Designing Fault-Tolerant Algorithms for Distributed Systems Using Communication Primitives

T K Srikanth
(Ph. D. Thesis)
TR 86-739
February 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

DESIGNING FAULT-TOLERANT ALGORITHMS FOR DISTRIBUTED SYSTEMS USING COMMUNICATION PRIMITIVES

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
T K Srikanth
January 1986

DESIGNING FAULT-TOLERANT ALGORITHMS FOR DISTRIBUTED SYSTEMS USING COMMUNICATION PRIMITIVES

T K Srikanth, Ph.D.
Cornell University 1986

Fault-tolerance is an important requirement in distributed computing systems. However, designing applications for distributed systems is a difficult task, particularly when components of the system can fail. The difficulty of this task increases with the severity of failures encountered. Arbitrary process failures are generally much harder to overcome than failures that are restricted, e.g., where processes only fail by halting. Thus, techniques that restrict the disruptive behavior of faulty processes can greatly simplify the design of fault-tolerant algorithms. Such techniques effectively provide reduction mechanisms from one class of failures to a more benign class.

Message authentication is an example of a technique that imposes restrictions on the behavior of faulty processes. This technique has been used to derive simple solutions to many problems of fault-tolerance for systems with arbitrary faults. To exploit the simplicity provided by authentication, we present communication primitives that provide properties of authentication without using digital signatures. These primitives can also be extended to provide properties beyond those of authentication, thereby further restricting the types of faults that have to be overcome.

These communication primitives lead to a general methodology for designing fault-tolerant algorithms. We first design the algorithm assuming that messages are signed. We then replace signed communication in this algorithm with our

broadcast primitive. This automatically yields an equivalent non-authenticated algorithm. We illustrate this methodology by deriving new solutions to the problems of distributed agreement and clock synchronization in the presence of faults. Our solutions to the problems of Byzantine Agreement, early-stopping Byzantine Agreement, Byzantine Elections, and clock synchronization are simpler and more efficient than those previously known. Furthermore, the clock synchronization algorithm that we propose is the first one that achieves optimal accuracy with respect to real time.

Biographical Sketch

T K Srikanth was born in Bombay on November 17, 1958. He did his early schooling in Madras and Bangalore. In 1976, he joined the Indian Institute of Technology, Madras, and graduated in 1981 with a Bachelor of Technology degree in Mechanical Engineering. In the Fall of 1981, he enrolled in the graduate Computer Science program at Cornell University. He received his Master of Science degree in January 1985.

To my parents

Acknowledgements

I am extremely grateful to Sam Toueg, who has been an advisor, guide, and friend over the last few years. His contribution to this thesis has been substantial. Without his help, advice, encouragement, and patient prodding, I could not have completed this work. It has been a pleasure to work with him. I would also like to thank John Gilbert and M.J. Todd for serving on my committee, attending my exams, and helping me with my thesis. Thanks also to Tom Coleman for his comments on the thesis.

The time I have spent at the Computer Science Department here has been most enjoyable. I am thankful to everyone in the department - faculty, staff, and students - for creating a pleasant, friendly atmosphere. I am particularly grateful to the residents of Carpenter, especially those responsible for many stimulating and enlightening discussions on everything from computer science to politics. And, needless to say, for the great dinners. A lot of people outside the department have also helped in making my stay here pleasant. Thanks are due to Suresh, Hari, Kishore, Natarajan, and Pradeep, who managed to put up with me at home.

Above all else, I am grateful to my parents, my brother, grandparents, and other relatives, whose support, encouragement, and guidance has been invaluable.

Table of Contents

Table of Figures

CHAPTER 1

Introduction

An increasing number of practical computer applications are being developed on distributed systems. A distributed computing system is a collection of independent processes that share no memory and exchange messages through a communication network. An important requirement for many distributed applications is that they be *fault-tolerant*, that is, they continue to function correctly despite failures.

However, reasoning about computations in distributed systems is a difficult task, and particularly so when components of the system can fail. A process that fails can exhibit behavior that ranges from simply halting to performing arbitrary actions. In general, the design of fault-tolerant algorithms becomes easier as more restrictive failure assumptions are made.

In this thesis, we describe mechanisms that effectively restrict the disruptive behavior of faulty processes. Such mechanisms *reduce* the problem of designing algorithms for a certain failure model to that of designing algorithms for less severe failures. This allows algorithms developed for a simple model of failure to be automatically translated into solutions for a less restrictive model of failure. With a sequence of such reduction mechanisms that spans the entire range of faulty behavior, algorithms designed to overcome simple halting failures can be progressively translated to overcome arbitrary failures.

