r, m) sent from a processor p to the processor q in E'. By property 1 and Lemma 9, $m = [\textsc{msg}_{r,p}]$.

We now show that the message $(r, [\textsc{msg}_{r,p}])$ is delivered to processor q in execution E'. Processor q takes an infinite number of steps in execution E'. By the definition of E', $[\textsc{msg}_{r',q}] \neq \perp$ for infinitely many r'. By Lemma 1, $[\textsc{msg}_{r',q}] \neq \perp$ for all r'. So $[\textsc{msg}_{r+1,q}] \neq \perp$. By construction, the schedule $\beta(r, q, [\textsc{msg}])$ includes the event $(q, p, (r, [\textsc{msg}_{r,p}]))$. Thus the message (r, m) is delivered in the execution E'. □

Lemma 11: *If E is any execution of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$, then the executions E and $E' = \text{crash}(E)$ have the same set of faulty processors and the same inputs to the correct processors.*

Proof: It is immediate that executions E and E' have the same faulty processors. We now show that the correct processors have the same inputs in executions E and E'. Let p be an arbitrary correct processor. Suppose that in execution E processor p has input $v \in V$. Processor p broadcasts the message $(1, p, 0, v)$ in its first step in execution E. By Lemma 5, $[\text{RAW}_{1,p}] = v$. By the definition of filter, $[\text{MSG}_{1,p}] = v$. Thus, the input to processor p in execution E' is v. □

If E is an execution of either protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ or protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$, processor p is correct in E, and $r \in \mathcal{N}$, then we define $\text{state}(p, r, E)$ to be the $(r+1)$-st value that processor p assigns to the variable STATE in the execution E. For example, $\text{state}(p, 0, E)$ is the input to processor p.

Lemma 12: *If E is an execution of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ and $E' = \text{crash}(E)$, then $\text{state}(p, r, E) = \text{state}(p, r, E')$ for all $(p, r) \in (N - F) \times \mathcal{I}^+$.*

Proof: If $r = 0$ then the claim follows from Lemma 11. Suppose instead that $r \geq 1$. By the construction of the execution E',

$$\text{state}(p, r, E') = \mathcal{S}(p, r+1, \text{pick}(\text{support}(p, r, [\text{MSG}]), \langle [\text{MSG}_{r,1}], \ldots, [\text{MSG}_{r,n}] \rangle)).$$

By the definition of support

$$\mathcal{S}(p, r+1, \text{pick}(\text{support}(p, r, [\text{MSG}]), \langle [\text{MSG}_{r,1}], \ldots, [\text{MSG}_{r,n}] \rangle)) = [\text{MSG}_{r+1,p}].$$

By the definition of filter, $[\text{MSG}_{r+1,p}] = [\text{RAW}_{r+1,p}]$. By Lemma 5, $\text{state}(p, r, E) = [\text{RAW}_{r+1,p}]$. Thus $\text{state}(p, r, E') = \text{state}(p, r, E)$. □

Theorem 13: *If $n \geq 4t + 1$, then the following two conditions hold.*

- *Correctness condition: If protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ satisfies some correctness predicate $\mathcal{C}$ then so does protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$.*
- *Termination condition: If protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ terminates then so does protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$.*

Proof: We verify that the two conditions are satisfied.

Correctness condition: Suppose protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ satisfies correctness predicate $\mathcal{C}$. Let $E = (F, I, \sigma)$ be an arbitrary deciding execution of $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$.

Let $E' = \text{crash}(E)$. By Lemma 10, E' is an execution of $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$. Suppose $E' = (F', I', \sigma')$. By Lemma 11, $F = F'$ and $\text{inp}(E) = \text{inp}(E')$. By Lemma 12, E' is a deciding execution and $\text{ans}(E) = \text{ans}(E')$. Therefore, $\mathcal{C}(\text{inp}(E), \text{ans}(E)) = \mathcal{C}(\text{inp}(E'), \text{ans}(E'))$ and $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ also satisfies correctness predicate $\mathcal{C}$.

Termination condition: We prove the contrapositive of the claim. Suppose that protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ does not terminate. By the definition of termination, there is some non-deciding execution E of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$. Let $E' = \text{crash}(E)$. By Lemma 10, E' is an execution of $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$. By Lemma 12, execution E' is a non-deciding execution of $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$. Thus, protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ does not terminate. □

Theorem 14: *For all $I \in V^n$, suppose that all timed executions of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ with input I terminate by time r and $n \geq 4t + 1$. Then all timed executions of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ with input I terminate by time $2 \cdot r$.*

Proof: We claim that all correct processors decide by the end of asynchronous round r in all timed executions of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ with input I. Suppose not. Then, there is some timed execution E of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ with input I in which a correct processor decides in asynchronous round r' for some $r' > r$. We can construct a synchronous execution E' of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ in which the sequence of values assigned to the variable STATE at each correct processor is the same as it is in execution E. It should be clear that each correct processor decides in the same asynchronous round in executions E and E'. For the synchronous execution E', consider the obvious 1-bounded timing where receiving an asynchronous round r'' message takes place at time r''. In this timed execution there is a correct processor which decides at time r'. This contradicts the assumption that there is no such execution.

Let E be an arbitrary timed execution of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ with input I. It follows from Lemma 12 that if there is a correct processor in the execution $\text{crash}(E)$ of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ that decides in asynchronous round r' for some r', then the same correct processor decides in asynchronous round r' in execution E of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$. It follows from the claim that all correct processors in execution E decide by asynchronous round r.

We can show by induction on r' that in the timed execution E of protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ all processors end asynchronous round r' at or before time $2 \cdot r'$. Thus all correct processors in the timed execution E decide by time $2 \cdot r$. □

4.6 An Alternative Object Protocol

The alternative version of our compiler operates by translating an instance of Protocol 1 customized by A, V, $\mathcal{S}$, and $\mathcal{D}$ into an instance of Protocol 3 customized by A, V, $\mathcal{S}$, and $\mathcal{D}$. Throughout the rest of this chapter an instance of Protocol 3 customized with A, V, $\mathcal{S}$, and $\mathcal{D}$ is denoted $\mathcal{T}(A, V, \mathcal{S}, \mathcal{D})$. Protocol 3 requires that the redundancy be at least three. It uses the function filter defined in Subsection 4.2.

The only difference between Protocol 3 and Protocol 2 is that in Protocol 3 we use a different set of communication primitives to install elements in the message array RAW. They are the communication primitives developed by Bracha [5].

```
    Initialization for processor $p$:
        STATE ← the initial value of processor $p$
        $\text{VOTE}_{l,q,i,u} \leftarrow \perp$ for all $(l,q,i,u) \in \mathcal{I}^+ \times N \times \mathcal{N} \times N$
        $\text{RAW}_{l,q} \leftarrow \perp$ for all $(l,q) \in \mathcal{I}^+ \times N$
        $\text{MSG}_{l,q} \leftarrow \perp$ for all $(l,q) \in \mathcal{I}^+ \times N$

1.  for $r \leftarrow 1$ to $\infty$ do
2.      broadcast $(r, p, 0, \text{STATE})$
3.      until $\text{MSG}_{r,p} \neq \perp$ and $|\{q \in N \mid \text{MSG}_{r,q} \neq \perp\}| \geq n - t$ do
4.          receive any message $(l, q, i, m)$ from any processor $u$
5.          if $\text{VOTE}_{l,q,i,u} = \perp$ then
6.              $\text{VOTE}_{l,q,i,u} \leftarrow m$
7.              $\text{NUM} \leftarrow |\{s \in N \mid \text{VOTE}_{l,q,i,s} = m\}|$
8.              if $i = 0$ and $q = u$ then broadcast $(l, q, 1, m)$
9.              if $i = 1$ and $\text{NUM} = n - t$ then broadcast $(l, q, 2, m)$
10.             if $i = 2$ and $\text{NUM} = n - 2t$ then broadcast $(l, q, 2, m)$
11.             if $i = 2$ and $\text{NUM} = n - t$ then $\text{RAW}_{l,q} \leftarrow m$
12.             $\text{MSG} \leftarrow$ filter(RAW)
13.     $\text{STATE} \leftarrow \mathcal{S}(p, r, \langle \text{MSG}_{r,1}, \ldots, \text{MSG}_{r,n} \rangle)$
14.     $\text{DECISION} \leftarrow \mathcal{D}(p, r, \text{STATE})$
15.     if $\text{DECISION} \neq \perp$ then decide DECISION
```

Protocol 3: The Object Protocol (Byzantine Faults, $n \geq 3t + 1$)

Theorem 15: *If $n \geq 3t + 1$, then the following two conditions hold.*

- *Correctness condition: If protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ satisfies some correctness predicate $\mathcal{C}$ then so does protocol $\mathcal{T}(A, V, \mathcal{S}, \mathcal{D})$.*

- *Termination condition: If protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ terminates then so does protocol $\mathcal{T}(A, V, \mathcal{S}, \mathcal{D})$.*

Proof sketch: We begin by proving the analogues of Lemmas 5, 6, 7 and 8. The remainder of the proof is identical to the proof of Theorem 13. □

Theorem 16: *For all $I \in V^n$, suppose that all timed executions of protocol $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$ with input I terminate by time r and $n \geq 3t + 1$. Then all timed executions of protocol $\mathcal{T}(A, V, \mathcal{S}, \mathcal{D})$ with input I terminate by time $3 \cdot r$.*

Proof: Similar to the proof of Theorem 14. □

5. Approximate Agreement Protocols

We use our compiler to simplify the design of a new approximate agreement protocol that operates in the Byzantine fault model. In Subsection 5.1 we review some definitions and basic results regarding multisets. In Subsection 5.2 we give an approximate agreement protocol that operates in the crash fault model. In Subsection 5.3 we prove our approximate agreement protocol correct in the crash fault model. In Subsection 5.4 we apply the two versions of our compiler to the approximate agreement protocol given in Subsection 5.2. In the Byzantine fault model, we compare the performance of our compiled protocols with the performance of the protocol of Dolev *et al.*

5.1 Preliminary Definitions

We give some definitions and prove some basic facts about multisets. The presentation in this subsection borrows heavily from Dolev *et al.* [19]. Lemma 19 of this subsection is very similar to Lemma 5 of Dolev *et al.*

We view a finite multiset U of reals as a function $U : \mathcal{R} \rightarrow \mathcal{N}$ that is nonzero on at most finitely many $r \in \mathcal{R}$. Intuitively, the function U assigns a multiplicity to each real number. In the remainder of this section the term *multiset* always refers to finite multisets of reals as described above.

The *cardinality* of a multiset U is given by $\sum_{r \in \mathcal{R}} U(r)$ and is denoted by $|U|$. A multiset is *empty* if its cardinality is 0; otherwise it is *nonempty.* Multiset U is a *subset* of multiset V, written $U \subset V$, if $U(r) \leq V(r)$ for all $r \in \mathcal{R}$. The *minimum* $\min(U)$ of a nonempty multiset U is given by $\min(U) = \min\{r \in \mathcal{R} | U(r) \neq 0\}$. The *maximum* $\max(U)$ is defined analogously. If multiset U is nonempty, let $\rho(U)$ (the *range* of U) be the closed interval $[\min(U), \max(U)]$, and let $\delta(U)$ (the *diameter* of U) be $\max(U) - \min(U)$.

For the remainder of this chapter, let $c = \lfloor (n-1)/t \rfloor$. The constant c, which is the floor of the redundancy, plays a role in the definition of our averaging functions and, as we will see in Theorem 25, is the convergence rate of our approximate agreement protocol. Suppose that U is a multiset with $|U| = n - t$. Let $u_0 \leq u_1 \leq \ldots \leq u_{n-t-1}$ be the elements of U in nondecreasing order. Define select(U) to be the multiset consisting of the elements $u_0, u_t, \ldots, u_{(c-1)\cdot t}$. Thus select$(U)$ chooses the smallest element of U and every t-th element thereafter. The *median* of multiset U, written median(U), is defined to be u_m where $m = \lfloor |U|/2 \rfloor$. The *mean* of multiset U, written mean(U), is defined to be

$$\text{mean}(U) = \sum_{r \in \mathcal{R}} \frac{r \cdot U(r)}{|U|}.$$

In our approximate agreement protocol we will use the two averaging functions median(U) and mean(select(U)). The next three lemmas characterize the convergence properties of these functions.

Lemma 17: *If U and V are multisets such that $V \subset U$, then* $\text{median}(V) \in \rho(U)$ *and* $\text{mean}(\text{select}(V)) \in \rho(U)$.

Proof: This is immediate from the definition of the averaging functions. □

Lemma 18: *If U, V and W are multisets such that $|V| = |W| = n - t$, $V \subset U$, $W \subset U$, and $|U| \leq n$, then* $\text{median}(V) \in \rho(W)$.

Proof: Let $v_0 \leq v_1 \leq \ldots \leq v_{n-t-1}$ be the elements of V, let $w_0 \leq w_1 \leq \ldots \leq w_{n-t-1}$ be the elements of W, and let $u_0 \leq u_1 \leq \ldots \leq u_{|U|-1}$ be the elements of U. We calculate that

$$\begin{aligned}
\text{median}(V) &\geq v_t && \text{Because } |V| \geq 2t+1.\\
&\geq u_t && \text{Because } V \subset U.\\
&\geq w_0 && \text{Because } W \subset U \text{ and } |U| - |W| \leq t.\\
&= \min(W).
\end{aligned}$$

Thus $\text{median}(V) \geq \min(W)$. By a similar argument $\text{median}(V) \leq \max(W)$. It follows that $\text{median}(V) \in \rho(W)$. □

Lemma 19: *If U, V and W are multisets such that $|V| = |W| = n - t$, $V \subset U$, $W \subset U$, and $|U| \leq n$, then* $|\text{mean}(\text{select}(V)) - \text{mean}(\text{select}(W))| \leq \delta(U)/c$.

Proof: Let $v_0 \leq v_1 \leq \ldots \leq v_{c-1}$ be the elements of select(V) and let $w_0 \leq w_1 \leq \ldots \leq w_{c-1}$ be the elements of select(W).

We claim that $\max(v_i, w_i) \leq \min(v_{i+1}, w_{i+1})$ for $0 \leq i \leq c-2$. Let $u_0 \leq u_1 \leq \ldots \leq u_{|U|-1}$ be the elements of U. Observe that $v_i \leq u_{(i+1)\cdot t} \leq v_{i+1}$ because $V \subset U$ and because there are at most t elements of U that are not in V. Similarly, $w_i \leq u_{(i+1)\cdot t} \leq w_{i+1}$. Thus $\max(v_i, w_i) \leq \min(v_{i+1}, w_{i+1})$ for $0 \leq i \leq c-2$. This concludes the proof of the claim.

Let $x = |\,\mathrm{mean}(\mathrm{select}(V)) - \mathrm{mean}(\mathrm{select}(W))|$. We use the claim in the calculation that follows.

$$\begin{aligned}
x &= \left|\left(\frac{1}{c}\cdot\sum_{i=0}^{c-1} v_i\right) - \left(\frac{1}{c}\cdot\sum_{i=0}^{c-1} w_i\right)\right| \\
&= \frac{1}{c}\cdot\left|\sum_{i=0}^{c-1}(v_i - w_i)\right| \\
&\leq \frac{1}{c}\cdot\sum_{i=0}^{c-1}|v_i - w_i| \qquad \text{By the triangle inequality.} \\
&= \frac{1}{c}\cdot\sum_{i=0}^{c-1}(\max(v_i, w_i) - \min(v_i, w_i)) \\
&= \frac{1}{c}\cdot\left(\max(v_{c-1}, w_{c-1}) - \min(v_0, w_0) + \sum_{i=0}^{c-2}(\max(v_i, w_i) - \min(v_{i+1}, w_{i+1}))\right) \\
&\leq (\max(v_{c-1}, w_{c-1}) - \min(v_0, w_0))/c \qquad \text{By the claim.} \\
&\leq (\max(U) - \min(U))/c \qquad \text{Because } V \subset U \text{ and } W \subset U. \\
&= \delta(U)/c.
\end{aligned}$$

□

5.2 The Protocol

Our approximate agreement protocol is given as Protocol 4. A processor begins the protocol by assigning its input value to the variable VAL. The protocol is organized into a series of asynchronous rounds. In each asynchronous round each processor that is still operating broadcasts the value of VAL, waits to receive at least $n-t$ values broadcast in the current asynchronous round, places the multiset of these $n-t$ values in the variable SAMPLE, and applies an averaging function to SAMPLE to get a new value for VAL. In the first two asynchronous rounds the averaging function used is median. In subsequent asynchronous rounds it is mean ∘ select. In asynchronous round 2 each processor that is operating calculates an upper bound on the number of asynchronous rounds required and stores the bound in the variable ROUNDS. When sufficient asynchronous rounds have elapsed, a processor decides on the current value of VAL as its answer.

```
    Initialization for processor $p$:
        VAL $\leftarrow$ the initial value of processor $p$
 1. for $r \leftarrow 1$ to $\infty$ do
 2.     broadcast $(r, \text{VAL})$
 3.     wait to receive $(r, *)$ from $n - t$ processors
 4.     let SAMPLE be the multiset of values received in the previous step
 5.     if $r = 1$ then
 6.         VAL $\leftarrow$ median(SAMPLE)
 7.     if $r = 2$ then
 8.         VAL $\leftarrow$ median(SAMPLE)
 9.         ROUNDS $\leftarrow 2 + \lceil \log_c(\max(1, \delta(\text{SAMPLE})/\epsilon)) \rceil$
10.         if $r =$ ROUNDS then decide VAL
11.     if $r \geq 3$ then
12.         VAL $\leftarrow$ mean(select(SAMPLE))
13.         if $r =$ ROUNDS then decide VAL
```

Protocol 4: An Approximate Agreement Protocol (Crash Faults, $n \geq 3t + 1$)

Some straightforward translation is necessary to put Protocol 4 in standard form. We omit the details.

5.3 Proof of Correctness

For all $r \geq 1$, we say that a processor p *finishes* asynchronous round r if it completes the last instruction in the code for asynchronous round r.

Lemma 20: *In every execution of Protocol 4, for all r, all correct processors eventually finish asynchronous round r.*

Proof: An easy induction on r. □

In an execution of Protocol 4 we let X_0 denote the multiset containing the inputs to all of the correct processors and we let X_r denote the multiset containing the value of the variable VAL at the end of asynchronous round r for all processors that finish asynchronous round r. It follows from Lemma 20 that $|X_r| \geq n - t$ for all r.

The next three lemmas help establish the convergence of our approximate agreement protocol. In proving these lemmas, we use the properties of our two averaging functions that we proved in Subsection 5.1.

Lemma 21: *If $r \geq 1$, then $\rho(X_r) \subset \rho(X_{r-1})$.*

Proof: There are two cases. Either $r = 1$ or $r \geq 2$.

Case 1: ($r = 1$) Let U be the multiset of inputs to all processors. Because there are n processors, $|U| = n$. Let W be the multiset of inputs to an arbitrarily chosen set of $n - t$ correct processors. Clearly, $W \subset X_0 \subset U$ and $|W| = n - t$. Let p be an arbitrary processor that finishes asynchronous round 1. Let V be the multiset of values received by processor p in asynchronous round 1. Because there are only crash faults, $V \subset U$. From step 2 of the code, $|V| = n - t$. We have established that the multisets U, V, and W satisfy the preconditions of Lemma 18; therefore, median(V) $\in \rho(W) \subset \rho(X_0)$. Because median($V$) is an arbitrarily chosen element of X_1, we have that $\rho(X_1) \subset \rho(X_0)$.