Many models have been proposed for characterizing the behavior of faulty processes. The simplest and most restrictive model of failure is that of *crash* faults, where processes fail by simply halting. At the other end of the spectrum are *malicious* or *arbitrary* failures. This is the least restrictive model of failure. In this model, we pessimistically assume that faulty processes not only deviate arbitrarily from their algorithms, but also attempt to confound the system by colluding among themselves [Peas80].

In between these two extremes, several models of failure have been proposed. *Omission* failures, where a process fails by halting or by failing to send messages, is a model proposed by Hadzilacos [Hadz83a]. A more severe type of failure, proposed by Perry and Toueg [Perr84a], is *sr−omission*, or omission to send and receive, where processes can fail to send or receive messages. Other intermediate failure modes have also been identified, e.g., *timing* failures [Coan85a, Cris84].

These models form a hierarchy of increasingly severe types of failure. In general, fault-tolerant algorithms become more complicated and more expensive to execute as the failures they tolerate become less restrictive. An algorithm designed to overcome arbitrary failures can tolerate worst-case scenarios, but the cost of designing and running the algorithm could be quite high. On the other hand, assuming that processes fail only by crashing might provide easy and inexpensive solutions, but also a correspondingly low degree of reliability. In practice, the type of failure assumed when designing an algorithm is dictated by factors like the technology of the system, the degree of reliability required for a given application, and the cost of designing and running the algorithm.

Hadzilacos [Hadz83a] describes communication primitives that effectively restrict the behavior of a process that fails by omission to that of a process that fails by crash. This allows algorithms developed to overcome crash failures to be "automatically" translated to overcome omission failures. When dealing with malicious failures, it is commonly assumed that the message system supports an authentication mechanism such as digital signatures [Diff76]. With such a mechanism, faulty processes are prevented from altering messages and from spontaneously generating spurious messages, thus effectively restricting their disruptive behavior. This greatly simplifies the task of designing algorithms for systems with malicious failures. It also often results in algorithms that are simpler and tolerate more faults than algorithms that do not assume authentication.

As will be seen in Chapter 2, using signatures is not without cost. In this thesis, we describe broadcast primitives that provide properties of authentication, thereby imposing restrictions on the behavior of faulty processes, without using digital signatures. This leads to a general methodology for the design of algorithms to overcome arbitrary failures. We first design an algorithm assuming that messages are signed. Then, replacing signed communication in this algorithm with our broadcast primitive automatically results in an equivalent non-authenticated algorithm. These broadcast primitives can also be extended to provide properties beyond those of authentication, thereby further restricting the types of faults that have to be overcome.

Apart from simplifying the design of fault-tolerant algorithms, this approach also unifies a large class of results. Previously, different algorithms were often derived for the same problem, i.e., a different solution for each model of failure. For instance, solutions for systems with message authentication did not directly provide corresponding solutions for systems without authentication. Our broadcast primitives and translation techniques yield uniform solutions for both systems.

We initially restrict our attention to *synchronous* systems. In these systems, there are bounds on the relative speeds of processes and on the message delivery rates. To illustrate our approach to the design of fault-tolerant algorithms, we concentrate on two important problems in distributed systems: reaching agreement and maintaining synchronized clocks. We first consider algorithms for Byzantine Agreement [Peas80, Lamp82a]. These algorithms allow processes to agree on the value transmitted by a distinguished process, despite arbitrary behavior by some subset of the processes. We derive an algorithm that tolerates arbitrary failures without digital signatures and is simpler than previously known algorithms.

We then derive an algorithm for *early−stopping* Byzantine Agreement. These algorithms allow processes to reach agreement in time proportional to the number of failures that actually occur during an execution of the algorithm. Our approach results in an algorithm that is simpler and more efficient than those previously known. We then derive the first known solution for Byzantine Elections [Merr84] that does not require message authentication.

We then consider systems in which there is no bound on the rate at which messages are delivered. Such systems are referred to as *asynchronous* systems. To restrict the behavior of faulty processes in asynchronous systems, we derive a broadcast primitive that provides properties of authentication in such systems. This primitive is then used to translate an authenticated randomized Byzantine Agreement algorithm in [Toue84b] into one that does not require signatures.

Finally, we consider systems in which there is a known bound on the message delays and in which processes are provided with "hardware" clocks whose rate of drift from real time is bounded. In these systems, processes can maintain "logical" clocks that are never too far apart from each other by running a *clock synchronization algorithm.* However, all previous solutions to this problem have an undesirable property: the drift from real time of these logical clocks exceeds that of the hardware clocks. That is, synchronization is achieved at the cost of a loss in accuracy with respect to real time.

We derive the first known synchronization algorithm that guarantees that logical clocks have the same accuracy as the underlying hardware clocks. We develop our algorithm by first assuming message authentication, thereby greatly simplifying the problem. We derive an algorithm for such a system and then translate this to an algorithm that does not require signatures, by applying the asynchronous broadcast primitive. This algorithm shows that accuracy need not be sacrificed for the sake of synchronization, even for systems with malicious failures and no message authentication.

CHAPTER 2

Authenticated broadcasts in synchronous systems

2.1. Introduction

Designing fault-tolerant algorithms to overcome arbitrary failures in distributed systems is a difficult task for several reasons. A faulty process can send conflicting information along disjoint paths to different processes. A faulty process relaying a message can intercept it and tamper with it. It can also generate spurious messages and pretend to be forwarding them on behalf of other processes.

Mechanisms such as authentication effectively restrict the externally visible behavior of faulty processes. Informally, authentication allows processes to generate messages that cannot be forged or modified by other processes. This prevents a faulty process from changing a message it relays, or introducing a new message into the system and claiming to have received it from some other process. This restriction on the behavior of faulty processes not only simplifies the design of fault-tolerant algorithms, but often results in algorithms that are simpler, more efficient, and tolerate more faults in the system than algorithms without authentication [Fisc83].

Cryptographic techniques that provide digital signatures can be used for message authentication [Rive78]. However, all known cryptographic schemes have disadvantages. They all require some computational and communication overhead. Furthermore, none of them has been proven unconditionally secure from attacks by malicious processes. In fact, malicious processes can break such schemes by computing or guessing the signature of another process. Although the probability of such an occurrence can be very small, it is nevertheless non-zero.

Our goal is to exploit some of the advantages of authentication without paying the price of digital signatures. We first identify properties of authenticated broadcasts. We then derive a broadcast primitive that provides these properties without using signatures. This primitive is the basis of a methodology for designing algorithms that tolerate malicious faults without authentication. Such algorithms can be designed in two stages: we first derive an authenticated algorithm, i.e., we assume that messages are authenticated. We then replace signed communication in the original algorithm with this broadcast primitive. This automatically translates the original algorithm to an equivalent non-authenticated algorithm. The resulting algorithm is as simple as the original authenticated algorithm, and furthermore it has the same proof of correctness. The design of some algorithms is further simplified by assuming that the message system provides properties beyond those of authentication. Where possible, we extend the broadcast primitive to provide these additional properties.

It should be noted that authentication is different from *secrecy* [Tane81]. Cryptographic techniques can be used to achieve both. Our primitives simulate authentication, but not secrecy.

In this chapter, we first identify properties provided by authenticated broadcasts in synchronous systems. We then present a broadcast primitive that provides these properties without using signatures. In later chapters, this primitive is strengthened to provide certain additional properties beyond those of message authentication. These primitives are then used to derive simple and efficient non-authenticated algorithms.

2.2. The model

We now define terms and state some assumptions that will be used throughout this thesis. A process is said to be *correct* if its behavior conforms to its algorithm. A *failure* or *fault* occurs if a process deviates from its algorithm, and the process is then said to be *faulty*.

We consider distributed systems of n processes, at most t of which can be faulty during the execution of any algorithm. We assume a completely connected communication network that is reliable and error-free. That is, messages are never lost or altered by the communication system. Thus, we need only consider process faults. We also assume that the immediate sender of a message can be identified by the process receiving the message. The assumption that the network is fully connected is made only for convenience. As will be seen later, this condition can be relaxed, subject to certain minimum connectivity requirements.

In this chapter and the next, we consider synchronous systems. In such systems, we assume that computation proceeds in synchronized *rounds*. Informally, a round is the interval of time during which a process first broadcasts messages, then waits to receive messages sent by processes in the same round, and finally performs some local computation and changes its state accordingly. We assume the existence of some mechanism that allows all correct processes to know when the first round begins [Coan85a].

2.3. Properties of authenticated broadcasts

Consider an algorithm that proceeds in synchronous rounds and uses authenticated broadcasts. A process p *broadcasts* a message m in round k by sending signed copies of the triple (p,m,k) to all processes (including itself). A process that receives (p,m,k) *accepts* it if it can verify p's signature. We denote these two operations by *broadcast* (p,m,k) and *accept* (p,m,k). Let **M** be the set of possible values for the message m.