Case 2: ($r \geq 2$) Let $U = X_{r-1}$. Let p be an arbitrary processor that finishes asynchronous round r. Let V be the multiset of values received by processor p in asynchronous round r. Because there are only crash faults, $V \subset U$. If $r = 2$ then let a = median(V); otherwise, let a = mean(select(V)). We have established that the multisets U and V satisfy the preconditions of Lemma 17; therefore, $a \in \rho(X_{r-1})$. Because a is an arbitrarily chosen element of X_r, we have that $\rho(X_r) \subset \rho(X_{r-1})$. □

Lemma 22: *If Y is the multiset of values received by an arbitrary correct processor in asynchronous round 2, then $\delta(X_2) \leq \delta(Y)$.*

Proof: Let $U = X_1$. Because there are n processors, $|U| \leq n$. Let p be an arbitrary processor that finishes asynchronous round 2. Let V be the multiset of values received by processor p in asynchronous round 2. Let $W = Y$. Because there are only crash faults, $V \subset U$ and $W \subset U$. From step 2 of the code, $|V| = |W| = n - t$. We have established that the multisets U, V, and W satisfy the preconditions of Lemma 18; therefore, median(V) $\in \rho(W) = \rho(Y)$. Because median(V) is an arbitrarily chosen element of X_2, we have that $\rho(X_2) \subset \rho(Y)$. It is immediate that $\delta(X_2) \leq \delta(Y)$. □

Lemma 23: *If $r \geq 3$, then $\delta(X_r) \leq \delta(X_{r-1})/c$. (Recall that c is the floor of the redundancy.)*

Proof: Let $U = X_{r-1}$. Because there are n processors, $|U| \leq n$. Let p and q be two arbitrary processors that finish asynchronous round r. Let V be the multiset of values received by processor p in asynchronous round r and let W be the multiset of values received by processor q in asynchronous round r. Because there are only crash faults, $V \subset U$ and $W \subset U$. From step 2 of the code, $|V| = |W| = n - t$. We have established that the multisets U, V, and W satisfy the preconditions of Lemma 19; therefore, $|\text{mean}(\text{select}(V)) - \text{mean}(\text{select}(W))| \leq \delta(X_{r-1})/c$. Because mean(select($V$)) and mean(select($W$)) are arbitrarily chosen elements of X_r, we have that $\delta(X_r) \leq \delta(X_{r-1})/c$. □

Theorem 24: *In the crash fault model, Protocol 4 solves the approximate agreement problem.*

Proof: We show that the agreement, validity, and termination conditions are satisfied.

Agreement condition: Let p and p' be arbitrary correct processors. Suppose that processor p decides v in asynchronous round r and processor p' decides v' in asynchronous round r'. Without loss of generality assume that $r \leq r'$. Let Y be the multiset of values received by processor p in asynchronous round 2.

We claim that $\delta(Y)/c^{i-2} \geq \delta(X_i)$ for all $i \geq 2$. The proof of the claim is by induction on i. The basis ($i = 2$) is immediate from Lemma 22. The inductive step is immediate from Lemma 23.

From steps 9, 10 and 13 of the code, $r = 2 + \lceil \log_c(\max(1, \delta(Y)/\epsilon)) \rceil$. It follows that $\epsilon \geq \delta(Y)/c^{r-2}$. By the claim, $\epsilon \geq \delta(X_r)$. Clearly, $v \in X_r$ and $v' \in X_{r'}$. By repeated application of Lemma 21, $v \in \rho(X_r)$. Thus $|v - v'| \leq \delta(X_r) \leq \epsilon$.

Validity condition: If v is the decision of some correct processor, then there is some r such that $v \in X_r$. By repeated application of Lemma 21, $\rho(X_r) \subset \rho(X_0)$. Thus $v \in \rho(X_0)$ and there are correct processors with inputs $\min(X_0)$ and $\max(X_0)$ such that such that $\min(X_0) \leq v \leq \max(X_0)$.

Termination condition: Let p be an arbitrary correct processor. Processor p assigns some value—an integer greater than or equal to 2—to the variable ROUNDS in asynchronous round 2; it never changes the variable ROUNDS after asynchronous round 2. Eventually, processor p calculates that r = ROUNDS and it decides on some answer in either step 10 or step 13. Thus each correct processor eventually decides. Because there are a finite number of correct processors, all correct processors eventually decide. □

We say that an approximate agreement protocol has *convergence rate* l if there is some constant k such that in every timed execution where the multiset of inputs to the correct processors is X, all correct processors decide by time $k + \lceil \log_l(\max(1, \delta(X)/\epsilon)) \rceil$.

Theorem 25: *The convergence rate of Protocol 4 is c.*

Proof: For all $r \geq 1$, it is easy to see that in any timed execution of Protocol 4, asynchronous round r ends by time r. Thus it is sufficient to show that every correct processor decides by asynchronous round $2 + \lceil \log_c(\max(1, \delta(X_0)/\epsilon)) \rceil$.

Let p be an arbitrary correct processor. Let Y be the multiset of values received by processor p in asynchronous round 2. It is clear that $Y \subset X_0$. So, $\delta(Y) \leq \delta(X_0)$ and processor p assigns the value $2 + \lceil \log_c(\max(1, \delta(Y)/\epsilon)) \rceil$ to the variable ROUND. Thus processor p decides by asynchronous round $2 + \lceil \log_c(\max(1, \delta(X_0)/\epsilon)) \rceil$. □

5.4 Approximate Agreement with Byzantine Faults

So far in this section, we have developed an approximate agreement protocol that tolerates crash faults. We now apply the two versions of the compiler developed in Section 4 to produce approximate agreement protocols that tolerate Byzantine faults.

It is possible to express Protocol 4 in the standard form defined in Subsection 4.1. That is, there are A, V, $\mathcal{S}$, and $\mathcal{D}$ such that Protocol 4 can be expressed as $\mathcal{P}(A, V, \mathcal{S}, \mathcal{D})$. For the remainder of this chapter we let A, V, $\mathcal{S}$, and $\mathcal{D}$ be chosen in that way.

Theorem 26: *In the Byzantine fault model, for $c \geq 4$, protocol $\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$ solves the approximate agreement problem with a convergence rate of $\sqrt{c}$.*

Proof: Correctness follows from Theorems 13 and 24. It follows from Theorems 14 and 25 that the convergence rate is $\sqrt{c}$. □

Theorem 27: *In the Byzantine fault model, for $c \geq 3$, protocol $\mathcal{T}(A, V, \mathcal{S}, \mathcal{D})$ solves the approximate agreement problem with a convergence rate of $\sqrt[3]{c}$.*

Proof: Correctness follows from Theorems 15 and 24. It follows from Theorems 16 and 25 that the convergence rate is $\sqrt[3]{c}$. □

In Tables 1 and 2, which follow, we compare the Dolev *et al.* protocol with the two compiled versions of our approximate agreement protocol. To compare the convergence rates, we need to overcome one obstacle. For our definition of convergence rate, the Dolev *et al.* protocol, as published, has no bounded convergence rate. The difficulty lies with the method that correct processors use to estimate the number of asynchronous rounds required until termination. Faulty processors can cause this estimate to be unboundedly pessimistic. This difficulty could easily be overcome if the Dolev *et al.* protocol were modified to use an estimation method similar to the one used in our Protocol 4. To allow for a fair comparison of convergence rates we assume that the Dolev *et al.* protocol has been modified in this way.

In Table 1 we compare the convergence rates and the minimum required redundancy. We can see that the asymptotic convergence rate of the Dolev *et al.* protocol is better than the asymptotic convergence rate of either compiled version of our protocol. Our protocols, however, operate with a smaller amount of redundancy.

Protocol	Convergence Rate	Minimum Redundancy
$\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$	$\sqrt{c}$	4
$\mathcal{T}(A, V, \mathcal{S}, \mathcal{D})$	$\sqrt[3]{c}$	3
Dolev et al.	$\lfloor (c-1)/2 \rfloor$	5

Table 1: Comparison of Performance

In Table 2 we give numerical values of the convergence rate for specific small values of c. The data in Table 2 shows that, for any system with $n \leq 7t$, there is a compiled version of our approximate agreement protocol that has a better convergence rate than the Dolev *et al.* protocol.

	$c = 3$	$c = 4$	$c = 5$	$c = 6$	$c = 7$	$c = 8$
$\mathcal{Q}(A, V, \mathcal{S}, \mathcal{D})$	—	2	2.24	2.45	2.65	2.83
$\mathcal{T}(A, V, \mathcal{S}, \mathcal{D})$	1.44	1.59	1.71	1.82	1.91	2
Dolev et al.	—	—	2	2	3	3

Table 2: Convergence Rate for Specific Values of c

5 A Simple and Efficient Randomized Agreement Protocol

We give a randomized agreement protocol that tolerates Byzantine processor faults—the worst kind of processor fault commonly investigated. This protocol operates in a synchronous system of n processors, at most t of which may fail. It reaches agreement in $O(t/\log n)$ expected rounds and $O(n^2 t/\log n)$ expected message bits independent of the distribution of processor faults. This performance is further improved to a constant expected number of rounds and $O(n^2)$ expected message bits if the distribution of processor faults is assumed to be uniform. In either event, the protocol improves on the known lower bound on rounds for deterministic protocols. Some other advantages are that the protocol requires no cryptographic techniques, that the amount of local computation is small, and that the expected number of random bits used per processor is only two.

1. Introduction

We present a synchronous randomized protocol that solves the agreement problem in a distributed system of n processors, at most t of which may fail with Byzantine faults. It terminates in an expected $O(t/\log n)$ rounds and works for any $n \geq 3t+1$. This contrasts with the lower bound of $t+1$ rounds for deterministic protocols that was shown by Fischer and Lynch [26]. Other advantages of our protocol are the following. It is simple and efficient enough to be of practical use.

This chapter is joint work with Benny Chor and was published in a different form in *IEEE Transactions on Software Engineering* [9].

It terminates faster than any possible deterministic protocol. It operates for small, practical values of n and t. It does not require any cryptographic techniques.

In the model in which it operates, if $n = O(t)$ our protocol has the smallest expected running time of any known agreement protocol. In other models there are many alternative protocols (see [2], [6], [7], [24], and [43]). Most of these protocols have performance that is better than our protocol, but operate under more benign failure assumptions or with n much larger than t. We compare our protocol with these alternatives in Section 6 of this chapter.

It is useful to consider the total number of processors that is required to tolerate t processor faults. This motivates us to define the *redundancy* of a system of n processors as $(n - 1)/t$. Lamport, Shostak, and Pease [34] have shown that no deterministic non-authenticated protocol is possible unless the redundancy is at least 3. An easy extension of their proof shows that this same amount of redundancy is required for randomized protocols. It is desirable to minimize the redundancy in a system in order to reduce the cost of hardware. The randomized agreement protocol that we present operates for any redundancy of 3 or more. We also present a variant protocol that uses only half the rounds used by the basic protocol but that requires a redundancy of 5 or more.

We assume that each processor is equipped with a primitive mechanism for making private random choices, but there is no primitive mechanism for making a random choice visible to all correct processors. If we were to postulate the existence of some mechanism for making a global random choice, then agreement could be reached in a constant expected number of rounds using techniques developed by Ben-Or [2] and by Rabin [43]. A central component of the known randomized agreement protocols is the synthesis of some mechanism for making global random choices. Ben-Or and Rabin each have different approaches to producing global random choices. Each of their approaches incurs certain costs.

Ben-Or uses a scheme that works well when the redundancy is high but that has a low probability of success when the redundancy is low. In his scheme, each processor chooses a random bit locally. If one result predominates by a sufficient margin then a global random choice has been produced; otherwise, no usable global random choice has been produced. A spread of sufficient size is likely only when the total number of processors is large relative to the number of faults. As a consequence, his protocol requires either a large (exponential) number of rounds or a high amount of redundancy (equal to $\sqrt{n}$).

Rabin produces global random choices efficiently; however, he assumes a different, more powerful model of computation. His global random choices are precomputed by a trusted dealer that splits the results of each global random choice so that $t + 1$ processors can determine the result, but t processors have no information. The distributed, split global random choices are an expensive resource that is partially consumed at each execution of the protocol. In some applications, it may be unrealistic to assume that a trusted dealer exists. Rabin's protocol is a good option in those applications where his model seems realistic.

The protocol that we will describe generates global random choices more efficiently than Ben-Or's protocol and uses these global random choices in a way similar to Ben-Or's scheme. Our protocol requires none of the extra machinery of Rabin's protocol. In particular, no reliable, trusted dealer is required. One way in which our technique is inferior to theirs is that ours works only in synchronous systems while theirs work in both synchronous and asynchronous systems. Our technique does not generalize to the asynchronous case.

A contribution of this chapter is our new technique for generating global random choices that, while not perfect, are of sufficient quality to permit our protocol to make rapid progress. Our technique works roughly as follows. For each global random choice that is attempted, a small group of g processors is assigned the task of choosing. Each processor in this group chooses a random bit locally and broadcasts the result. The global random choice generated by this group is the majority of individual outcomes. If more than half the processors are faulty, they can bias the global choice any way they want or cause processors to see conflicting results. But, if fewer than half are faulty, there is a sufficiently large probability that at least $(g + 1)/2$ correct processors will choose the same bit (provided g is not too big). If this happens then the majority is determined regardless of what the faulty processors do. With no more than t faulty processors, there are fewer than $2t/g$ disjoint groups with a majority of faulty processors. After at most that many attempts, a group with a majority of correct processors will be reached. We will show that this leads to fast termination of the protocol.

The requirements that we impose on our global random choices are weak. We do not require that the correct processors always agree on the result, nor do we require that the result is always random. What we do require is that if the global random choice is attempted often enough, then there will be one result that is random and that is observed by all correct processors.

We now give an outline of the remainder of the chapter. In Section 2 we define the randomized agreement problem. In Section 3 we describe the model in which we will solve the randomized agreement problem. In Section 4 we present our new randomized agreement protocol, prove that the protocol is correct, and analyze the performance of the protocol. In Section 5 we present various optimizations of our protocol. In Section 6 we compare our protocol with the alternatives.

2. The Randomized Agreement Problem

Randomized agreement protocols are run by a distributed system of n processors, at most t of which may fail. We let $N = \{1, \ldots, n\}$. Each processor starts the protocol with an input value v from a fixed set V of legal inputs. If, in some execution of the protocol, v_p is the input to processor p for all $p \in N$ then we say that $\langle v_1, \ldots, v_n \rangle$ is the *input vector* for that execution. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on an element of V as its answer. There are three conditions that the correct processors must satisfy.

- *Agreement condition:* All correct processors that decide reach the same decision.
- *Validity condition:* If all correct processors start the protocol with input v then v is the decision of all of the correct processors that decide.
- *Termination condition:* For any adversary and for any input vector, the probability that all correct processors decide by round r tends to 1 as r tends to infinity.

3. The Model

Communication is over a network that is fully connected and reliable. The computation takes place in a series of rounds. In each round each correct processor first makes a local random choice, then sends messages, then receives messages, and finally makes a local state change. Correct processors make their local random choices fairly and send messages according to their programs. Failed processors can send arbitrary messages.

In a real system, we may not be sure what failure modes may occur. Our goal is to show that, for a wide range of possible failures, our protocol works correctly. We do this by imagining a powerful adversary that can select the components of the system that fail and that can control the behavior of the failed components. We achieve our goal by proving that, no matter what the adversary does, our protocol behaves correctly.

It should be clear that we cannot allow our adversary unlimited capabilities. If we did, the adversary could cause all of the components in the system to fail and no protocol would be possible. In the remainder of this subsection we describe the capabilities of our adversary.

The principal capability of our adversary is that it can select which processors fail. We define this as *subverting* a processor. There are three aspects of the subversion of processors on which we will elaborate: the selection of which processors to subvert, the control of subverted processors, and the computational resources available to the adversary in developing its strategy.

During the execution of the agreement protocol, the adversary can dynamically select which processors to subvert (up to the limit of t faulty processors). The selection can be based on the following: the code executed by the various processors, the messages previously sent by any of the processors, the internal state of the processors, and the previous random choice of the processors. We require, however, that the adversary select the processors that it will subvert in round r at the beginning of round r. Specifically, we do not allow it to decide which processors to subvert in round r based on the results of any random choices made by any processors in round r or later.

The messages that are sent by failed processors are under the control of the adversary. The messages that are sent in round r by failed processors can be based on any of the following: the code, the current (round r) and prior state of any of the correct or faulty processors, and the current (round r) and prior random choices of any of the correct or faulty processors. However, the messages of the failed processors in round r cannot be based on future (round $r + 1$ or later) random choices.

We assume that the adversary can use the optimal strategy based on the information currently available. There is no requirement that this strategy be efficiently computable or even computable. This is in contrast to the assumption commonly (and necessarily) made in analyzing cryptographic protocols that the adversary is limited to a polynomial amount of computation. The only limitation we impose is that the strategy of the adversary not be based on any ability to predict accurately the outcome of future random choices.

In any execution of a protocol, we let $C_r \subset N$ be the set of processors that are correct during round r. Recall that the adversary is constrained when it selects which processors fail. It must make its selection at the start of each round. Thus in any execution the set C_r is fixed at the start of round r before any messages are sent and before any local random choices are made. We say that processor p is correct if there is no r such that $p \in C_r$. For the remainder of this chapter we will use the C_r notation.

4. Solution to the Randomized Agreement Problem

We give our protocol for the randomized agreement problem, prove that it is correct, and analyze its performance.

4.1 The Randomized Agreement Protocol

For simplicity, the protocol given here is binary (reaches agreement on one bit). It can easily be extended to be multivalued (reach agreement on arbitrary values) using the technique given in Section 2 of Chapter 3.

The protocol is parameterized by g, the group size; n, the number of processors; and t, the number of faults tolerated. We require that $n \geq 3t+1$, $t \geq 1$, g is odd, and $n \bmod g \leq n - 2t$. Processors are assigned to $\lfloor n/g \rfloor$ disjoint groups of size g. If n is not evenly divisible by g then there are some processors that belong to no group. We index the groups by $\{1, \ldots, \lfloor n/g \rfloor\}$. An arbitrary processor p is assigned to a group by the function

$$\text{group}(p) = \begin{cases} 1 + \lfloor (p-1)/g \rfloor & \text{if } p \leq g \cdot \lfloor n/g \rfloor; \\ \perp & \text{otherwise.} \end{cases}$$

If $\text{group}(p) = \perp$ then processor p belongs to no group.