Since a message broadcast by a correct process in a synchronous system is received by all correct processes in the same round, and since signatures of correct processes cannot be forged, authenticated broadcasts satisfy the following two properties:

1. (*Correctness*) If correct process p broadcasts (p,m,k) in round k, then every correct process accepts (p,m,k) in the same round.

2. (*Unforgeability*) If process p is correct and does not broadcast (p,m,k), then no correct process ever accepts (p,m,k).

One advantage of authentication is that it prevents faulty processes from modifying broadcasts of correct processes or from sending messages on behalf of correct processes. This restriction is captured by the *unforgeability* property. Authentication has another advantage: if a correct process accepts a message signed by a process p, it cannot be sure that other correct processes have also accepted this message. However, if it relays p's signed message, all correct processes receive this message, verify p's signature, and accept the message. Thus, a process accepting a signed message in a certain round can relay this message to all other processes to ensure that all processes accept this message by the next round. Therefore, if processes immediately relay every message they accept, then message authentication provides the following additional property:

3. (*Relay*) If a correct process accepts (p,m,k) in round $r \geq k$, then every other correct process accepts (p,m,k) in round $r+1$ or earlier.

By *correctness*, messages broadcast by correct processes are accepted by all correct processes in the same round. On the other hand, a message broadcast by a faulty process p and later relayed by other processes might be accepted by a correct process many rounds after it was first broadcast by p. Thus, a correct process might accept (p,m,k) in some round $r \geq k$.

It should be noted that receiving a relayed message m with p's signature does not necessarily imply that p is indeed the originator of the message. In fact, p could have been malicious and given its signature to another process. This process could then originate m "signed by p" [Lamp82a]. Therefore, *unforgeability* only guarantees that if a process accepts (p,m,k), it can infer that, *if* p is correct, then p is the originator of the message.

In the next section we derive a broadcast primitive that provides the three properties described above.

2.4. An implementation of authenticated broadcasts

Implementing authentication using digital signatures is one way of providing the properties of *correctness*, *unforgeability* and *relay* described in the previous section. In this section, we describe a broadcast primitive that provides these three properties of authenticated broadcasts without using signatures. Informally, we achieve this by requiring that to broadcast a message, a process has to use a set of processes as "witnesses" of this event. A correct process accepts a message only when it knows that there are sufficient witnesses to this broadcast. This prevents a faulty process from claiming to have received messages that were not sent to it, and allows a correct process that accepts a message to later prove that the message was indeed sent. This primitive requires $n > 3t$ processes. We later show that no broadcast primitive can provide these three properties without using signatures, if $n \leq 3t$.

Two kinds of messages are sent in this algorithm: the sender initially sends messages of type *init* to all processes (including itself). These processes now act as "witnesses" to this broadcast. Each such process then sends a message of type *echo* to all processes. A process that receives $t+1$ echoes of a message becomes a witness to the broadcast, for it knows that at most t processes are faulty and there is therefore at least one correct process among these $t+1$ processes. A process that receives $2t+1$ echoes *accepts* the message since there are enough correct witnesses. The primitive in Figure 2.1 simulates the authenticated broadcast of a message m by a process p during round k of a synchronized algorithm.

An algorithm such as that of Figure 2.1 can be viewed at two different levels. At one level, processes exchange high-level messages with the primitive operations *broadcast* and *accept*. A round is the interval of time taken to exchange these *logical* messages. At a lower level of the algorithm, these logical messages are implemented by the exchange of messages of type *init* and *echo*. We define a *phase* to be the actual interval of time it takes for a process to send such low-level messages to all processes, receive low-level messages sent by processes in the same phase, and perform some local computation. With the primitive of Figure 2.1, it takes two phases of synchronized exchange of low-level messages for each high-level message broadcast by a correct process to be accepted. Therefore, each round of the primitive is implemented by two phases of message exchange: round r of the primitive is made up of phases $2r-1$ and $2r$.

To prove that this primitive provides the three properties described above, we first prove the following lemma.