The protocol is organized as a series of *blocks*. Each block consists of two rounds, an odd numbered round followed by an even numbered round. The block number is used to select one group to play a special role in the current global random choice. The special group in block b is the group active(b), where active is defined to be the function

$$\text{active}(b) = 1 + (b-1) \bmod \lfloor n/g \rfloor.$$

As the block number b increases the function active(b) cycles through the groups. We say that processor p is *active* in block b if $\text{group}(p) = \text{active}(b)$.

```
    Initialization for processor $p$:
        VAL $\leftarrow$ the initial value of processor $p$
    Code for processor $p$ in round $r$:
1.      if $r$ is even and group($p$) = active($r/2$)
2.          then LOCAL $\leftarrow$ PICK()
3.          else LOCAL $\leftarrow \bot$
4.      broadcast (VAL, LOCAL)
5.      receive ($\text{MSG}_q, \text{BIT}_q$) from processor $q$ for $1 \leq q \leq n$
6.      let ANS be the most frequent non-$\bot$ message among the $\text{MSG}_q$
            (break ties arbitrarily)
7.      let NUM be the number of occurrences of ANS
8.      if $r$ is odd then
9.          if NUM $\geq n - t$ then VAL $\leftarrow$ ANS else VAL $\leftarrow \bot$
10.     if $r$ is even then
11.         let GLOBAL be the most frequent non-$\bot$ message among the $\text{BIT}_q$ for
                $q \in \{u \mid \text{group}(u) = \text{active}(r/2)\}$ (break ties arbitrarily)
12.         if NUM $\geq n - 2t$ then VAL $\leftarrow$ ANS else VAL $\leftarrow$ GLOBAL
13.         if NUM $\geq n - t$ and have not decided yet then decide VAL
```

Protocol 1: A Randomized Agreement Protocol

Our randomized agreement protocol is given as Protocol 1. We give an informal description of the protocol before proving it correct. We describe the behavior of an arbitrary correct processor p in a block b that consists of rounds r and $r + 1$. In round $r + 1$ all of the correct active processors locally choose a random bit (by calling PICK) and broadcast the result. Processor p sets GLOBAL, its estimate of the result of the global random choice, to be the most frequent non-$\bot$ bit that it receives from the active processors. The variable VAL holds the value, if any, that processor p currently *prefers* as its answer. In each round processor p *votes* for (*i.e.*, sends a message containing) VAL and then updates its preference based on the votes it receives and based on other information available to it. In round r the number of values preferred by correct processors is reduced to at most one—call it the b-*persistent* value. The protocol ensures that in round $r+1$ no correct processor votes for any value other than the b-persistent value. In round $r + 1$ if processor p receives at least $n - 2t$ votes for some value v, which must be the unique b-persistent value, then it sets VAL to v; otherwise, it sets VAL to GLOBAL. If processor p gets at least $n - t$ round $r + 1$ votes for some v then it decides v—all other processors will receive at least $n - 2t$ votes for v and, as we will show in Lemma 2, all correct processors will decide v by round $r + 3$.

4.2 Proof of Correctness

In the following proof of Protocol 1 we append two subscripts to each variable from the protocol. The first subscript, say r, is a natural number and the second subscript, say p, is in N. By this notation we mean the following. If $r \geq 1$ we mean the value of the subscripted variable at processor p at the *end* of round r. If $r = 0$ we mean the value of the subscripted variable at processor p at the *start* of round 1. For example, if $r \geq 1$ then $\text{VAL}_{r,p}$ is the value of variable VAL at processor p at the end of round r.

We say that v is a *value* if $v \in \{0, 1\}$. Specifically, $\perp$ is not a value. In any execution of Protocol 1, for all $b \geq 1$ and for $r = 2b - 1$, the value v is *b-persistent* if there is some correct processor p such that $\text{VAL}_{r,p} = v$. For all $b \geq 1$ and for all $v \in \{0, 1\}$, we say that v is *b-good* if either v is equal to a b-persistent value or there is no b-persistent value. For all r we say that processor p *votes* for value v in round r if it sends any round r message containing v as its first component. A correct processor votes for at most one value in each round, but a faulty processor may vote for many values by sending conflicting votes to different recipients. If there is some even r and some v such that $\text{GLOBAL}_{r,p} = v$ for all correct processors p, then we say that v is the *result* of the global random choice performed in block $r/2$; otherwise we say that the result is $\perp$ (undefined).

Lemma 1: *For any $b \geq 1$, there is at most one b-persistent value and there is at least one b-good value.*

Proof: We show each of the two claims.

Proof that there is at most one b-persistent value: Assume not. Let $r = 2b - 1$. Then, there are values v and v' and correct processors p and q such that $\text{VAL}_{r,p} = v \neq v' = \text{VAL}_{r,q}$. In round r processor p must have received at least $n - t$ votes for value v and processor q must have received at least $n-t$ votes for value v'. Therefore, at least $n - 2t \geq t + 1$ processors including at least one correct processor voted for both v and v'. This is impossible behavior for a correct processor, contradiction.

Proof that there is at least one b-good value: This is immediate from the first part of Lemma 1 and from the definition of b-good value. □

Lemma 2: *If there is some value v and some $r \geq 0$ such that r is even and $\text{VAL}_{r,p} = v$ for all correct processors p then any correct processor that has not decided by round r will decide v in round $r + 2$.*

Proof: There are at least $n-t$ correct processors that all broadcast v in round $r+1$. In round $r+1$ all correct processors receive at least $n-t$ votes for v and at most t votes for any $v' \neq v$. Therefore all correct processors broadcast v in round $r+2$. In round $r+2$ all correct processors receive at least $n-t$ votes for v and at most t votes for any $v' \neq v$. Therefore, any correct processor that has not previously decided will decide v in round $r+2$. □

Lemma 3: *For all $b \geq 1$, if the result of the block b global random choice is a b-good value then all correct processors decide by round $2b+2$.*

Proof: Let b be an arbitrary block and let $r = 2b$. There are two cases.

Case 1: (There is a b-persistent value.) Let v be the b-persistent value, which is unique by Lemma 1. By the definition of b-good, v is the only b-good value. In round r any correct processor receives at most t votes for any value $v' \neq v$. Thus in step 12 every correct processor p either sets $\text{VAL}_{r,p}$ to v or to $\text{GLOBAL}_{r,p}$. If the result of the block b global random choice is v, then $\text{VAL}_{r,p} = v$ for all correct processors p and by Lemma 2 all correct processors decide by round $r+2 = 2b+2$.

Case 2: (There is no b-persistent value.) In round r any correct processor receives at most t votes for any value v. Thus in step 12 an arbitrary correct processor p sets $\text{VAL}_{r,p}$ to $\text{GLOBAL}_{r,p}$. If the result of the block b global random choice is any b-good value $v \neq \perp$, then $\text{VAL}_{r,p} = v$ for all correct processors p and by Lemma 2 all correct processors decide by round $r+2 = 2b+2$. □

We now give the probability model that we will use to analyze the probability of termination and the expected running time of Protocol 1. Our sample space consists of all of the infinite strings over the alphabet $\{0,1\}$. The character in position i of a string represents the local random choice made by processor p in round r where $i = p + (r-1) \cdot n$. Given an input vector, an adversary, and a point in the sample space, the behavior of all processors (correct and faulty) in all rounds is determined. Our analysis is for an arbitrary, fixed adversary and input vector. For all $b \geq 1$, let s_b be the event that the result of the block b global random choice is a b-good value and let f_b be the complement of s_b. For all $r \geq 1$, let d_r be the event that all correct processors decide by round r.

In the remainder of this chapter T, R, and A are random variables. Let T be the smallest b such that the event s_b occurs. Let R be the smallest r such that the event d_r occurs. Let

$$\mathcal{A} = \{\langle a_1, \ldots, a_{\lfloor n/g \rfloor}\rangle \mid a_i \geq 0 \text{ for all } i \text{ and } t = \textstyle\sum_{i=1}^{\lfloor n/g \rfloor} a_i\}.$$

Let the random variable A be $\langle a_1, \ldots, a_{\lfloor n/g \rfloor} \rangle \in \mathcal{A}$ if the total number of processors in group j that fail is a_j for all $j \in \{1, \ldots, \lfloor n/g \rfloor\}$. For all $b \geq 1$ and for all $a \in \mathcal{A}$, let $q(b, a) = \Pr\{f_b \mid f_{b-1} \wedge \ldots \wedge f_1 \wedge A = a\}$.

Recall that g is the number of processors in each group and that g is required to be odd. Let m denote the number of processors that constitute a majority of a group. Thus $g = 2m - 1$.

Lemma 4: *For all* $b \geq 1$

$$\Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b\} \leq \max_{a \in \mathcal{A}} q(1, a) \cdot q(2, a) \cdots q(b, a).$$

Proof: We calculate that

$$\begin{aligned}
\Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b\} &= \sum_{a' \in \mathcal{A}} \Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b \mid A = a'\} \cdot \Pr\{A = a'\} \\
&\leq \sum_{a' \in \mathcal{A}} \left(\max_{a \in \mathcal{A}} \Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b \mid A = a\}\right) \cdot \Pr\{A = a'\} \\
&= \left(\max_{a \in \mathcal{A}} \Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b \mid A = a\}\right) \cdot \sum_{a' \in \mathcal{A}} \Pr\{A = a'\} \\
&= \max_{a \in \mathcal{A}} \Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b \mid A = a\}.
\end{aligned}$$

We simplify $\Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b \mid A = a\}$ as follows

$$\begin{aligned}
\Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b \mid A = a\} &= \prod_{i=1}^{b} \Pr\{f_i \mid f_{i-1} \wedge \ldots \wedge f_1 \wedge A = a\} \\
&= q(1, a) \cdot q(2, a) \cdots q(b, a).
\end{aligned}$$

Using this expression for $\Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b \mid A = a\}$, we calculate that

$$\Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b\} \leq \max_{a \in \mathcal{A}} q(1, a) \cdot q(2, a) \cdots q(b, a). \qquad \square$$

Lemma 5: *For all* $b \geq 1$ *and for all* $a \in \mathcal{A}$

$$q(1, a) \cdot q(2, a) \cdots q(b, a) \leq (1 - 1/2^m)^{\lfloor b/n \rfloor}.$$

Proof: Suppose $a = \langle a_1, \ldots, a_{\lfloor n/g \rfloor} \rangle$. By assumption, $n \bmod g \leq n - 2t$. So

$$\begin{aligned}
n - (n \bmod g) &\geq n - (n - 2t) \\
g \cdot \lfloor n/g \rfloor &\geq 2t \\
m \cdot \lfloor n/g \rfloor &> t
\end{aligned}$$

Therefore, by the pigeon-hole principle, there is some group, say group i, that contains at least m correct processors (*i.e.*, $a_i < m$). For all b, if active$(b) = i$ and $f_1 \wedge f_2 \wedge \cdots \wedge f_{b-1}$ then, with probability at least $1/2^m$, there are m correct processors in group i that all toss a b-good value. (By Lemma 1 there is at least one b-good value.) Group i is active at least once every n blocks. Thus, $q(1,a) \cdot q(2,a) \cdots q(b,a) \leq (1 - 1/2^m)^{\lfloor b/n \rfloor}$. □

In Theorem 6 we prove the correctness of Protocol 1. The analysis of the expected running time of the protocol follows in Subsection 4.3.

Theorem 6: *Protocol 1 solves the randomized agreement problem.*

Proof: We show that the agreement, validity, and termination conditions are satisfied.

Agreement condition: Suppose that round r is the first round in which a correct processor decides and suppose that correct processor p decides v in round r. Clearly, r is even. In round r processor p got $n - t$ votes for v. At least $n - 2t$, which is at least one, were from correct processors, so v is the $(r/2)$-persistent value, which is unique by Lemma 1. Therefore, in round r all correct processors get at least $n - 2t$ votes for v and at most t votes for any $v' \neq v$. Thus $\text{VAL}_{r,q} = v$ for any correct processor q. So, any correct processor that decides in round r decides v. By Lemma 2, all of the other correct processors decide v in round $r + 2$.

Validity condition: Suppose all correct processors start the protocol with input v. It is immediate that $\text{VAL}_{0,p} = v$ for all correct processors p. By Lemma 2, all correct processors decide v in round 2.

Termination condition: Recall that d_r is the event that all correct processors decide by round r. We calculate

$$\begin{aligned}
\lim_{r\to\infty} \Pr\{d_r\} &\geq \lim_{b\to\infty} \Pr\{s_1 \vee s_2 \vee \ldots \vee s_b\} && \text{By Lemma 3.}\\
&= 1 - \lim_{b\to\infty} \Pr\{f_1 \wedge f_2 \wedge \ldots \wedge f_b\} \\
&\geq 1 - \lim_{b\to\infty} (1 - 1/2^m)^{\lfloor b/n \rfloor} && \text{By Lemmas 4 and 5.}\\
&= 1
\end{aligned}$$

□

We are now in a position to explain why randomization is necessary for the correct operation of our protocol. By Lemma 3, in any block b, if the result of the block b global random choice is a b-good value then all correct processors will decide by the end of block $b+1$. The set of b-good values can, however, be determined by the adversary in the first round of block b. If the random choice were replaced with some predetermined value, say v_b, then the adversary might be able to cause the set of b-good values to be $\{0,1\} - \{v_b\}$. In our scheme, this strategy is unavailable to the adversary because the random choice is not performed until the second round of block b—after the identity of the set of b-good values has already been fixed.

4.3 Analysis of the Protocol

Using the probability model defined in Subsection 4.2, we analyze the computational resources used by Protocol 1 under the assumption that $g = 2 \cdot \lfloor \frac{1}{2} \log n \rfloor - 1$. (We first use this assumption in Lemma 13.) We show that the expected number of rounds to reach agreement is $O(t/\log n)$, the expected number of message bits sent by all correct processors is $O(n^2 t/\log n)$, and the expected number of random bits that each processor is required to generate locally is less than 2. We remark that the amount of local computing resources used by each correct processor is small.

Lemma 7:

$$\mathrm{Exp}[T] \leq \max_{a \in \mathcal{A}} \mathrm{Exp}[T \mid A = a]$$

Proof: We calculate that

$$\begin{aligned}
\mathrm{Exp}[T] &= \sum_{a' \in \mathcal{A}} \mathrm{Exp}[T \mid A = a'] \cdot \Pr\{A = a'\} \\
&\leq \sum_{a' \in \mathcal{A}} \left(\max_{a \in \mathcal{A}} \mathrm{Exp}[T \mid A = a] \right) \cdot \Pr\{A = a'\} \\
&= \left(\max_{a \in \mathcal{A}} \mathrm{Exp}[T \mid A = a] \right) \cdot \sum_{a' \in \mathcal{A}} \Pr\{A = a'\} \\
&= \max_{a \in \mathcal{A}} \mathrm{Exp}[T \mid A = a].
\end{aligned}$$

□

Lemma 8: *For all $a \in \mathcal{A}$*

$$\mathrm{Exp}[T \mid A = a] \leq 1 + \sum_{b=1}^{\infty} q(1,a) \cdot q(2,a) \cdots q(b,a).$$

Proof: Let a be an arbitrary element of $\mathcal{A}$. By the definition of expected value

$$\mathrm{Exp}[T \mid A = a] = \sum_{b=1}^{\infty} b \cdot \Pr\{s_b \wedge f_{b-1} \wedge \ldots \wedge f_1 \mid A = a\}.$$

We simplify $\Pr\{s_b \wedge f_{b-1} \wedge \ldots \wedge f_1 \mid A = a\}$ as follows

$$\begin{aligned}
&\Pr\{s_b \wedge f_{b-1} \wedge \ldots \wedge f_1 \mid A = a\} \\
&\quad = \Pr\{s_b \mid f_{b-1} \wedge \ldots \wedge f_1 \wedge A = a\} \cdot \Pr\{f_{b-1} \wedge \ldots \wedge f_1 \mid A = a\} \\
&\quad = \Pr\{s_b \mid f_{b-1} \wedge \ldots \wedge f_1 \wedge A = a\} \cdot \prod_{i=1}^{b-1} \Pr\{f_i \mid f_{i-1} \wedge \ldots \wedge f_1 \wedge A = a\} \\
&\quad = (1 - q(b,a)) \cdot q(b-1,a) \cdot q(b-2,a) \cdots q(1,a).
\end{aligned}$$

Using this expression for $\Pr\{s_b \wedge f_{b-1} \wedge \ldots \wedge f_1 \mid A = a\}$, we calculate that

$$\begin{aligned}
\mathrm{Exp}[T \mid A = a] &= \sum_{b=1}^{\infty} b \cdot q(1,a) \cdot q(2,a) \cdots q(b-1,a) \cdot (1 - q(b,a)) \\
&= \sum_{b=1}^{\infty} b \cdot q(1,a) \cdots q(b-1,a) - \sum_{b=1}^{\infty} b \cdot q(1,a) \cdots q(b,a) \\
&= 1 + \sum_{b=1}^{\infty} q(1,a) \cdot q(2,a) \cdots q(b,a). \qquad \Box
\end{aligned}$$

In order to bound the series q with a periodic series $\hat{q}$, we make the following definition. For all $b \geq 1$ and for all $a = \langle a_1, \ldots, a_{\lfloor n/g \rfloor} \rangle$ in $\mathcal{A}$, let

$$\hat{q}(b,a) = \begin{cases} 1 & \text{if } c < m; \\ 1 - \sum_{i=m}^{c} \binom{c}{i} \left(\frac{1}{2}\right)^c & \text{otherwise,} \end{cases}$$

where $c = g - a_{\mathrm{active}(b)}$. The significance of this definition is that c is a lower bound on the number of correct processors in group active(b) and $\sum_{i=m}^{c} \binom{c}{i} \left(\frac{1}{2}\right)^c$ is the probability that at least m correct processors out of a group of c correct processors will select a particular predetermined value.

Lemma 9: *For all $b \geq 1$ and for all $a \in \mathcal{A}$ it is the case that $q(b,a) \leq \hat{q}(b,a)$.*

Proof: Consider an arbitrary $b \geq 1$ and an arbitrary $a = \langle a_1, \ldots, a_{\lfloor n/g \rfloor} \rangle$ in $\mathcal{A}$. Let $c = g - a_{\mathrm{active}(b)}$. There are two cases. If $c < m$ then $q(b,a) \leq 1 = \hat{q}(b,a)$. For the remainder of this proof we assume that $c \geq m$.

By definition $q(b,a) = \Pr\{f_b | f_{b-1} \wedge \ldots \wedge f_1 \wedge A = a\}$. So we are considering only those executions for which $A = a$. In all of these executions there are c processors in group active(b) that are correct throughout the execution. Let r be the first round of block b. Clearly, there are at least c processors in group active(b) that are correct during round r. Select the lexicographically least set of c processors that are in group active(b) and correct during round r. By assumption, the adversary has no knowledge of any round r random choices when it selects which processors fail in round r. Using this limitation on the adversary and the existence of a b-good value for all $b \geq 1$ (shown in Lemma 1), we proceed to calculate a bound on $q(b,a)$.

In a collection of exactly c correct processors the conditional probability that exactly i correct processors choose a b-good value given that no previous group has chosen a good value is at least $\binom{c}{i}\left(\frac{1}{2}\right)^c$. So, in a group with at least c correct processors the conditional probability that at least m correct processors choose a b-good value given that no previous group has chosen a good value is at least $\sum_{i=m}^{c}\binom{c}{i}\left(\frac{1}{2}\right)^c$. Because $q(b,a)$ is the probability of the complement of this event, we have that

$$q(b,a) \leq 1 - \sum_{i=m}^{c}\binom{c}{i}\left(\tfrac{1}{2}\right)^c = \hat{q}(b,a). \qquad \square$$

Lemma 10: *For all $a \in \mathcal{A}$*

$$\operatorname{Exp}[T \mid A = a] \leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}(1,a)\cdots\hat{q}(b,a)\right)\left(1 + \sum_{i=1}^{\infty}(\hat{q}(1,a)\cdots\hat{q}(\lfloor n/g \rfloor, a))^i\right).$$

Proof: This is immediate from Lemmas 8 and 9 and from the periodicity of $\hat{q}$. $\square$

In Section 4.4 we will use Lemmas 7 and 10 to calculate explicit bounds on Exp[T] for particular small values of n and t. We now proceed to calculate a closed-form bound on Exp[T]. As a first step we find it convenient to make the following definition. Let

$$\hat{q}_i = \begin{cases} 1 & \text{if active}(i) \leq \lfloor t/m \rfloor; \\ 1 - 1/2^m & \text{otherwise.} \end{cases}$$

Lemma 11:

$$\operatorname{Exp}[T] \leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b\right) + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b\right)\left(\sum_{i=1}^{\infty}(\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i\right)$$

Proof: Let $a = \langle a_1, \ldots, a_{\lfloor n/g \rfloor}\rangle$ be an arbitrary element of $\mathcal{A}$. If $m \leq c \leq 2m-1$

$$\begin{aligned}
\sum_{i=m}^{c}\binom{c}{i}\left(\frac{1}{2}\right)^c \geq \binom{c}{m}\left(\frac{1}{2}\right)^c &= \left(\prod_{i=1}^{c-m}\frac{i+m}{i}\right)\left(\frac{1}{2}\right)^c \\
&\geq \left(\prod_{i=1}^{c-m} 2\right)\left(\frac{1}{2}\right)^c \qquad \text{Because } i \leq c-m < m. \\
&= 2^{c-m}\left(\frac{1}{2}\right)^c = \left(\frac{1}{2}\right)^m
\end{aligned}$$

Using this inequality and the definition of $\hat{q}$, we have

$$\hat{q}(i,a) \leq \begin{cases} 1 & \text{if } a_i \geq m; \\ 1 - \left(\frac{1}{2}\right)^m & \text{otherwise.} \end{cases}$$

Recall that $\sum_{i=1}^{\lfloor n/g \rfloor} a_i \leq t$ by the definition of $\mathcal{A}$. Thus $|\{i \mid a_i \geq m\}| \leq \lfloor t/m \rfloor$. So $|\{i \in \{1, \ldots, \lfloor n/g \rfloor\} \mid \hat{q}(i,a) \leq 1 - \left(\frac{1}{2}\right)^m\}| \geq \lfloor n/g \rfloor - \lfloor t/m \rfloor$. Using this, it is straightforward to show that

$$\begin{aligned} \hat{q}(1,a) \cdot \hat{q}(2,a) \cdots \hat{q}(\lfloor n/g \rfloor, a) &\leq (1 - 1/2^m)^{\lfloor n/g \rfloor - \lfloor t/m \rfloor} \\ &= \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor}, \end{aligned}$$

and

$$\begin{aligned} \sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}(1,a) \cdot \hat{q}(2,a) \cdots \hat{q}(b,a) &\leq \sum_{b=1}^{\lfloor t/m \rfloor} 1 + \sum_{b=\lfloor t/m \rfloor + 1}^{\lfloor n/g \rfloor} (1 - 1/2^m)^{b - \lfloor t/m \rfloor} \\ &= \sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b. \end{aligned}$$

Using these inequalities and Lemma 10, we have that

$$\begin{aligned} \operatorname{Exp}[T \mid A = a] &\leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}(1,a) \cdots \hat{q}(b,a) \right) \left(1 + \sum_{i=1}^{\infty} (\hat{q}(1,a) \cdots \hat{q}(\lfloor n/g \rfloor, a))^i \right) \\ &\leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b \right) \left(1 + \sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i \right). \end{aligned}$$

Because this bound holds for an arbitrarily chosen a,

$$\max_{a' \in \mathcal{A}} \operatorname{Exp}[T \mid A = a'] \leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b \right) \left(1 + \sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i \right).$$

By Lemma 7,

$$\operatorname{Exp}[T] \leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b \right) \left(1 + \sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i \right).$$

Rearranging, we have

$$\mathrm{Exp}[T] \leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b\right) + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b\right)\left(\sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i\right),$$

which is what we sought to show. □

Thus $\mathrm{Exp}[T]$ is bounded above by the sum of three terms. The first term is the constant 1. In Lemma 12 we give an upper bound on the second term. In Lemma 14 we show that the third term is $o(1)$.

Lemma 12:

$$\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b < \frac{t}{m} + 2^m - 1$$

Proof: We calculate that

$$\begin{aligned}\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b &= \sum_{b=1}^{\lfloor t/m \rfloor} 1 + \sum_{b=1}^{\lfloor n/g \rfloor - \lfloor t/m \rfloor} (1 - 1/2^m)^b \\ &< \frac{t}{m} + 2^m - 1.\end{aligned}$$ □

The following is a technical lemma used in the proof of Lemma 14.

Lemma 13: *If* $g = 2 \cdot \lfloor \frac{1}{2} \log n \rfloor - 1$ *and* $1 \leq t < \frac{1}{3}n$ *then* $\lfloor n/g \rfloor - \lfloor t/m \rfloor > n/12m$.

Proof: By elementary calculus, $\frac{1}{6}n > (\log n) - 2$ for all $n \geq 1$. So,

$$\tfrac{1}{6}n > (\log n) - 2 \geq 2 \cdot \lfloor \tfrac{1}{2} \log n \rfloor - 2 = g - 1.$$

Using the fact that $\frac{1}{6}n > g - 1$ we calculate

$$\tfrac{5}{6}n < n - (g-1) \leq n - n \bmod g = g \cdot \lfloor n/g \rfloor.$$

Thus $\frac{5}{6}n < g \cdot \lfloor n/g \rfloor$. Using this and the observations that $2m > g$ and that $-2m \cdot \lfloor t/m \rfloor \geq -2t$ we calculate that

$$\begin{aligned}2m \cdot \lfloor n/g \rfloor - 2m \cdot \lfloor t/m \rfloor &> \tfrac{5}{6}n - 2t \\ \lfloor n/g \rfloor - \lfloor t/m \rfloor &> (\tfrac{5}{6}n - 2t)/2m.\end{aligned}$$

Using the bound $\frac{1}{3}n > t$, we have

$$\lfloor n/g \rfloor - \lfloor t/m \rfloor > n/12m.$$ □

Lemma 14: *If* $g = 2 \cdot \lfloor \frac{1}{2} \log n \rfloor - 1$ *then*

$$\left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b \right) \left(\sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i \right) = o(1).$$

Proof: We calculate that

$$\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor} = \left(1 - \frac{1}{2^m}\right)^{\lfloor n/g \rfloor - \lfloor t/m \rfloor}.$$

By Lemma 13,

$$\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor} < \left(1 - \frac{1}{2^m}\right)^{n/12m}.$$

Because $m \leq \frac{1}{2} \log n$, we get

$$\left(1 - \frac{1}{2^m}\right)^{n/12m} \leq \left(1 - \frac{1}{\sqrt{n}}\right)^{n/6 \log n} < \left(\frac{1}{e}\right)^{\sqrt{n}/6 \log n}$$

where $e = 2.718+$, the base of the natural logarithms. Because $n \geq 3t+1$ and $t \geq 1$, we have that $n \geq 4$. It is easy to verify that $\sqrt{n}/6 \log n \geq \frac{1}{6}$ for all $n \geq 4$. Thus, for all $n \geq 4$, we can calculate that $(1/e)^{\sqrt{n}/6 \log n} < \frac{7}{8}$. Using basic properties of geometric series we have that

$$\sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i < 8 \cdot \left(\frac{1}{e}\right)^{\sqrt{n}/6 \log n}.$$

Using Lemma 12 and the bound that $1 \leq m \leq \frac{1}{2} \log n$ it follows that

$$\begin{aligned} \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b \right) \left(\sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i \right) &< \left(\frac{t}{m} + 2^m \right) \sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i \\ &< (t + \sqrt{n}) \cdot 8 \cdot (1/e)^{\sqrt{n}/6 \log n} \\ &< 16n \cdot (1/e)^{\sqrt{n}/6 \log n} \\ &= o(1). \end{aligned}$$

□

In Theorem 15 we prove an upper bound on the expected number of rounds for the last correct processor in Protocol 1 to decide. This bound holds for all adversaries and for all possible inputs to Protocol 1.

Theorem 15: *For group size $g = 2 \cdot \lfloor \frac{1}{2} \log n \rfloor - 1$ in Protocol 1,*

$$\mathrm{Exp}[R] < \frac{4t}{\log n} + 2\sqrt{n} + 2 + o(1) = O\left(\frac{t}{\log n}\right).$$

Proof: By Lemma 11,

$$\mathrm{Exp}[T] \leq 1 + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b\right) + \left(\sum_{b=1}^{\lfloor n/g \rfloor} \hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_b\right)\left(\sum_{i=1}^{\infty} (\hat{q}_1 \cdot \hat{q}_2 \cdots \hat{q}_{\lfloor n/g \rfloor})^i\right).$$

and by Lemmas 12 and 14,

$$\mathrm{Exp}[T] < \left(\frac{t}{m} + 2^m\right) + o(1).$$

Because $\frac{1}{2}\log n - 1 \leq m \leq \frac{1}{2}\log n$, we have that

$$\mathrm{Exp}[T] < \frac{2t}{\log n} + \sqrt{n} + o(1) = O\left(\frac{t}{\log n}\right).$$

By Lemma 3, $\mathrm{Exp}[R] \leq 2 \cdot \mathrm{Exp}[T] + 2$. So, we have that the expected number of rounds is bounded above by $4t/\log n + 2\sqrt{n} + 2 + o(1)$. □

We remark that by setting $m = (1-\epsilon)\log n$, we have

$$\frac{t}{m} + 2^m = \frac{t}{(1-\epsilon)\log n} + 2n^{1-\epsilon}.$$

This means that by taking somewhat larger size groups, the constant 1 in the $O(t/\log n)$ expression for the expected number of blocks can be achieved asymptotically. On the other hand, if the group size is decreased slightly by taking $m = \frac{1}{2}(1-\epsilon)\log n$, we get

$$\mathrm{Exp}[T] < \frac{2t}{(1-\epsilon)\log n} + \sqrt{n^{1-\epsilon}} + o(1).$$

This will be significant for achieving early stopping in case the actual number of faults is very small (see Subsection 5.2 for details).

We calculate the expected number of message bits sent by Protocol 1. Individual messages have a constant size. In each round there are $O(n^2)$ messages sent. By Theorem 15, the expected number of rounds is $O(t/\log n)$. Therefore, over the course of the protocol, an expected $O(n^2 t/\log n)$ message bits are sent. We remark that the message complexity can be improved to $O(nt^2/\log n)$ using relay processors, a technique due to Dolev and Strong [21].

In randomized protocols, the number of random bits used by each processor is important. Current physical devices for producing random bits are rather slow. If a large number of random bits are required, then pseudo-random number generators are often used. Plumstead [42] showed that the fast, linear congruence generators are not secure. After seeing a few outcomes, an adversary can predict the remaining bits. Secure pseudo-random number generators, based on cryptographic techniques, are known to exist under certain intractability assumptions (see [4] and [1]). However, they require a lot of computation, so we are better off if we can avoid using them altogether. A surprising result is the number of random bits used by our protocol. For sufficiently large n, the expected number of global random choices is bounded above by

$$\mathrm{Exp}[T] < \frac{2t}{\log n} + \sqrt{n} + o(1) < \frac{n}{\log n}.$$

There are $n/\log n$ groups. Therefore the expected number of times we cycle through all groups is bounded above by 1 if n is large. At each cycle one random bit is used by each processor. Therefore, the expected number of random bits used by each processor is bounded above by 1. From Table 1 (in the next subsection) we calculate that for smaller values of n the expected number of random bits per processor is bounded above by 2. Slow physical generators are good enough then, and it is not necessary to resort to pseudo-random number generators.

4.4 Performance of the Protocol in Small Systems

The analysis of Theorem 15 is somewhat loose; we believe that the upper bound given for Exp[T] is too big by a constant factor. This looseness comes from the approximations made in the proof of Lemma 11. In this subsection we consider a tighter bound that we calculate explicitly for small systems. We do this to validate our claim that Protocol 1 works well in small, practical systems. In Theorem 16, which follows, we give a tighter open-form bound on Exp[T]. We will use Theorem 16 to calculate some specific values.

Theorem 16:

$$\mathrm{Exp}[T] \le \max_{a \in \mathcal{A}} \frac{1 + \sum_{b=1}^{\lfloor n/g \rfloor - 1} \hat{q}(1,a) \cdot \hat{q}(2,a) \cdots \hat{q}(b,a)}{1 - \hat{q}(1,a) \cdot \hat{q}(2,a) \cdots \hat{q}(\lfloor n/g \rfloor, a)}$$

n	t	g	Exp[T]	n	t	g	Exp[T]
4	1	1	3.2	55	18	5	8.4
7	2	3	4.0	58	19	7	8.2
10	3	3	4.4	61	20	5	9.0
13	4	3	4.7	64	21	7	8.3
16	5	3	5.1	67	22	7	8.9
19	6	3	5.4	70	23	7	8.6
22	7	3	5.9	73	24	7	9.0
25	8	5	5.7	76	25	7	9.5
28	9	5	6.6	79	26	7	9.3
31	10	5	6.3	82	27	7	9.7
34	11	5	7.1	85	28	7	9.7
37	12	5	6.8	88	29	7	10.0
40	13	5	6.9	91	30	7	10.0
43	14	5	7.5	94	31	7	10.4
46	15	5	7.5	97	32	7	10.7
49	16	7	7.5	100	33	9	10.3
52	17	5	8.1	103	34	9	10.9

Table 1: Performance of Protocol 1 in Small Systems

Proof: The proof is straightforward from Lemmas 7 and 10. □

We can view n, g, and t as parameters that influence Exp[T]. In Table 1 we list some values of Exp[T] for small systems. For each system size considered, the table shows the group size that minimizes Exp[T].

5. Extensions

We present various extensions to Protocol 1. In Subsection 5.1 we present an alternative analysis for Protocol 1 for the case where processor failures are assumed to be uniformly randomly distributed. In Subsection 5.2 we explain how the performance of Protocol 1 can be improved when the actual number of faults that occurs in an execution is less than t. In Subsection 5.3 we show that it is possible to transform Protocol 1 into one that achieves simultaneous termination with high probability. In Subsection 5.4 we present an alternative protocol that doubles the speed of Protocol 1 at the cost of requiring somewhat more redundancy.

5.1 Uniformly Distributed Processor Failures

In the previous analysis we assumed that an adversary controls both the selection of which processors fail and the behavior of the failed processors. In this subsection we analyze the performance of our protocol under a weaker adversary. We assume that the distribution of processor faults is uniform. That is, each of the $\binom{n}{t}$ ways of distributing t faults among the n processors is equally likely. We retain the assumption that failed processors are under the control of the adversary and therefore behave in a way that will cause our protocol the most difficulty.

In Theorem 17 we show that for $g = 1$ and uniform processor faults our protocol terminates in a constant expected number of rounds. Because $g = 1$, each group consists of a single processor.

Theorem 17: *If the group size, g, is* 1 *and processor faults are uniformly distributed, then* $\text{Exp}[R] \leq 8$.

Proof: By Lemma 3, $\text{Exp}[R] = 2 \cdot \text{Exp}[T] + 2$. We show that $\text{Exp}[T] \leq 3$. For all $b \geq 1$, the probability that the block b random choice of a correct processor is a b-good value is at least $\frac{1}{2}$ (by Theorem 6 and the fairness of the choice). The probability that processor p is correct is at least $\frac{2}{3}$ because $n \geq 3t+1$. Therefore, the probability that the random choice of processor p is a b-good value is at least $\frac{1}{3}$. The conditional probability that the random choice of processor p is a b-good value given that previous random choices have been bad is at least as great as the unconditional probability (at least $\frac{1}{3}$). Therefore, $\text{Exp}[T]$ is at most the mean of a geometric random variable with parameter $\frac{1}{3}$, which is 3. □

We calculate the expected number of message bits sent by the protocol. Individual messages have a constant size. On each round, there are $O(n^2)$ messages sent. By Theorem 17, the expected number of rounds is 8. Therefore, over the course of the protocol, an expected $O(n^2)$ message bits are sent. This can be reduced to $O(nt)$ using the relay-processor technique of Dolev and Strong [21].

Termination in a constant expected number of rounds is an attractive feature of the protocol; however, this feature is not unique to randomized protocols. Under the same assumption of uniform randomly distributed processor failures, a deterministic protocol due to Reischuk [44] also terminates in a constant expected number of rounds.

5.2 Early Stopping

Our protocol is resilient to t faults, but the actual number of faulty processors, f, might be smaller than the upper bound t. A desirable property of any agreement protocol is that agreement be reached early in this case. Dolev, Reischuk, and Strong [20] have studied early termination for deterministic agreement protocols. From the analysis in Subsection 4.3, it follows that the expected number of rounds to reach agreement in the presence of f faults is bounded above by $4f/\log n + 2\sqrt{n} + o(1)$. Thus early stopping is automatically achieved. Furthermore, for the range $\sqrt{n}\log n \leq f < n/3$, agreement is reached in $O(f/\log n)$ rounds.

We can modify Protocol 1 to get uniform $O(f/\log n)$ expected time for termination for all values of f. To do that, we slightly decrease the size of each group by setting $m = \frac{1}{2}(1-\epsilon)\log n$. This yields

$$\mathrm{Exp}[T] < \frac{2f}{(1-\epsilon)\log n} + \sqrt{n^{1-\epsilon}} + o(1).$$

If $f > \sqrt{n}$, then $\sqrt{n^{1-\epsilon}}$ is $o(f/\log n)$ and agreement is reached in $O(f/\log n)$ rounds.

For f in the range $f \leq \sqrt{n}$, the standard deviation of the number of ones in $n - f$ independent unbiased random bits is larger than f. If we take $g = n$ (all processors are in the same group), then with a constant probability (about 1/3) the deviation from the average of the number of ones in the $n - f$ random bits chosen by correct processors exceeds f. Therefore, there is a constant probability that termination will be reached at the end of each block. Thus the expected run time in this case is constant. (This idea is similar to the one used in [2] and [7]).

The situation now is that large values of f can be handled efficiently by using small groups ($g = (1-\epsilon)\log n$), and small values of f can be handled efficiently by using large groups ($g = n$). However, we want our protocol to handle both cases, without knowing in advance which one it faces. In order to accommodate both small and large values of f, we will alternate between the small groups and the large ones. In the even blocks, the global random choices will be performed by groups of size $(1-\epsilon)\log n$. In the odd blocks, the global random choices will be performed by groups of size n.

To analyze the run time, we distinguish between the two cases. If f is large ($f > \sqrt{n}$), we might as well assume that the odd blocks are useless and contribute nothing to termination. However, the expected time to reach agreement in this case is at most twice the number of even blocks to reach agreement, which is $O(f/\log n)$. Similarly, if $f \leq \sqrt{n}$, then the expected number of odd blocks to reach agreement is $O(1)$. By alternating between odd and even blocks, we only increase the hidden constants in these expressions by a factor of 2. Thus, our interleaved protocol yields the following expression for $\mathrm{Exp}[T]$, the expected number of rounds to reach agreement:

$$\mathrm{Exp}[T] \leq \begin{cases} O(f/\log n) & \text{if } f > \sqrt{n}; \\ O(1) & \text{otherwise.} \end{cases}$$

5.3 Simultaneous Termination

One disadvantage of our protocol is that even though all correct processors start the protocol in exactly the same round, they might decide in different rounds. In this subsection we show that a minor modification of Protocol 1 yields an almost certain simultaneous termination, that is, all correct processors decide in exactly the same round with overwhelming probability. This is done without violating the agreement, validity, and termination requirements, which will still be achieved with probability 1. The expected running time is changed only by a small multiplicative constant.

The modification is quite simple. We know that the expected number of blocks to reach agreement in Protocol 1 is at most $2t/\log n + \sqrt{n} + o(1)$ and that the tail probability for the number of blocks converges rapidly. For example, the probability that more than $3t/\log n$ blocks will be needed is no greater than $(1/e)^{\sqrt{n}/6\log n}$. In the modified protocol, each correct processor will just delay its decision until block $3t/\log n$ (if it would have made its decision before that block) and behave as in Protocol 1 otherwise. This guarantees simultaneous termination by block $3t/\log n$ with probability at least $1 - (1/e)^{\sqrt{n}/6\log n}$. Greater confidence of simultaneous termination can be achieved at the cost of more rounds.