Algorithm for broadcasting and accepting (p,m,k):

Round k:

Phase $2k-1$: process p sends $(init,p,m,k)$ to all processes;

Phase $2k$: each process executes the following for each $m \in \mathbf{M}$:

if received $(init,p,m,k)$ from p in phase $2k-1$
→ send $(echo,p,m,k)$ to all **fi**

if received $(echo,p,m,k)$ from at least $n-t$ distinct processes in phase $2k$
→ *accept* (p,m,k) **fi**

Round $r \geq k+1$:

Phases $2r-1, 2r$: each process executes the following:

if received $(echo,p,m,k)$ from at least $n-2t$ distinct processes in previous phases
and not sent $(echo,p,m,k)$
→ send $(echo,p,m,k)$ to all **fi**

if received $(echo,p,m,k)$ from at least $n-t$ distinct processes
in this and previous phases
→ *accept* (p,m,k) **fi**

Figure 2.1. A broadcast primitive to simulate authenticated broadcasts

---

Lemma 2.1: If a correct process sends $(echo,p,m,k)$ then p must have sent $(init,p,m,k)$ to at least one correct process in phase $2k-1$.

Proof: Let l be the earliest phase in which any correct process q sends $(echo,p,m,k)$. If $l>2k$, process q must have received $(echo,p,m,k)$ messages from at least $n-2t$ distinct processes. Therefore, it must have received $(echo,p,m,k)$ from at least one correct process in phase $l-1$ or earlier. Hence, some correct process sends $(echo,p,m,k)$ before phase l, a contradiction. Therefore, $l=2k$, and q must have received $(init,p,m,k)$ in phase $2k-1$. □

Theorem 2.2: The broadcast primitive in Figure 2.1 provides the *correctness*, *unforgeability* and *relay* properties.

Proof:

Correctness: Since p is correct, every process receives $(init,p,m,k)$ in phase $2k-1$ and every correct process sends $(echo,p,m,k)$ in phase $2k$. Hence, every process receives $(echo,p,m,k)$ from at least $n-t$ distinct processes in phase $2k$ and every correct process accepts (p,m,k) in phase $2k$, i.e., in round k.

Unforgeability: If p is correct and does not broadcast (p,m,k), it does not send any $(init,p,m,k)$ message in phase $2k-1$. If any correct process accepts (p,m,k), it must have received $(echo,p,m,k)$ messages from at least $n-t$ processes. Hence, at least $n-2t$ correct processes must have sent $(echo,p,m,k)$ messages. By Lemma 2.1, p must have sent $(init,p,m,k)$ to at least one correct process in phase $2k-1$, a contradiction.

Relay: Let q have accepted (p,m,k) during phase i, where $i=2r-1$ or $2r$. Process q must have received $(echo,p,m,k)$ from at least $n-t$ distinct processes by phase i. Hence, every correct process receives messages $(echo,p,m,k)$ from at least $n-2t$ distinct processes by phase i, and therefore sends $(echo,p,m,k)$ by phase $i+1$. Hence, every correct process receives $(echo,p,m,k)$ from at least $n-t$ distinct processes by phase $i+1$ and accepts (p,m,k) by the end of phase $i+1$, i.e., in round $r+1$ or earlier. □

As presented here, the broadcast primitive does not terminate in a fixed number of rounds. If the broadcaster is faulty, other faulty processes can collude with the sender so that arbitrarily many rounds elapse before any correct processes accepts the broadcaster's message. However, in an application that terminates in r rounds, processes stop executing the primitive at the end of round r.

In computing the message complexity of the primitive, we consider only messages sent by correct processes. It is not possible to restrict the number of messages sent by faulty processes.

Lemma 2.3: The total number of messages sent by all the correct processes for each broadcast by a correct process is $O(n^2)$.

Proof: When a correct process p broadcasts (p,m,k), each correct process broadcasts one $(echo,p,m,k)$ message to each process. Hence, the total number of messages sent by all correct processes is $O(n^2)$. □

Although the primitive requires $O(n^2)$ messages for each broadcast by a correct process, it also provides for the automatic relay of all broadcasts accepted by correct processes. Thus, processes need never explicitly relay messages they accept. In the applications we study, this compensates for the cost of each broadcast.

To reduce the message complexity from that shown in Lemma 2.3, we isolate a set of $3t+1$ processes called the *reflectors*, and execute the primitive with n replaced by $3t+1$ as follows: only reflectors send *echo* messages to all the other processes, other processes only receive these messages and accept messages according to the primitive. It can be verified that the properties of Theorem 2.2 still hold. Since there are $O(t)$ reflectors, the total number of messages sent by all correct processes for each broadcast by a correct process is $O(nt)$.