In this subsection we have shown that there is a randomized agreement protocol that has an expected running time of $O(t/\log n)$ rounds and that achieves simultaneous termination with very high probability. This result contrasts sharply with the lower bound proved in Chapter 6; there we show that any randomized agreement protocol that achieves simultaneous termination with probability 1 must necessarily have an expected running time of at least $t + 1$ rounds.

5.4 An Alternative Protocol

The randomized protocol presented in Subsection 4.1 requires redundancy of 3 and uses two rounds per block. If the system redundancy is increased to 5 then one round per block suffices. Thus the expected number of rounds is cut by a factor of 2. The code for this case ($n \geq 5t + 1$) is given as Protocol 2.

In the following discussion and proof of Protocol 2 we append two subscripts to each variable from the protocol. This notation has the same meaning that it did for Protocol 1. When we discuss Protocol 2 we will use all of the terminology defined in Subsection 4.1 except that we give replacement definitions for "persistent" and "good." Protocol 2 is similar to, but simpler than Protocol 1. The correctness

```
    Initialization for processor $p$:
        VAL $\leftarrow$ the initial value of processor $p$
    Code for processor $p$ in round $r$:
1.      if group($p$) = active($r$) then LOCAL $\leftarrow$ PICK() else LOCAL $\leftarrow \perp$
2.      broadcast (VAL, LOCAL)
3.      receive ($\text{MSG}_q$, $\text{BIT}_q$) from processor $q$ for $1 \leq q \leq n$
4.      let ANS be the most frequent non-$\perp$ message among the $\text{MSG}_q$
            (break ties arbitrarily)
5.      let NUM be the number of occurrences of ANS
6.      let GLOBAL be the most frequent non-$\perp$ message among the $\text{BIT}_q$ for
            $q \in \{u \mid \text{group}(u) = \text{active}(r/2)\}$   (break ties arbitrarily)
7.      if NUM $\geq n - 2t$ then VAL $\leftarrow$ ANS else VAL $\leftarrow$ GLOBAL
8.      if NUM $\geq n - t$ and have not decided yet then decide VAL
```

Protocol 2: A Variant Randomized Agreement Protocol

proof of Protocol 2 follows the same outline as the correctness proof of Protocol 1. We begin the proof of Protocol 2 with replacement definitions of "persistent" and "good." For all $r \geq 1$ we say that value v is *r-persistent* if

$$|\{p \in C_r \mid \text{VAL}_{r-1,p} = v\}| \geq |C_r| - 2t.$$

For all $r \geq 1$ and for all $v \in \{0, 1\}$, we say that v is r-good if either v is equal to the r-persistent value or if there is no r-persistent value.

Lemma 18: *For any $r \geq 1$, there is at most one r-persistent value and there is at least one r-good value.*

Proof: We show each of the two claims.

Proof that there is at most one r-persistent value: Assume not. Let r be any round in which the claim fails. There are distinct values v and v' and sets

$$S = \{p \in C_r \mid \text{VAL}_{r-1,p} = v\} \quad \text{and} \quad S' = \{p \in C_r \mid \text{VAL}_{r-1,p} = v'\}$$

such that $|S| \geq |C_r| - 2t$ and $|S'| \geq |C_r| - 2t$. It is immediate that $S \cap S' = \emptyset$. Because S and S' are disjoint sets contained in C_r, we have

$$\begin{aligned} |S| + |S'| &\leq |C_r| \\ |C_r| - 2t + |C_r| - 2t &\leq |C_r| \\ |C_r| &\leq 4t. \end{aligned}$$

Thus there are at most $4t$ correct processors during round r. Because the total number of processors is at least $5t+1$ there must be more than t processors that have failed, contradiction.

Proof that there is at least one r-good value: This is immediate from the first part of Lemma 18 and from the definition of r-good value. □

Lemma 19: *If there is some value v and some $r \geq 0$ such that $\text{VAL}_{r,p} = v$ for all correct processors p then any correct processor that has not decided by round r will decide v in round $r+1$.*

Proof: There are at least $n-t$ correct processors that all broadcast v in round $r+1$. In round $r+1$ all correct processors receive at least $n-t$ votes for v and at most t votes for any $v' \neq v$. Therefore, any correct processor that has not previously decided will decide v in round $r+1$. □

Lemma 20: *Let r be an arbitrary round and let v be a value that is not r-persistent. In round r, any correct processor receives fewer than $n-2t$ votes for v.*

Proof: Let $S = \{p \in C_r \mid \text{VAL}_{r-1,p} = v\}$. Because v is not r-persistent we have that $|S| < |C_r| - 2t$. Let q be an arbitrary processor in $C_r - S$. Processor q is correct in round r and has $\text{VAL}_{r-1,q} \neq v$. It therefore does not vote for v. The only processors that vote for v are those in S and those in $N - C_r$. Using the observations that $|S| < |C_r| - 2t$ and $|N - C_r| = n - |C_r|$, it is easy to calculate that $|S| + |N - C_r| < n - 2t$. Consider an arbitrary processor $p \subset C_r$. Processor p receives at most one vote for v from each processor in $S \cup (N - C_r)$. Thus processor p receives fewer than $n-2t$ votes for v. □

Lemma 21: *For all $r \geq 1$, if the result of the round r global random choice is an r-good value then all correct processors decide by round $r+1$.*

Proof: Let r be an arbitrary round. There are two cases.

Case 1: (There is an r-persistent value.) Let v be the r-persistent value, which is unique by Lemma 18. By the definition of b-good, v is the only b-good value. In round r any correct processor receives fewer than $n-2t$ votes for any value $v' \neq v$ (by Lemma 20). Thus in step 7 every correct processor p either sets $\text{VAL}_{r,p}$ to v or to $\text{GLOBAL}_{r,p}$. If the result of the round r global random choice is v, then $\text{VAL}_{r,p} = v$ for all correct processors p and by Lemma 19 all correct processors decide by round $r+1$.

Case 2: (There is no r-persistent value.) In round r any correct processor receives fewer than $n - 2t$ votes for any value v (by Lemma 20). Thus in step 7 an arbitrary correct processor p sets $\text{VAL}_{r,p}$ to $\text{GLOBAL}_{r,p}$. If the result of the round r global random choice is any b-good value $v \neq \perp$, then $\text{VAL}_{r,p} = v$ for all correct processors p and by Lemma 19 all correct processors decide by round $r + 1$. □

Theorem 22: *Protocol 2 solves the randomized agreement problem.*

Proof: We show that the agreement, validity, and termination conditions are satisfied.

Agreement condition: Suppose that round r is the first round in which a correct processor decides and suppose that correct processor p decides v in round r. In round r processor p got at least $n - t$ votes for v. At least $n - 2t$ of these votes are from processors in C_r, so v is the r-persistent value, which is unique by Lemma 18. Therefore, in round r all correct processors get at least $n - 2t$ votes for v and (by Lemma 20) fewer than $n - 2t$ votes for any $v' \neq v$. Thus $\text{VAL}_{r,q} = v$ for any correct processor q. So, any correct processor that decides in round r decides v and by Lemma 19 all of the other correct processors decide v in round $r + 1$.

Validity condition: Suppose all correct processors start the protocol with input v. It is immediate that $\text{VAL}_{0,p} = v$ for all correct processors p. By Lemma 19, all correct processors decide v in round 1.

Termination condition: This is analogous to the proof of termination for Protocol 1. □

In Theorem 22 we proved the correctness of Protocol 2. In Protocol 2 random choices are used in the same way that they are in Protocol 1. This makes the analysis of the expected running time of Protocol 2 identical to the analysis in Subsection 4.3. The analysis is not repeated here.

6. Comparison with Alternative Protocols

We compare our protocol with some alternative protocols (all synchronous). We consider the alternatives in three groups: randomized non-cryptographic protocols, randomized cryptographic protocols, and deterministic protocols.

6.1 Non-Cryptographic Protocols

Ben-Or [2] and Bracha and Toueg [7] investigated non-cryptographic solutions to the randomized agreement problem in the presence of Byzantine processor faults.

Their primary interest was asynchronous protocols, but simple variants of their protocols operate in the synchronous case. Because the topic of this chapter is synchronous protocols, we only consider the synchronous variants of their protocols. In the synchronous case, the protocol due to Bracha and Toueg is similar to the one due to Ben-Or. For brevity, we only compare our protocol with Ben-Or's protocol.

When the redundancy, r, of a system of processors is $\Omega(t)$, Ben-Or's protocol terminates in a constant expected number of rounds. For practical systems, however, it is desirable to operate at a lower redundancy in order to minimize the cost of computer hardware. The optimal value is $r = 3$. Unfortunately, for any r that is $O(1)$, Ben-Or's protocol requires an exponential number of rounds. Compared to Ben-Or's protocol, ours is more efficient for practical amounts of system redundancy.

6.2 Cryptographic Protocols

We consider cryptographic protocols by Rabin [43]; Bracha [6]; and Dwork, Shmoys, and Stockmeyer [24]. These protocols all use cryptographic techniques to conceal information from faulty processors.

Rabin's protocol terminates in a constant expected number of rounds; however, it requires more resources than our protocol. In particular, it requires a trusted dealer that distributes random values before the start of the protocol. The underlying mechanism is authentication and Shamir's [45] shared secret. If the requirements for Rabin's protocol can be met at a reasonable cost, then his protocol is an attractive choice. We believe, however, that in practical systems it is often unrealistic to assume the existence of a trusted dealer.

The randomized protocol due to Bracha terminates in $O(\log n)$ expected rounds. The principal advantage of his protocol is this fast asymptotic performance. The principal disadvantages are the use of cryptographic techniques, a seemingly high constant factor in the run time, relatively high communications costs, and the use of a hard-to-compute local graph partition. (The only known deterministic protocols that compute this partition run in exponential time.)

A new protocol by Dwork, Shmoys, and Stockmeyer terminates in $O(\log \log n)$ expected rounds in the same model as Bracha's protocol. The principal advantage of their protocol is its extremely fast asymptotic performance. The protocol has the same disadvantages as Bracha's protocol, including its use of a hard-to-compute local graph partition.

Another protocol by Dwork, Shmoys, and Stockmeyer terminates in $O(1)$ expected rounds and requires that the redundancy of the system of processors is $\Omega(\log n)$. This protocol does not require any hard-to-compute local graph partition. The principal advantage of this protocol is its efficiency. The principal disadvantages of this protocol are its use of cryptographic techniques and its relatively high redundancy requirement.

6.3 Deterministic Protocols

It is also possible to compare our protocol with deterministic agreement protocols. A representative deterministic protocol is due to Srikanth and Toueg [46]. There are trade-offs between their protocol and ours. The principal advantages of the deterministic protocol are that it uses a fixed number of rounds and that all processors decide at the same round. The principal advantages of our protocol are that the expected number of rounds is small, that the expected number of message bits is small, and that only two rounds are required if the input to all processors is the same. Our protocol cannot ensure simultaneous (in the same round) termination; however, using the techniques of Subsection 5.3, this property can be achieved with high probability.

6 Simultaneity is Harder than Agreement

We investigate the number of rounds of message exchange required for correct processors to act simultaneously (in the same round) in certain synchronous fault-tolerant distributed systems. In particular, we prove a strong lower bound on the number of rounds of message exchange that any randomized protocol requires to solve either the simultaneous agreement problem or the distributed firing squad problem. It is known that any protocol that solves either of these problems and that is resilient to t processor faults has at least one execution that lasts at least $t+1$ rounds. We strengthen that bound by showing that all normal executions of such a protocol last at least $t+1$ rounds. The restriction to normal executions is a technical one that excludes certain executions in which a fortuitous pattern of processor faults enables early termination. The lower bounds proved in this chapter contrast with known protocols that achieve agreement on a value (without simultaneity) in fewer than $t+1$ rounds in some normal executions. Our results are proved for randomized protocols, for a benign failure model (crash faults), and for a weak adversary. They apply *a fortiori* to deterministic protocols, to more malicious failure models, and to stronger adversaries.

1. Introduction

We prove a strong lower bound on the number of rounds of communication

This chapter is joint work with Cynthia Dwork; a preliminary version appeared in the *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems* [13].

that any randomized protocol requires to solve either the simultaneous agreement problem or the distributed firing squad problem. In this chapter we formulate deterministic protocols as a special case of randomized protocols. The lower bound that we prove for randomized protocols holds *a fortiori* for deterministic protocols.

We restrict our attention to problems that are solved in a system of n processors that communicate via a fully connected reliable message system. For some fixed t, a protocol is required to work correctly in any execution in which at most t processors fail, but may produce arbitrary results if more than t processors fail. For the problems that we consider it is known that any protocol that solves the problem has at least one execution that lasts at least $t+1$ rounds. In this chapter we strengthen that bound by showing that all normal executions of such a protocol last at least $t+1$ rounds. An execution of a simultaneous agreement protocol is normal if the number of processors that fail by the end of round r is at most r for all $r \in \{1, \ldots, t\}$. In Subsection 3.2 we give a slightly different definition of a normal execution for the randomized distributed firing squad problem.

Our lower bound applies to normal executions only. We believe that this limitation to normal executions is merely a technical restriction. An adversary can ensure that all executions of a protocol are normal by limiting the rate at which processors fail. Our lower bound implies that a protocol can terminate before round $t+1$ only if a large number of failures occur early in an execution. One would hope that such a pattern of failures would be rare in a reliable fault-tolerant distributed system. Thus, our lower bound removes much of the incentive to design protocols that terminate in fewer than $t+1$ rounds. An important special case of our lower bound is that every failure-free execution of a correct protocol must run for at least $t+1$ rounds.

Our approach is to prove lower bounds for extremely weak variants of the simultaneous agreement problem and the distributed firing squad problem. These are the lazy simultaneous weak agreement problem (defined in Subsection 2.1 of this chapter) and the lazy distributed firing squad problem (defined in Subsection 3.1). These weak problem variants seem interesting only in the context of lower bounds. For example, each can be solved by a protocol that does nothing. The lower bound that we prove for weak variants holds *a fortiori* for stronger variants.

In order to demonstrate a concrete application of the lower bounds that we will prove, we give informal definitions of standard variants of the simultaneous agreement problem and the distributed firing squad problem. We call these variants the randomized simultaneous agreement problem and the randomized distributed firing squad problem. We use the term "randomized" because the termination condition of each problem is probabilistic. The problem definitions that we give suffice for both randomized and deterministic protocols; however, if we know that a protocol is deterministic it is straightforward to replace the termination condition with an equivalent condition that makes no mention of probabilities. For example, the equivalent termination condition for the randomized simultaneous agreement problem is that all correct processors eventually decide.

In the randomized simultaneous agreement problem each processor begins with an input chosen from a fixed set V. The objective is for all of the correct processors to agree on one element of V subject to the following four conditions.

- *Agreement condition:* All correct processors that decide reach the same decision.

- *Validity condition:* If all correct processors start the protocol with input v then v is the decision of all of the correct processors that decide.

- *Simultaneity condition:* If any correct processor decides then all correct processors decide in the same round.

- *Termination condition:* The probability that all correct processors decide by round r tends to 1 as r tends to infinity.

In the randomized distributed firing squad problem each processor may, at any round, receive a request to fire. This request comes from some unspecified source outside of the system of processors. We would like for all of the correct processors to respond to this request by simultaneously firing (*i.e.,* entering a distinguished state). In any protocol that solves this problem the processors must satisfy the following three conditions.

- *Validity condition:* No correct processor fires unless some processor receives a request to fire.

- *Simultaneity condition:* If any correct processor fires then all correct processors fire in the same round.

- *Termination condition:* If any correct processor receives a request to fire then the probability that all correct processors fire by round r tends to 1 as r tends to infinity.

A key assumption must be made if randomization is to be a useful tool for solving consensus problems. This assumption is that there is some measure of independence between the failure modes of the system and the random choices of correct processors. We capture this assumption by imagining that the selection of which processors fail and the behavior of the failed processors are under the control of an adversary that has specified, limited powers. A formal description of our adversary appears in Subsection 2.2. We prove our lower bound for a weak adversary. This contributes to the strength of the bound.

The consensus problems for which we obtain a strong lower bound require simultaneous (in the same round) action by all correct processors. Similar bounds seem to be impossible to obtain for consensus problems without a simultaneity requirement. Consider, for example, the randomized eventual agreement problem, which is just the randomized simultaneous agreement problem without the simultaneity requirement. There are many protocols for this problem that terminate in fewer than $t+1$ rounds in some normal executions, for example, the protocol given in Chapter 5 has normal executions that terminate in 2 rounds for all t. The pattern is as follows: problems that require simultaneous action by the correct processors take at least $t+1$ rounds in all normal executions, but problems that merely require agreement on some value have protocols that terminate faster in some normal executions. This is why we say that simultaneity is harder than agreement.

We use an assortment of techniques to prove the various lower bounds given in this chapter. For the lazy simultaneous weak agreement problem we use a standard technique [15], [20], [21], [22], [26], [37], [38], [39], [33] to give a direct proof that there is no deterministic protocol that beats the bound. We then use a reduction to show that the bound holds for randomized protocols. Specifically, we show that if there is any randomized protocol that beats the bound, then there is a deterministic protocol that does the same. We prove our lower bound for the randomized distributed firing squad problem by reducing the lazy simultaneous weak agreement problem to the lazy distributed firing squad problem. We use a different reduction from the one that Coan, Dolev, Dwork, and Stockmeyer [12] used to show a worst-case lower bound for the distributed firing squad problem. They used a reduction suitable for showing worst-case lower bounds. We use a new reduction suitable for showing a lower bound on all normal executions.

There has been a long history of work on lower bounds on the number of rounds required to solve various consensus problems in various fault models. The earliest lower bounds shown are for worst-case performance. Fischer and Lynch [26] showed that, in the worst case, $t + 1$ rounds are required to solve either the simultaneous or the eventual agreement problem in the Byzantine fault model. The lower bound was extended to the authenticated Byzantine fault model by DeMillo, Lynch, and Merritt [15] and by Dolev and Strong [21] and to the failure-by-omission fault model by Hadzilacos [29]. It was further extended to the crash fault model by Lamport and Fischer [33]. By reducing the agreement problem to the distributed firing squad problem, Coan, Dolev, Dwork, and Stockmeyer [12] showed that in the worst case $t + 1$ rounds are required to solve the distributed firing squad problem. These bounds are tight in the sense that there are matching protocols that terminate in $t + 1$ rounds. Nevertheless, the bounds are weak in that they admit the possibility that there are many executions that terminate faster.

Dolev, Reischuk, and Strong [20] were the first to investigate consensus protocols that sometimes terminate in fewer than $t+1$ rounds. They began by distinguishing between simultaneous agreement and eventual agreement. For both variants of the agreement problem they parameterized the lower bounds by f, the actual number of failures in a given execution of an agreement protocol. They showed that for all $f \in \{0, \ldots, t\}$ and for every protocol for simultaneous agreement there is at least one execution with f failures that lasts for at least $t + 1$ rounds. They further showed that for any $f \in \{0, \ldots, t\}$ and for every protocol for eventual agreement there is at least one execution with f failures that lasts for at least $\min(f + 2, t + 1)$ rounds. Their results were for deterministic protocols only. The Dolev, Reischuk, and Strong lower bound is for the model with authenticated Byzantine failures. Their technique was extended to crash faults by Lamport and Fischer [33].

In recent work Dwork and Moses [22] and Moses and Tuttle [39] have used the theory of knowledge [30] to develop tight bounds on the number of rounds required to solve some consensus problems in all (including non-normal) executions. Each paper restricts its attention to deterministic protocols. For deterministic protocols, each of the papers subsumes the lower bounds of this chapter. The work of Dwork and Moses characterizes the crash fault model. The work of Moses and Tuttle characterizes various failure-by-omission fault models. Devising tight bounds for the Byzantine fault model remains an open question.

2. Lower Bounds for Agreement

In this section we develop a strong lower bound on the number of rounds that a randomized protocol requires to solve the lazy simultaneous weak agreement problem. We begin in Subsection 2.1 by defining this problem. In Subsection 2.2 we give the formal model in which this problem is solved. In Subsection 2.3 we review the known lower bound for deterministic protocols. Finally, in Subsection 2.4 we prove our lower bound for randomized protocols by showing that the existence of a randomized protocol that beats the bound implies the existence of a deterministic protocol that does the same.

2.1 The Problem

The lazy simultaneous weak agreement problem differs from the randomized simultaneous agreement problem only in that the termination condition is deleted (making the problem lazy) and the validity condition is relaxed (making the problem weak). A protocol for this problem is run by a distributed system of n processors, at most t of which may fail. Each processor starts the protocol with an input value v from a fixed set V of legal inputs. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on an element of V as its answer. There are three conditions that the correct processors must satisfy.

- *Agreement condition:* All correct processors that decide reach the same decision.
- *Validity condition:* If all processors are correct and if all processors start the protocol with input v, then v is the decision of all of the processors that decide.
- *Simultaneity condition:* If any correct processor decides then all correct processors decide in the same round.

2.2 The Model

We model an agreement protocol as a synchronous system of automata. Throughout this chapter we let n be the number of processors in the system, we let $N = \{1, \dots, n\}$, and we let $t \leq n-2$ be an upper bound on the number of processor faults that a protocol need tolerate. A protocol $\mathcal{P}$ is described by the following.

- D is the set of possible outcomes from the random choice performed by each processor each round.
- V is the input set.
- Q is the set of processor states.
- $q_0 \in Q$ is the initial state in which each processor begins the protocol.

- M is the set of messages.

- $\mu_{p,q}{:}\mathcal{I}^+ \times V \times D \times Q \rightarrow M$, for $(p,q) \in N^2$, is the message generation function for messages sent from processor p to processor q. ($\mathcal{I}^+$ denotes the set of positive integers.) The first component of the domain of $\mu_{p,q}$ is the current round number. The second is the input to processor p. The third is the current local random choice of processor p. The fourth is the current state of processor p.

- $\delta_p : (M \cup \{\perp\})^n \rightarrow Q$, for $p \in N$, is the state transition function for processor p. We model the absence of a message from some (failed) processor by $\perp$. (The prior state of processor p is omitted from the domain of δ_p because it would be redundant. Processor p can send any required information in a message to itself.)

- $\gamma_p : Q \rightarrow \{\perp\} \cup V$, for $p \in N$, is the decision function for processor p.

Each processor starts an execution of protocol $\mathcal{P}$ in the initial state q_0. The execution consists of a series of rounds. In each round each correct processor makes a local random choice, sends messages to the other processors, receives messages from the other processors, and makes a local state change. We assume *ordered sending:* the i-th message sent by any correct processor in any round is sent to processor i. In any execution of protocol $\mathcal{P}$ a correct processor behaves according to its transition rules during the entire execution. A faulty processor behaves according to its transition rules during some prefix of the execution, then it stops sending messages. The messages sent by a correct processor depend on its current state and on its current local random choice.

Formally, an *execution* of protocol $\mathcal{P}$ is a triple (C, I, H) where C is a function from $N \times \mathcal{I}^+$ to D, where $I \in V^n$, and where H is a function from $N \times \mathcal{I}^+ \times N$ to $\{\perp, \surd\}$. In an execution E we say that C is the *random-choice history,* I is the *input vector,* and H is the *message history.* For all $r \in \mathcal{I}^+$ and for all processors p, the value of $C(p,r)$ is the round r random choice of processor p. I is the vector of inputs to all of the processors. For all $r \in \mathcal{I}^+$ and for all processors p and q, the value of $H(p,r,q)$ determines whether there is a round r message from processor p to processor q. If $H(p,r,q) = \perp$ then processor p sends no round r message to processor q; otherwise, processor p sends the message that is specified by its protocol for its current state and current random choice. If $H(p,r,q) = \surd$ for all $(r,q) \in \mathcal{I}^+ \times N$ then processor p is *correct;* otherwise, processor p is *faulty.*

We impose two constraints on H. First we require that there are at most t faulty processors. Second, we model ordered sending with crash faults by requiring that, for all $(r, r') \in \mathcal{I}^+ \times \mathcal{I}^+$ and for all $(p, q, q') \in N^3$, the following holds: if $H(p, r, q) = \perp$ and $H(p, r', q') = \surd$ then either $r' < r$ or $r' = r$ and $q' < q$. If $H(p, r, n) = \perp$ then we say that processor p *fails by* round r.

An execution $E = (C, I, H)$ is *normal* if for all $r \leq t$ at most r processors fail by round r.

We now give an inductive definition of the round r state of processor p in execution E of protocol $\mathcal{P}$, which we denote state(p, r, E). Assume that $E = (C, I, H)$ where $I = \langle i_1, \ldots, i_n \rangle$. We define state$(p, 0, E) = q_0$ and for all $r \in \mathcal{I}^+$ we define

$$\text{state}(p, r, E) = \begin{cases} \delta_p\langle m_1, \ldots, m_n \rangle & \text{if } H(p, r, n) = \surd; \\ \perp & \text{otherwise,} \end{cases}$$

where

$$m_q = \begin{cases} \mu_{q,p}(r, i_q, C(q, r), \text{state}(q, r-1, E)) & \text{if } H(q, r, p) = \surd; \\ \perp & \text{otherwise.} \end{cases}$$

Correct processor p decides v in round r of execution E if $\gamma_p(\text{state}(p, r, E)) = v$ and $\gamma_p(\text{state}(p, r', E)) = \perp$ for all $r' < r$. The running time of an execution is the number of rounds until the last correct processor decides.

When analyzing the performance of a randomized consensus protocol, it is convenient to assume that the selection of which components fail and the behavior of the failed components are under the control of an adversary. Our lower bound is strong because we prove it for an extremely weak adversary. Specifically, our adversary selects the message history at the start of an execution without ever seeing either the inputs or the random choices. In contrast, a stronger adversary might be allowed to base its actions on the previous behavior (*e.g.,* random choices) in an execution. Because it is weak, our adversary can be modeled as a message history. It should be immediate that for any protocol $\mathcal{P}$, input vector I, random-choice history C and adversary A there is a unique execution $E = (C, I, A)$.

Having defined our adversary, we can now define the expected running time of a fixed protocol $\mathcal{P}$. Let T be a random variable that given an execution of protocol $\mathcal{P}$ returns the running time of the execution. For a fixed adversary A and input vector I, let the expected value of T, taken over the random choices, be denoted $E(T_{A,I})$. Define the expected running time for protocol $\mathcal{P}$ to be $\max_{A,I}(E(T_{A,I}))$.

We model deterministic protocols as a special case of randomized protocols. Specifically, a protocol is deterministic if $|D| = 1$ where D is the set of possible outcomes from the random choice performed by each processor each round. Whenever $|D| = 1$ we adopt the convention that $D = \{0\}$. We use $\mathbf{0}$ to denote the random-choice history that is 0 for all processors and for all rounds.

2.3 The Lower Bound for Deterministic Protocols

We state, without proof, a lower bound for deterministic protocols. The bound is a straightforward extension of a well-known result [15], [20], [21], [22], [26], [37], [38], [39], [33]. Its proof is given in Appendix A of this chapter.

Theorem 1: *Let $\mathcal{P}$ be any deterministic protocol that solves the lazy simultaneous weak agreement problem. In any normal execution of $\mathcal{P}$ no correct processor decides before round $t+1$.*

2.4 The Lower Bound for Randomized Protocols

We prove that, for any protocol $\mathcal{P}$ that solves the lazy simultaneous weak agreement problem and for any normal execution E of protocol $\mathcal{P}$, no correct processor decides before round $t+1$. We do this by showing how to transform an arbitrary protocol for the problem into a deterministic protocol for the same problem. Our transformation preserves the existence of normal executions that terminate in fewer than $t+1$ rounds.

We define a function α that, given an arbitrary protocol for the lazy simultaneous weak agreement problem and an arbitrary random-choice history, produces a deterministic protocol for the lazy simultaneous weak agreement problem. Let protocol $\mathcal{P} = (D, V, Q, q_0, M, \mu, \delta, \gamma)$ be an arbitrary lazy simultaneous weak agreement protocol and let C be an arbitrary random-choice history. We define $\alpha(\mathcal{P}, C)$ to be the protocol $(\{0\}, V, Q, q_0, M, \mu', \delta, \gamma)$ where $\mu'_{p,q}(r, v, \mathbf{0}, s) = \mu_{p,q}(r, v, C(p,r), q)$ for all $(p,q) \in N^2$.

Lemma 2: *Let $\mathcal{P} = (D, V, Q, q_0, M, \mu, \delta, \gamma)$ be any protocol that solves the lazy simultaneous weak agreement problem, let C be an arbitrary random-choice history, and let $\mathcal{P}' = \alpha(\mathcal{P}, C)$. Let $E' = (\mathbf{0}, \langle i_1, \ldots, i_n\rangle, H)$ be any execution of protocol $\mathcal{P}'$. If $E = (C, \langle i_1, \ldots, i_n\rangle, H)$ is an execution of protocol $\mathcal{P}$, then* $\text{state}(p, r, E') = \text{state}(p, r, E)$ *for all $(p, r) \in N \times \{0, 1, \ldots\}$.*

Proof: Suppose that $\mathcal{P}' = (\mathbf{0}, V, Q, q_0, M, \mu', \delta, \gamma)$. The proof is by induction on r.

Basis: $(r = 0)$ It is immediate that $\text{state}(p, 0, E') = q_0 = \text{state}(p, 0, E)$.

Induction: If processor p fails by round r in message history H then it is immediate that $\text{state}(p, r, E') = \perp = \text{state}(p, r, E)$. So, for the remainder of the proof we suppose that processor p does not fail by round r. Thus $H(p, r, n) = \surd$.

For all $(q, s) \in N^2$, let

$$m'_{q,s} = \begin{cases} \mu'_{q,s}(r, i_q, \mathbf{0}(q, r), \text{state}(q, r-1, E')) & \text{if } H(q, r, s) = \surd; \\ \perp & \text{otherwise,} \end{cases}$$

and let

$$m_{q,s} = \begin{cases} \mu_{q,s}(r, i_q, C(q, r), \text{state}(q, r-1, E)) & \text{if } H(q, r, s) = \surd; \\ \perp & \text{otherwise.} \end{cases}$$

We claim that $m'_{q,s} = m_{q,s}$ for all $(q, s) \in N^2$. If $H(q, r, s) = \perp$ then it is immediate that the claim is true. If $H(q, r, s) = \surd$ then we calculate that

$$\begin{aligned} m'_{q,s} &= \mu'_{q,s}(r, i_q, \mathbf{0}(q, r), \text{state}(q, r-1, E')) \\ &= \mu_{q,s}(r, i_q, C(q, r), \text{state}(q, r-1, E)) \\ &= m_{q,s} \end{aligned}$$

Having proved the claim that $m'_{q,s} = m_{q,s}$ for all $(q, s) \in N^2$, we now conclude the induction step by calculating that

$$\begin{aligned} \text{state}(p, r, E') &= \delta_p(m'_{1,p}, \ldots, m'_{n,p}) \\ &= \delta_p(m_{1,p}, \ldots, m_{n,p}) \\ &= \text{state}(p, r, E). \end{aligned}$$ □

Theorem 3: *Let $\mathcal{P}$ be any protocol that solves the lazy simultaneous weak agreement problem and let C be an arbitrary random-choice history. If $\mathcal{P}' = \alpha(\mathcal{P}, C)$ then protocol $\mathcal{P}'$ solves the lazy simultaneous weak agreement problem.*

Proof: The proof is by contradiction. Suppose that protocol $\mathcal{P}'$ does not solve the lazy simultaneous weak agreement problem. Then there is some execution $E = (\mathbf{0}, I, H)$ of protocol $\mathcal{P}'$ for which some correctness condition is violated. By Lemma 2, the same correctness condition is violated in execution (C, I, H) of protocol $\mathcal{P}$. This contradicts the assumption that protocol $\mathcal{P}$ solves the lazy simultaneous weak agreement problem. □

Lemma 4: *Let $\mathcal{P}$ be any protocol that solves the lazy simultaneous weak agreement problem. Let $E = (C, I, H)$ be any normal execution of protocol $\mathcal{P}$ in which the* *correct processors decide in some round r. If $\mathcal{P}' = \alpha(\mathcal{P}, C)$ then $E' = (0, I, H)$ is a normal execution of protocol $\mathcal{P}'$ in which the correct processors decide by round r.*

Proof: By assumption, execution E is normal. Executions E and E' have identical message histories. Therefore, execution E' is normal.

Having shown that execution E' is normal, we now show that all correct processors decide by round r in execution E'. By assumption, the correct processors decide by round r in execution E. By Lemma 2 and the fact that protocols $\mathcal{P}$ and $\mathcal{P}'$ have identical decision functions, correct processors decide in the same round in these two protocols. Therefore, the correct processors decide by round r in execution E' of protocol $\mathcal{P}'$. □

Theorem 5: *Let $\mathcal{P}$ be any protocol that solves the lazy simultaneous weak agreement problem. Let E be any normal execution of protocol $\mathcal{P}$. The earliest round in which the correct processors can decide in execution E is round $t + 1$.*

Proof: The proof is by contradiction. Suppose that the correct processors decide before round $t + 1$ in execution E of protocol $\mathcal{P}$. Suppose $E = (C, I, H)$. Let $\mathcal{P}' = \alpha(\mathcal{P}, C)$. Protocol $\mathcal{P}'$ solves the lazy simultaneous weak agreement problem by Theorem 3. By Lemma 4, there is some normal execution of protocol $\mathcal{P}'$ in which the correct processors decide before round $t + 1$. By Theorem 1, such an execution is impossible, contradiction. □

It is immediate from Theorem 5 that $t + 1$ is a lower bound on the expected number of rounds required to solve the lazy simultaneous weak agreement problem.

3. Lower Bounds for Distributed Firing Squad

We develop a strong lower bound on the number of rounds required to solve the lazy distributed firing squad problem. In Subsection 3.1 we give the correctness conditions for the lazy distributed firing squad problem. In Subsection 3.2 we give the formal model in which the lazy distributed firing squad problem is solved. In Subsection 3.3 we prove our lower bound on the number of rounds required to solve the lazy distributed firing squad problem. Our proof is by reducing the lazy distributed firing squad problem to the lazy simultaneous weak agreement problem.

It is common [8], [12] to assume that processors that solve the distributed firing squad problem have no access to a global clock. We prove our lower bound for a system of processors that have access to a reliable global clock that indicates the current round number. This more powerful model strengthens our lower bound.

3.1 The Problem

A protocol for the lazy distributed firing squad problem is run by a distributed system of n processors, at most t of which may fail. Each processor may receive one or more request to fire during the execution of a protocol. Each correct processor may fire at any point during the execution of the protocol. There are two conditions that the correct processors must satisfy.

- *Validity condition:* No correct processor fires unless some processor receives a request to fire.
- *Simultaneity condition:* If any correct processor fires then all correct processors fire in the same round.

3.2 The Model

We model a distributed firing squad protocol as a synchronous system of automata. We continue to follow the convention that n is the number of processors in the system, $N = \{1, \ldots, n\}$, and $t \leq n - 2$ is an upper bound on the number of processor faults that a protocol need tolerate. A protocol $\mathcal{P}$ is described by the following.

- D is the set of possible outcomes from the random choice performed by each processor each round.
- Q is the set of processor states.
- $q_0 \in Q$ is the initial state in which each processor begins the protocol.
- M is the set of messages.
- $\mu_{p,q} : \mathcal{I}^+ \times \{0,1\} \times D \times Q \rightarrow M$, for $(p,q) \in N^2$, is the message generation function for messages sent from processor p to processor q. The first component of the domain of $\mu_{p,q}$ is the current round number. The second is equal to 1 if processor p receives a request in the current round and 0 otherwise. The third is the current local random choice of processor p. The fourth is the current state of processor p.
- $\delta_p : (M \cup \{\bot\})^n \rightarrow Q$, for $p \in N$, is the state transition function for processor p. We model the absence of a message from some (failed) processor by $\bot$.
- $\gamma_p : Q \rightarrow \{\bot, \text{FIRE}\}$, for $p \in N$, is the decision function for processor p.

Each processor starts an execution of protocol $\mathcal{P}$ in the initial state q_0. The execution consists of a series of rounds. In each round each correct processor makes a local random choice, possibly receives a request to fire, sends messages to the other processors, receives messages from the other processors, and makes a local state change. In any execution of protocol $\mathcal{P}$ a correct processor behaves according to its transition rules during the entire execution. A faulty processor behaves according to its transition rules during some prefix of the execution, then it stops sending messages. The messages sent by a correct processor depend on its current state, on the presence or absence of a request to fire, and on its current random choice.

Formally, an *execution* of protocol $\mathcal{P}$ is a triple (C, W, H) where C is a function from $N \times \mathcal{I}^+$ to D, where W is a function from $N \times \mathcal{I}^+$ to $\{0,1\}$, and where H is a function from $N \times \mathcal{I}^+ \times N$ to $\{\perp, \surd\}$. In an execution E we say that C is the *random-choice history,* W is the *request history,* and H is the *message history.* For all $r \in \mathcal{I}^+$ and for all processors p, the value of $C(p,r)$ is the round r random choice of processor p. For all $r \in \mathcal{I}^+$ and for all processors p, $W(p,r) = 1$ if processor p receives a request to fire in round r and $W(p,r) = 0$ otherwise. Message histories for distributed firing squad protocols are identical to message histories for agreement protocols as described in Subsection 2.2. We use the same terminology and impose the same restrictions.

In execution $E = (C, W, H)$, round $l \geq 0$ is *quiescent* if for all $(p,r) \in N \times \{1, \ldots, l\}$ it is the case that $W(p,r) = 0$ and processor p does not fail by round l; otherwise, round l is *active.* Execution $E = (C, W, H)$ is *l-normal* if there is some quiescent round l such that for all $r \in \{l, \ldots, l+t\}$ at most $r - l$ processors fail by round r. Execution E is *normal* if it is l-normal for some l.

We now give an inductive definition of the round r state of processor p in execution $E = (C, W, H)$ of protocol $\mathcal{P}$, which we denote $\text{state}(p,r,E)$. We define $\text{state}(p,0,E) = q_0$ and for all $r \in \mathcal{I}^+$ we define

$$\text{state}(p,r,E) = \begin{cases} \delta_p\langle m_1, \ldots, m_n\rangle & \text{if } H(p,r,n) = \surd; \\ \perp & \text{otherwise,} \end{cases}$$

where

$$m_q = \begin{cases} \mu_{q,p}(r, W(q,r), C(q,r), \text{state}(q, r-1, E)) & \text{if } H(q,r,p) = \surd; \\ \perp & \text{otherwise.} \end{cases}$$

A correct processor p fires in round r of execution E if $\gamma_p(\text{state}(p,r,E)) = \text{FIRE}$ and $\gamma_p(\text{state}(p,r',E)) = \perp$ for all $r' < r$. We measure the running time of an execution of a randomized distributed firing squad protocol as the number of active rounds that elapse until the last correct processor fires.

Our adversary and our method of calculating the expected cost of a distributed firing squad protocol are the obvious analogues of the adversary and method given for agreement protocols at the end of Subsection 2.2. We model deterministic distributed firing squad protocols in the same way that we modeled deterministic agreement protocols.

3.3 The Lower Bound

In this subsection we prove that all l-normal executions of a lazy distributed firing squad protocol take at least $l + t + 1$ rounds. We do this by reducing the lazy simultaneous weak agreement problem to the lazy distributed firing squad problem. We use a different reduction from the one that Coan, Dolev, Dwork, and Stockmeyer [12] used to show worst-case bounds for the distributed firing squad problem. Using our new reduction, we prove that if there is any l-normal execution of a lazy distributed firing squad protocol in which the correct processors fire before round $l+t+1$ then there is a normal execution of a lazy simultaneous weak agreement protocol in which all correct processors decide before round $t+1$. Because there are no normal executions of a lazy simultaneous weak agreement protocol in which the correct processors decide before round $t + 1$ (Theorem 1), we conclude that there are no l-normal executions of a lazy distributed firing squad protocol in which the correct processors decide before round $l + t + 1$.

We begin with an informal description of our reduction. We use an arbitrary lazy distributed firing squad protocol $\mathcal{P}$ as a basis for constructing a lazy simultaneous weak agreement protocol $\mathcal{P}'$. Protocol $\mathcal{P}'$ has input set $\{1,2\}$. In protocol $\mathcal{P}'$ two copies of protocol $\mathcal{P}$ are run in parallel. One copy corresponds to input 1, and the other corresponds to input 2. In protocol $\mathcal{P}'$ an arbitrary processor p decides in the earliest round in which it would fire in either of the simulated copies of $\mathcal{P}$. If one copy of $\mathcal{P}$ fires first then processor p decides on the value that corresponds to that copy. If both copies fire together then processor p decides 1. There are two components to each random choice made by processor p in protocol $\mathcal{P}'$. One component is used to provide random choices to the two simulated copies of $\mathcal{P}$. The other component is used to provide requests to fire to the simulated copy of $\mathcal{P}$ that corresponds to the input to processor p. Processor p gives no requests to the other simulated copy of $\mathcal{P}$. A formal description of this reduction follows.

We define a function β that, given an arbitrary protocol for the lazy distributed firing squad problem, produces a protocol for the lazy simultaneous weak agreement problem. Let protocol $\mathcal{P} = (D, Q, q_0, M, \mu, \delta, \gamma)$ be an arbitrary lazy distributed firing squad protocol. We define $\beta(\mathcal{P})$ to be the protocol $(D', V', Q', q_0', M', \mu', \delta', \gamma')$, which is given by the following.

- $D' = \{0,1\} \times D$. An element of D' is denoted $[b, c]$ where $b \in \{0,1\}$ and where $c \in D$. The first component of D' is used to simulate requests to protocol $\mathcal{P}$ and the second is used to simulate the random choices made in $\mathcal{P}$.

- $V' = \{1,2\}$.

- $Q' = Q^2$. An element of Q' is denoted $[q_1, q_2]$ where $q_1 \in Q$ and where $q_2 \in Q$. We identify $\perp$, the undefined state, with $[\perp, \perp]$.

- $q_0' = [q_0, q_0]$.

- $M' = M^2$. An element of M' is denoted $[m_1, m_2]$ where $m_1 \in M$ and where $m_2 \in M$. We identify $\perp$, the null message, with $[\perp, \perp]$.

- $\mu'_{p,q}$ is defined as follows:

  $$\mu'_{p,q}(r, v, [b, c], [q_1, q_2]) = [\mu_{p,q}(r, w_1, c, q_1), \mu_{p,q}(r, w_2, c, q_2)],$$

  where $w_i = b$ if $i = v$ and $w_i = 0$ otherwise. The role played by w_i is to provide requests to the one simulated copy of protocol $\mathcal{P}$ that corresponds to the input to processor p and to block requests to the other simulated copy of $\mathcal{P}$.

- δ'_p is defined as follows:

  $$\delta'_p([m_1, m_1'], \ldots, [m_n, m_n']) = [\delta_p(m_1, \ldots, m_n), \delta_p(m_1', \ldots, m_n')].$$

- γ'_p is defined as follows:

  $$\gamma'_p([q_1, q_2]) = \min\{i \mid \gamma_p(q_i) = \text{FIRE}\},$$

  where we follow the convention that $\min \emptyset = \perp$.

We define two functions, first and second. For any lazy distributed firing squad protocol $\mathcal{P}$, these functions yield executions of $\mathcal{P}$ when given an execution of the protocol $\beta(\mathcal{P})$. In Lemma 6 we will use these functions to characterize the relationship between executions of protocol $\mathcal{P}$ and executions of the protocol $\beta(\mathcal{P})$.

Let $\mathcal{P} = (D, Q, q_0, M, \mu, \delta, \gamma)$ be any protocol that solves the lazy distributed firing squad problem, let $\mathcal{P}' = \beta(\mathcal{P})$, and let $E' = (C', \langle i_1, \ldots, i_n \rangle, H)$ be any execution of protocol $\mathcal{P}'$. Recall that C' is a function from $N \times \mathcal{I}^+$ to $\{0,1\} \times D$. Let $B: N \times \mathcal{I}^+ \to \{0,1\}$ and $C: N \times \mathcal{I}^+ \to D$ be chosen so that C' is the cross product of the two functions B and C. Thus $C'(p,r) = [B(p,r), C(p,r)]$ for all $(p,r) \in N \times \mathcal{I}^+$. For all $j \in \{1,2\}$ let W_j be the request history

$$W_j(p,r) = \begin{cases} B(p,r) & \text{if } i_p = j; \\ 0 & \text{otherwise.} \end{cases}$$

We define $\text{first}(E') = (C, W_1, H)$ and $\text{second}(E') = (C, W_2, H)$.

Lemma 6: *Let $\mathcal{P} = (D, Q, q_0, M, \mu, \delta, \gamma)$ be any protocol that solves the lazy distributed firing squad problem, let $\mathcal{P}' = \beta(\mathcal{P})$, and let $E' = (C', \langle i_1, \ldots, i_n \rangle, H)$ be any execution of protocol $\mathcal{P}'$. If $E_1 = \text{first}(E')$ and $E_2 = \text{second}(E')$, then for all $(p,r) \in N \times \{0,1,\ldots\}$*

$$\text{state}(p,r,E') = \begin{cases} [\text{state}(p,r,E_1), \text{state}(p,r,E_2)] & \text{if } r = 0 \text{ or } H(p,r,n) = \surd; \\ \perp & \text{otherwise.} \end{cases}$$

Proof: Suppose that $\mathcal{P}' = (D', V', Q', q_0', M', \mu', \delta', \gamma')$. The proof is by induction on r.

Basis: $(r = 0)$ We calculate that

$$\begin{aligned} \text{state}(p,0,E') &= [q_0, q_0] \\ &= [\text{state}(p,0,E_1), \text{state}(p,0,E_2)]. \end{aligned}$$

Induction: If processor p fails by round r in message history H then it is immediate that $\text{state}(p,r,E') = \perp$. So, for the remainder of the proof we suppose that processor p does not fail by round r. Thus $H(p,r,n) = \surd$.

For all $(q,s,j) \in N^2 \times \{1,2\}$, let

$$m_{q,s} = \begin{cases} \mu'_{q,s}(r, i_q, C'(q,r), \text{state}(q, r-1, E')) & \text{if } H(q,r,s) = \surd; \\ \perp & \text{otherwise,} \end{cases}$$

and let

$$m^j_{q,s} = \begin{cases} \mu_{q,s}(r, W_j(q,r), C(q,r), \text{state}(q, r-1, E_j)) & \text{if } H(q,r,s) = \surd; \\ \perp & \text{otherwise.} \end{cases}$$

We claim that $m_{q,s} = [m^1_{q,s}, m^2_{q,s}]$ for all $(q,s) \in N^2$. If $H(q,r,s) = \perp$ then it is immediate that the claim is true. If $H(q,r,s) = \surd$ then we calculate that

$$\begin{aligned} m_{q,s} &= \mu'_{q,s}(r, i_q, C'(q,r), \text{state}(q, r-1, E')) \\ &= \mu'_{q,s}(r, i_q, [B(q,r), C(q,r)], [\text{state}(q, r-1, E_1), \text{state}(q, r-1, E_2)]) \\ &= [\mu_{q,s}(r, W_1(q,r), C(q,r), \text{state}(q, r-1, E_1)), \\ &\qquad \mu_{q,s}(r, W_2(q,r), C(q,r), \text{state}(q, r-1, E_2))] \\ &= [m^1_{q,s}, m^2_{q,s}] \end{aligned}$$

Having proved the claim that $m_{q,s} = [m^1_{q,s}, m^2_{q,s}]$ for all $(q,s) \in N^2$, we now conclude the induction step by calculating that

$$\begin{aligned} \text{state}(p,r,E') &= \delta'_p(m_{1,p}, \ldots, m_{n,p}) \\ &= \delta'_p([m^1_{1,p}, m^2_{1,p}], \ldots, [m^1_{n,p}, m^2_{n,p}]) \\ &= [\delta_p(m^1_{1,p}, \ldots, m^1_{n,p}), \delta_p(m^2_{1,p}, \ldots, m^2_{n,p})] \\ &= [\text{state}(p,r,E_1), \text{state}(p,r,E_2)]. \end{aligned}$$ □

Theorem 7: *Let $\mathcal{P} = (D, Q, q_0, M, \mu, \delta, \gamma)$ be any protocol that solves the lazy distributed firing squad problem. If $\mathcal{P}' = \beta(\mathcal{P})$, then the protocol $\mathcal{P}'$ solves the lazy simultaneous weak agreement problem.*

Proof: Suppose that $\mathcal{P}' = (D', V', Q', q'_0, M', \mu', \delta', \gamma')$ We show that the agreement, simultaneity, and validity conditions are satisfied.

Agreement and simultaneity conditions: Say that correct processor p decides v in round r of execution E' of protocol $\mathcal{P}'$. Let $E_1 = \text{first}(E')$ and let $E_2 = \text{second}(E')$. By Lemma 6, $\text{state}(q,r,E') = [\text{state}(q,r,E_1), \text{state}(q,r,E_2)]$ for all correct processors q. It is immediate from the definition of decides that $\gamma'_p(\text{state}(p,r,E')) = v$ and that $\gamma'_p(\text{state}(p,r',E')) = \perp$ for all $r' \in \{1, \ldots, r-1\}$. Using the definition of γ', we observe that

$$\gamma_p(\text{state}(p,r,E_v)) = \text{FIRE},$$
$$\gamma_p(\text{state}(p,r',E_j)) = \perp \text{ for all } (r',j) \in \{1, \ldots, r-1\} \times \{1,2\},$$

and

$$\text{if } v = 2 \text{ then } \gamma_p(\text{state}(p,r,E_1)) = \perp.$$

By the simultaneity condition satisfied by protocol $\mathcal{P}$ we have, for all correct processors q

$$\gamma_q(\text{state}(q,r,E_v)) = \text{FIRE},$$
$$\gamma_q(\text{state}(q,r',E_j)) = \perp \text{ for all } (r',j) \in \{1, \ldots, r-1\} \times \{1,2\},$$

and

$$\text{if } v = 2 \text{ then } \gamma_q(\text{state}(q, r, E_1)) = \perp.$$

Let q be any processor that is correct in execution E'. Using the definition of γ', we have that $\gamma'_q(\text{state}(q, r, E')) = v$ and that $\gamma'_q(\text{state}(q, r', E')) = \perp$ for all $r' \in \{1, \ldots, r-1\}$. Thus any correct processor decides v in round r of execution E'.

Validity condition: Say that all processors are correct and that all processors start execution E' of protocol $\mathcal{P}'$ with input v. There are two cases. Either $v = 1$ or $v = 2$. We argue the case when $v = 1$. The other case is similar. Let $E_1 = \text{first}(E')$ and let $E_2 = \text{second}(E')$. By Lemma 6, $\text{state}(p, r, E') = [\text{state}(p, r, E_1), \text{state}(p, r, E_2)]$ for all $p \in N$. By Lemma 6, no processor ever receives a request to fire in execution E_2. Thus $\gamma_p(\text{state}(p, r, E_2)) = \perp$ for all $(p, r) \in N \times \mathcal{I}^+$. From the definition of γ' we have that $\gamma'_p(\text{state}(p, r, E')) \in \{\perp, 1\}$ for all $(p, r) \in N \times \mathcal{I}^+$. Thus the decision of all of the processors that decide is 1. □

Lemma 8: *Let $\mathcal{P}$ be any protocol that solves the lazy distributed firing squad problem. Let $E = (C, W, H)$ be any normal execution of protocol $\mathcal{P}$ in which the correct processors fire in some round r. If $\mathcal{P}' = \beta(\mathcal{P})$ then there is some normal execution E' of protocol $\mathcal{P}'$ in which the correct processors decide by round r.*

Proof: This proof is in two parts. First we construct the execution E'. Second we show that it has the desired properties.

Construction of the execution E': We specify that $E' = (C', \langle 1, 1, \ldots, 1\rangle, H)$ where the random-choice history C' is defined to be $C'(p, r') = [W(p, r'), C(p, r')]$ for all $(p, r') \in N \times \mathcal{I}^+$.

Verification that the execution E' has the desired properties: We now show that execution E' is normal and that all correct processors decide by round r in execution E'.

Executions E and E' have the same message histories. Execution E is a normal execution of a lazy distributed firing squad protocol. The following property follows from the definitions of normal executions of lazy distributed firing squad and randomized simultaneous agreement protocols: if H' is the message history of any normal execution of a lazy distributed firing squad protocol then any execution of a randomized simultaneous agreement protocol with message history H' is normal. Thus execution E' is normal.

Having shown that execution E' is normal, we now show that all correct processors decide by round r in execution E'. For all $p \in N$ let γ_p be the decision function used by processor p in protocol $\mathcal{P}$ and let γ'_p be the decision functions used by processor p in protocol $\mathcal{P}'$. Note that $E = \text{first}(E')$. Let $F = \text{second}(E')$. By Lemma 6,

$$\text{state}(p, r, E') = [\text{state}(p, r, E), \text{state}(p, r, F)].$$

By assumption, the correct processors fire in round r in execution E. Thus, $\gamma_p(\text{state}(p, r, E)) = \text{FIRE}$. We have that $\gamma'_p([\text{state}(p, r, E), \text{state}(p, r, F)]) = 1$, by the definition of γ'. Therefore, the correct processors decide by round r in execution E' of protocol $\mathcal{P}'$. □

Lemma 9: *If there is a protocol $\mathcal{P}$ that solves the lazy distributed firing squad problem and that has an l-normal execution E in which all correct processors fire in round r, then there is a protocol $\mathcal{P}'$ that solves the lazy distributed firing squad problem and that has a 0-normal execution E' in which all correct processors fire in round $r - l$.*

Proof: The protocol $\mathcal{P}'$ is only slightly different from the protocol $\mathcal{P}$; it is constructed from $\mathcal{P}$ by having each processor p start in the state state(p, l, E) rather than in the state q_0. (We overcome the technical obstacle that our model requires that all n processors start in the same state by encoding the new start states in the message generation functions of protocol $\mathcal{P}'$.) For all r, in protocol $\mathcal{P}'$ each processor sends the messages that it would send in round $r + l$ of protocol $\mathcal{P}$. That is, if protocol $\mathcal{P}$ uses the message generation function $\mu_{p,q}(r, w, c, s)$, then protocol $\mathcal{P}'$ uses the message generation function $\mu'_{p,q}(r, w, c, s) = \mu_{p,q}(r + l, w, c, s)$. No other change is required.

The proof that protocol $\mathcal{P}'$ solves the lazy distributed firing squad problem is straightforward and is omitted. The execution E' is constructed from the execution E simply by discarding the first l rounds of the random-choice history, the request history, and the message history. Removing l quiescent rounds from the start of an l-normal execution in this way produces a 0-normal execution. Thus execution E' is 0-normal. It is straightforward to show by induction on r that $\text{state}(p, r', E') = \text{state}(p, l + r', E)$ for all $(p, r') \in N \times \mathcal{I}^+$. No processors fire in quiescent rounds of execution E by the validity condition satisfied by protocol $\mathcal{P}$. Thus, if the correct processors fire in round r in execution E then they fire in round $r - l$ in execution E'. □

Theorem 10: *Let $\mathcal{P}$ be any protocol that solves the lazy distributed firing squad problem. If E is any l-normal execution of protocol $\mathcal{P}$, then the earliest round in which the correct processors can fire in execution E is round $l + t + 1$.*

Proof: The proof is by contradiction. Suppose that the correct processors fire before round $l + t + 1$ in execution E of protocol $\mathcal{P}$. By Lemma 9, there is a protocol $\mathcal{P}'$ that solves the lazy distributed firing squad problem and that has a 0-normal execution E' in which the correct processors fire before round $t + 1$.

Let $\mathcal{P}'' = \beta(\mathcal{P})$. Protocol $\mathcal{P}''$ solves the lazy simultaneous weak agreement problem by Theorem 7. By Lemma 8, there is some normal execution of protocol $\mathcal{P}''$ in which the correct processors decide before round $t + 1$. By Theorem 5, such an execution is impossible, contradiction. □

It is immediate from Theorem 10 that $t + 1$ is a lower bound on the expected number of rounds required to solve the lazy distributed firing squad problem.

4. Tabulation of Known Bounds

We tabulate the known upper and lower bounds on rounds for various consensus problems. The variables used in the tables are n, the number of processors; t, a bound on the number of faults; and f, the number of faults that actually occur. We consider the distributed firing squad problem, the simultaneous agreement problem, and the eventual agreement problem. For each problem we consider a worst-case bound, a worst-case bound parameterized by f, and a bound on the expected number of rounds required by a randomized protocol. The lower bounds are in the crash fault model; the upper bounds are in the Byzantine fault model. For the agreement problems, the lower bounds assume weak agreement and the upper bounds assume strong agreement.

Lower bounds are given in Table 1. These bounds are for the crash fault model

	Distributed Firing Squad Problem	Simultaneous Agreement Problem	Eventual Agreement Problem
Worst-case rounds (deterministic)	$t+1$ **Coan, Dolev, Dwork, and Stockmeyer [12]**	$t+1$ **Lamport and Fischer [33]**	$t+1$ **Lamport and Fischer [33]**
Worst-case rounds parameterized by f (deterministic)	$t+1$ **New in this chapter.**	$t+1$ **Dolev, Reischuk, and Strong [20]**	$\min(f+2, t+1)$ **Dolev, Reischuk, and Strong [20]**
Expected rounds (randomized)	$t+1$ **New in this chapter.**	$t+1$ **New in this chapter.**	— **No non-trivial bound is known.**

Table 1: Lower Bounds for Crash Faults

and therefore also hold for more malicious failure models. Each entry in the table gives a bound and a reference to the paper where the bound first appeared.

Upper bounds (protocols) are given in Table 2 for comparison with the lower bounds. The upper bounds are all for the unauthenticated Byzantine fault model and therefore also work in more benign fault models. In all but two cases, the bounds are tight. In the case of early-stopping eventual agreement protocols, Dolev, Reischuk, and Strong [20] have a protocol that achieves the lower bound if the number of processors is large ($n \geq 2t^2+3t+5$). The tabulated protocol works for all $n \geq 3t+1$. In the case of randomized eventual agreement protocols, there are many protocols that improve on the tabulated one for more benign fault models. Protocols exist that achieve $O(1)$ rounds if n is $\Omega(t^2)$ [3], if only crash faults occur [10], or if there is a trusted dealer [43]. There is a protocol due to Bracha [6] that uses cryptography to terminate in $O(\log n)$ rounds. The performance of this protocol was improved to $O(\log\log n)$ rounds by Dwork, Shmoys, and Stockmeyer [24]. Another protocol by Dwork, Shmoys, and Stockmeyer uses cryptography to terminate in $O(1)$ rounds if n is $\Omega(t \cdot \log t)$.

	Distributed Firing Squad Problem	Simultaneous Agreement Problem	Eventual Agreement Problem
Worst-case rounds (deterministic)	$t+1$ **Burns and Lynch [8]**	$t+1$ **Lamport, Shostak, and Pease [34]**	$t+1$ **Lamport, Shostak, and Pease [34]**
Worst-case rounds parameterized by f (deterministic)	$t+1$ **Burns and Lynch [8]**	$t+1$ **Lamport, Shostak, and Pease [34]**	$\min(2f+5, 2t+3)$ **Dolev, Reischuk, and Strong [20]**
Expected rounds (randomized)	$t+1$ **Burns and Lynch [8]**	$t+1$ **Lamport, Shostak, and Pease [34]**	$O(t/\log n)$ **New in Chapter 5.**

Table 2: Upper Bounds for Unauthenticated Byzantine Faults

The protocol of Lamport, Shostak, and Pease [34] is a deterministic simultaneous agreement protocol that always terminates in $t+1$ rounds. It is therefore also a degenerate randomized protocol (in which processors ignore their random choices) that terminates in an optimal expected number of rounds. Each entry in the table gives a bound and a reference to the paper where the bound first appeared.

Appendix A to Chapter 6: Lower Bounds for Deterministic Agreement Protocols

We prove Theorem 1, which is a lower bound on the number of rounds required by any deterministic protocol for the lazy simultaneous weak agreement problem. All executions and protocols in this appendix are deterministic. Our proof is in the style developed by Merritt [38] as simplified by Dwork and Moses [22].

A.1 Directly Similar Executions and Similar Executions

We define two useful relations on executions—direct similarity and similarity. Recall that t is an upper bound on the number of processor failures. Let E and E' be executions of some protocol. E is *directly similar* to E', written $E \sim E'$, if there is some processor p such that $\text{state}(p,t,E) = \text{state}(p,t,E')$, and p is correct in E and E'. Note the special role played by round t: two executions are directly similar if they are indistinguishable to some correct processor at round t. The relation $\sim$ is symmetric. The relation *similar*, written $\approx$, is taken to be the reflexive, transitive closure of the relation $\sim$. Thus, $\approx$ is an equivalence relation.

We now prove three lemmas that establish some basic properties of the $\sim$ relation. In Lemma A–1 we prove that two executions are directly similar if they differ only in the sending of a single round t message (*i.e.,* the message is sent in one execution and is not sent in the other).

Lemma A–1: *Let $E = (\mathbf{0}, I, H)$ and $E' = (\mathbf{0}, I, H')$ be arbitrary executions. Let p and q be arbitrary processors. If the message histories H and H' are identical except that $H(p,t,q) = \surd$ and $H'(p,t,q) = \perp$, then $E \sim E'$.*

Proof: We claim that $\text{state}(s,r,E) = \text{state}(s,r,E')$ for all $(s,r) \in N \times \{0,\ldots,t\} - \{(p,t),(q,t)\}$. The proof of the claim is a straightforward induction on r and is omitted.

Processor p is faulty in execution E'. In every execution there are at least two correct processors because $n \geq t+2$. So, there must be some processor $u \neq q$ that is correct in execution E'. Any processor that is correct in execution E' is also correct in execution E. Thus processor u is correct in executions E and E'. By the claim, $\text{state}(u,t,E) = \text{state}(u,t,E')$. Thus $E \sim E'$. □

In Lemma A–2 we prove that two executions are directly similar if, for some r, they differ only in the sending of a single round r message to some processor that fails (in both executions) before sending any round $r+1$ messages.

Lemma A–2: *Let $E = (0, I, H)$ and $E' = (0, I, H')$ be arbitrary executions. Let $r \in \mathcal{I}^+$. Let p and q be arbitrary processors. If $H(q, r+1, 1) = \perp$ and if the message histories H and H' are identical except that $H(p, r, q) = \surd$ and $H'(p, r, q) = \perp$, then $E \sim E'$.*

Proof: We claim that $\text{state}(s, r', E) = \text{state}(s, r', E')$ for all $(s, r') \in N \times \{0, \ldots\} - \{(p, r), (q, r)\}$. The proof of the claim is a straightforward induction on r' and is omitted.

Processors p and q are faulty in execution E'. In every execution there is at least one correct processor because $n \geq t + 2$. So there must be some processor u that is correct in execution E'. Any processor that is correct in execution E' is also correct in execution E. Thus processor u is correct in executions E and E'. By the claim, $\text{state}(u, t, E) = \text{state}(u, t, E')$. Thus $E \sim E'$. □

In Lemma A–3 we prove that two executions are directly similar if they differ only in the input to some processor that fails before sending its first message.

Lemma A–3: *Let $E = (0, I, H)$ and $E' = (0, I', H)$ be arbitrary executions where $I = \langle i_1, \ldots, i_n \rangle$ and where $I' = \langle i'_1, \ldots, i'_n \rangle$. Let p be an arbitrary processor. If $i_q = i'_q$ for all $q \in N - \{p\}$ and if $H(p, 1, 1) = \perp$, then $E \sim E'$.*

Proof: We claim that $\text{state}(s, r, E) = \text{state}(s, r, E')$ for all $(s, r) \in N \times \{0, \ldots\} - (p, 0)$. The proof of the claim is a straightforward induction on r and is omitted.

Clearly, executions E and E' have the same set of faulty processors. In every execution there is at least one correct processor because $n \geq t + 2$. So, there must be some processor u that is correct in executions E and E'. By the claim, $\text{state}(u, t, E) = \text{state}(u, t, E')$. Thus $E \sim E'$. □

A.2 The Lower Bound

In this subsection we give a series of lemmas that culminate in our lower bound for agreement. Lemma A–4 is our key lemma in which we prove that two normal executions are similar if they differ only in the sending of a single round $r \in \{1, \ldots, t\}$ message (*i.e.*, the message is sent in one execution and not in the other) and if no processor ever fails after sending its last round r message.

Lemma A–4: *Let $E = (0, I, H)$ and $E' = (0, I, H')$ be arbitrary normal executions. Let $r \in \{1, \ldots, t\}$. Let p and q be arbitrary processors. If every processor* *that fails in message history H fails by round r and if the message histories H and H' are identical except that $H(p,r,q) = \surd$ and $H'(p,r,q) = \perp$, then $E \approx E'$.*

Proof: The proof is by reverse induction on r, from $r = t$ to $r = 1$.

Basis: $(r = t)$ By Lemma A–1, $E \sim E'$. Thus $E \approx E'$.

Induction: We make the following definitions. Let

$$J(s,r',u) = \begin{cases} \perp & \text{if } r' \geq r+1 \text{ and } s = q; \\ H(s,r',u) & \text{otherwise.} \end{cases}$$

The message history J is identical to the message history H except that in message history J processor q fails after sending its last round r message. Let

$$J'(s,r',u) = \begin{cases} \perp & \text{if } r' \geq r+1 \text{ and } s = q; \\ H'(s,r',u) & \text{otherwise.} \end{cases}$$

The message history J' is identical to the message history H' except that in message history J' processor q fails after sending its last round r message. Let $F = (\mathbf{0}, I, J)$ and let $F' = (\mathbf{0}, I, J')$. Observe that F and F' are normal executions. Observe that every processor that fails in message history J fails by round $r+1$ and every processor that fails in message history J' fails by round $r+1$. We now calculate

$$\begin{aligned} E &\approx F && \text{By repeated application of the induction hypothesis.} \\ &\sim F' && \text{By Lemma A–2.} \\ &\approx E' && \text{By repeated application of the induction hypothesis.} \end{aligned}$$ □

In Lemma A–5 we use Lemmas A–3 and A–4 to show that two failure-free normal executions are similar if they differ only in the input to one processor.

Lemma A–5: *Let $E = (\mathbf{0}, I, H)$ and $E' = (\mathbf{0}, I', H)$ be arbitrary normal executions where $I = \langle i_1, \ldots, i_n \rangle$ and where $I' = \langle i'_1, \ldots, i'_n \rangle$. Let p be an arbitrary processor. If $i_q = i'_q$ for all $q \in N - \{p\}$ and if all processors are correct in the message history H, then $E \approx E'$.*

Proof: We make the following definition. Let

$$J(s,r',u) = \begin{cases} \perp & \text{if } s = p; \\ \surd & \text{otherwise.} \end{cases}$$

Thus, the message history J is identical to the message history H except that in message history J processor p sends no messages. Let $F = (0, I, J)$ and let $F' = (0, I', J)$. We calculate that

$$\begin{aligned} E &\approx F && \text{By repeated application of Lemma A–4.} \\ &\sim F' && \text{By Lemma A–3.} \\ &\approx E' && \text{By repeated application of Lemma A–4.} \end{aligned}$$

□

In Lemma A–6 we use Lemmas A–4 and A–5 to prove the surprising result that all normal executions are similar.

Lemma A–6: *For all normal executions $E = (0, I, H)$ and $E' = (0, I', H')$ it is the case that $E \approx E'$.*

Proof: Let $J(p, r, q) = \surd$ for all $r \in \mathcal{I}^+$ and for all $(p, q) \in N^2$. Let $F = (0, I, J)$ and let $F' = (0, I', J)$. We calculate that

$$\begin{aligned} E &\approx F && \text{By repeated application of Lemma A–4.} \\ &\approx F' && \text{By repeated application of Lemma A–5.} \\ &\approx E' && \text{By repeated application of Lemma A–4.} \end{aligned}$$

□

Theorem 1: *Let $\mathcal{P}$ be any deterministic protocol that solves the lazy simultaneous weak agreement problem. In any normal execution of $\mathcal{P}$ no correct processor decides before round $t + 1$.*

Proof: The proof is by contradiction. Suppose there is a normal execution $E = (0, I, H)$ of protocol $\mathcal{P}$ in which some correct processor decides in some round r where $r \leq t$. By the simultaneity condition satisfied by $\mathcal{P}$, all correct processors decide in round r. By the agreement condition satisfied by $\mathcal{P}$, all correct processors reach the same decision. Without loss of generality suppose they all decide 0.

Let $H'(s, r', u) = \surd$ for all $r' \in \mathcal{I}^+$ and for all $(s, u) \in N^2$. Let $E' = (0, I', H')$ where I' is an n-element vector of ones. By the validity condition satisfied by protocol $\mathcal{P}$, no correct processor decides 0 in execution E'. By Lemma A–6, $E \approx E'$. Thus there is a chain of executions $E = E_1 \sim E_2 \sim \cdots \sim E_j = E'$. We have already shown that all of the correct processors in execution $E = E_1$ decide 0 in round r. For all $i \in \{2, \ldots, j\}$, it follows from the definition of $\sim$ and from the agreement and simultaneity conditions satisfied by protocol $\mathcal{P}$ that all of the correct processors in execution E_i decide 0 in round r. Thus all of the correct processors in E' decide 0 in round r, contradiction. □

References

1. W. Alexi, B. Chor, O. Goldreich, and C. Schnorr, "RSA/Rabin Bits are $\frac{1}{2} + \frac{1}{\text{poly}(\log N)}$ Secure," *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science,* pp. 449–457, October 1984.

2. M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing,* pp. 27–30, August 1983.

3. M. Ben-Or, "Fast Asynchronous Byzantine Agreement," *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing,* pp. 149–151, August 1985.

4. M. Blum and S. Micali, "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits," *SIAM Journal on Computing,* Vol. 13, No. 4, pp. 850–864, November 1984.

5. G. Bracha, "An Asynchronous $\lfloor (n-1)/3 \rfloor$-Resilient Consensus Protocol," *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* pp. 154–162, August 1984.

6. G. Bracha, "An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Algorithm," *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing,* pp. 316–326, May 1985.

7. G. Bracha and S. Toueg, "Resilient Consensus Protocols," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing,* pp. 12–26, August 1983.

8. J. Burns and N. Lynch, "The Byzantine Firing Squad Problem," *Advances in Computing Research: Parallel and Distributed Computing,* Vol. 4, JAI Press Inc., Greenwich, Connecticut, to appear. Available as Technical Report MIT/LCS/TM–275, Laboratory for Computer Science, MIT, April 1985.

9. B. Chor and B. Coan, "A Simple and Efficient Randomized Byzantine Agreement Algorithm," *IEEE Transactions on Software Engineering,* Vol. SE–11, No. 6, pp. 531–539, June 1985.

10. B. Chor, M. Merritt, and D. Shmoys, "Simple Constant-Time Consensus Protocols in Realistic Failure Models," *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing,* pp. 152–162, August 1985.

11. B. Coan, "A Communication-Efficient Canonical Form for Fault-Tolerant Distributed Protocols," *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing,* pp. 63–72, August 1986.

12. B. Coan, D. Dolev, C. Dwork, and L. Stockmeyer, "The Distributed Firing Squad Problem," *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing,* pp. 335–345, May 1985.

13. B. Coan and C. Dwork, "Simultaneity is Harder than Agreement," *Proceedings of the Fifth IEEE Symposium on Reliability in Distributed Software and Database Systems,* pp. 141–150, January 1986.

14. B. Coan and J. Lundelius, "Transaction Commit in a Realistic Fault Model," *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing,* pp. 40–51, August 1986.

15. R. DeMillo, N. Lynch, and M. Merritt, "Cryptographic Protocols," *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing,* pp. 383–400, May 1982.

16. D. Dolev, "The Byzantine Generals Strike Again," *Journal of Algorithms,* Vol. 3, No. 1, pp. 14–30, March 1982.

17. D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *Journal of the ACM,* Vol. 34, No. 1, pp. 77–97, January 1987.

18. D. Dolev, M. Fischer, R. Fowler, N. Lynch, and H. R. Strong, "An Efficient Algorithm for Byzantine Agreement without Authentication," *Information and Control,* Vol. 52, Nos. 1–3, pp. 257–274, January–March 1982.

19. D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl, "Reaching Approximate Agreement in the Presence of Faults," *Journal of the ACM,* Vol. 33, No. 3, pp. 499–516, July 1986.

20. D. Dolev, R. Reischuk, and H. R. Strong, "Eventual is Earlier than Immediate," *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science,* pp. 196–203, November 1982.

21. D. Dolev and H. R. Strong, "Polynomial Algorithms for Multiple Processor Agreement," *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing,* pp. 401–407, May 1982.

22. C. Dwork and Y. Moses, "Knowledge and Common Knowledge in a Byzantine Environment I: Crash Failures," *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge,* pp. 149–170, March 1986.

23. C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* pp. 103–118, August 1984.

24. C. Dwork, D. Shmoys, and L. Stockmeyer, "Flipping Persuasively in Constant Expected Time," *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science,* pp. 222–232, October 1986.

25. A. Fekete, "Asymptotically Optimal Algorithms for Approximate Agreement," *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing,* pp. 73–87, August 1986.

26. M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency," *Information Processing Letters,* Vol. 14, No. 4, pp. 183–186, June 1982.

27. M. Fischer, N. Lynch, and M. Merritt, "Easy Impossibility Proofs for Distributed Consensus Problems," *Distributed Computing,* Vol. 1, No. 1, pp. 26–39, January 1986.

28. M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM,* Vol. 32, No. 2, pp. 374–382, April 1985.

29. V. Hadzilacos, *Issues of Fault Tolerance in Concurrent Computations,* Ph. D. Thesis, Harvard University, June 1984. Available as Technical Report TR–11–84, Department of Computer Science, Harvard University, June 1984.

30. J. Halpern and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* pp. 50–61, August 1984. Revised as of January 1986 as Technical Report IBM–RJ–4421, IBM Corporation.

31. D. Knuth, "Big Omicron and Big Omega and Big Theta," *SIGACT News,* Vol. 8, No. 2, pp. 18–24, April–June 1976

32. L. Lamport, "The Weak Byzantine Generals Problem," *Journal of the ACM,* Vol. 30, No. 3, pp. 668–676, July 1983.

33. L. Lamport and M. Fischer, "Byzantine Generals and Transaction Commitment Protocols," manuscript, 1982.

34. L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 3, pp. 382–401, July 1982.

35. N. Lynch, M. Fischer, and R. Fowler, "A Simple and Efficient Byzantine Generals Algorithm," *Proceedings of the Second IEEE Symposium on Reliability in Distributed Software and Database Systems,* pp. 46–52, July 1982.

36. S. Mananey and F. Schneider, "Inexact Agreement: Accuracy, Precision, and Graceful Degredation," *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing,* pp. 237–249, August 1985.

37. M. Merritt, *Cryptographic Protocols,* Ph. D. Thesis, Georgia Institute of Technology, February 1983.

38. M. Merritt, private communication, November 1984.

39. Y. Moses and M. Tuttle, "Programming Simultaneous Actions Using Common Knowledge," *Algorithmica,* to appear. Preliminary version available in *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science,* pp. 208–221, October 1986.

40. M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM,* Vol. 27, No. 2, pp. 228–234, April 1980.

41. K. Perry, *Early Stopping Protocols for Fault-Tolerant Distributed Agreement,* Ph. D. Thesis, Cornell University, January 1985. Available as Technical Report TR–85–662, Department of Computer Science, Cornell University, February 1985.

42. J. Plumstead, "Inferring a Sequence Generated by a Linear Congruence," *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science,* pp. 153–159, November 1982.

43. M. Rabin, "Randomized Byzantine Generals," *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science,* pp. 403–409, November 1983.

44. R. Reischuk, "A New Solution for the Byzantine Generals Problem," *Information and Control,* Vol. 64, Nos. 1–3, pp. 23–42, January–March 1985.

45. A. Shamir, "How to Share a Secret," *Communications of the ACM,* Vol. 22, No. 11, pp. 612–613, November 1979.

46. T. Srikanth and S. Toueg, "Byzantine Agreement Made Simple: Simulating Authentication without Signatures," *Distributed Computing,* to appear. Available as Technical Report 84–623, Department of Computer Science, Cornell University, July 1984.

47. R. Turpin and B. Coan, "Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement," *Information Processing Letters,* Vol. 18, No. 2, pp. 73–76, February 1984.

48. J. Wensley, *et al.,* "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE,* Vol. 66, No. 10, pp. 1240–1255, October 1978.

Technical Biography of the Author

Brian A. Coan was born in Manhattan where he attended Trinity School, the oldest continually operating secondary school in the United States. He graduated in June 1973 and enrolled at Princeton University. In June 1977 he received the B. S. E. degree with high honors in Electrical Engineering and Computer Science from Princeton University. He then worked for one year for Amdahl Corporation designing a special-purpose compiler to be used as part of a new computer-aided design system. In August 1978 he joined Bell Telephone Laboratories, now known as AT&T Bell Laboratories, where he did cost-benefit studies for a proposed office automation system. In June 1979, while employed by AT&T Bell Laboratories, he received the M. S. degree in Computer Engineering from the Department of Electrical Engineering at Stanford University. In February 1982 he entered the doctoral program in Computer Science at the Massachusetts Institute of Technology. He anticipates receiving the Ph. D. degree in Computer Science in June 1987. Since January 1987, he has been a Member of the Technical Staff at Bell Communications Research.