However, a broadcast of a faulty process, with the collusion of other faulty processes, can cause correct processes to send $O(n^2)$ messages for each round of the algorithm. We can modify the primitive to reduce the number of messages sent as a result of such faulty broadcasts. This modified broadcast primitive imposes additional restrictions on the behavior of faulty processes and thereby reduces the number of messages sent. This primitive is described in detail in Section 3.2.4.

CHAPTER 3

Agreement algorithms

3.1. Introduction

In this chapter, we study the problem of reaching agreement among processes in a distributed system, even when some of them are faulty. This problem, called the *Byzantine Generals Problem* or *Byzantine Agreement*, has been extensively studied [Peas80, Lamp82a, Dole82c, Dole83], and is considered to be a central issue in the design of fault-tolerant systems [Moha83, Garc84]. Formally, Byzantine Agreement requires that when a message is sent by a *transmitter* to a set of processes, the following two conditions are met:

(1) *Agreement* : All correct processes agree on the same message,

(2) *Validity* : If the transmitter is correct, then all correct processes agree on its message.

If **M** is the set of messages that the transmitter can send, then correct processes must reach agreement on a message $m \in \mathbf{M} \cup \{\text{"sender faulty"}\}$.

It has been shown that even when processes fail only by crashing, at least $t+1$ phases of communication are needed by any algorithm that tolerates up to t faulty processes [Dole83, Fisc83, Hadz83b]. However, for this lower bound to apply, t processes must actually be faulty during execution of the algorithm, and it must be required that all correct processes reach agreement in the same phase. If fewer than t processes fail, and if processes can arrive at their decisions in different phases, it is possible to reach agreement earlier, in time proportional to the number of failures that actually occur. Such algorithms are said to exhibit *early stopping* [Dole82d].

In this chapter, we show how our approach of restricting the behavior of faulty processes with broadcast primitives results in solutions to the Byzantine Agreement and early-stopping Byzantine Agreement problems that are simpler and more efficient than those previously known.

Another variation of the Byzantine Agreement problem is the Byzantine Elections problem [Merr84]. Solutions to this problem allow processes to stop early under certain circumstances. We translate the authenticated algorithm in [Merr84] to derive the first known Byzantine Elections algorithm that does not require signatures.

3.2. Byzantine Agreement

3.2.1. Introduction

The Byzantine Agreement algorithm of Lamport et al. [Lamp82a] achieves the lower bound on the number of rounds by reaching agreement in $t+1$ phases without requiring message authentication. However, the number of messages is exponential in the number of rounds. Of the Byzantine Agreement algorithms in which only a polynomial number of messages are sent, the best known requires $2t+3$ phases and has a message complexity of $O(nt+t^3\log t)$ bits [Dole82b]. An algorithm by Coan [Coan85b] can stop in fewer phases, but has a higher message complexity. These algorithms are unintuitive and hard to understand, as are most non-authenticated algorithms. On the other hand, Dolev and Strong describe an authenticated Byzantine Agreement algorithm that is easy to understand and prove correct [Dole83]. We use our approach to automatically translate an authenticated algorithm similar to that in [Dole83] into an efficient non-authenticated algorithm for Byzantine Agreement. This solution improves on known algorithms by terminating in $2t+1$ phases using $O(nt^2 \log n)$ bits. Further, this algorithm is as simple as the authenticated algorithm from which it is translated and it has the same proof of correctness.

3.2.2. Byzantine Agreement using authenticated broadcasts

We now consider an authenticated algorithm for Byzantine Agreement. This algorithm, presented in Figure 3.1, is a simple modification of that in [Dole83]. We first restrict ourselves to the case where messages are binary, i.e., processes attempt to reach agreement on a value $m \in \{0,1\}$. Algorithms for multivalued agreement are described in Section 3.2.5. The following is an informal description of the binary algorithm.

The algorithm proceeds in synchronous rounds. In this algorithm, the only value broadcast by correct processes is 1. The value 0 is decided upon by default. A correct transmitter broadcasts 1 in round 1 if agreement is to be reached on the value 1. Otherwise, the transmitter remains silent. Process p sets the variable *value* to 1 in round r if it has accepted r messages, each message signed by a distinct process, and if the originator of the round 1 message is the transmitter. If p sets *value* to 1 in round r and has not already broadcast a message, it signs and broadcasts its own message in round $r+1$ and relays the r messages that caused it to set its *value* to 1. The *correctness* and *relay* properties of authenticated broadcasts ensure that p's signed message, and all r messages that caused p to set *value* to 1, are accepted by all correct processes by round $r+1$. Hence, they too will set *value* to 1.

If 0 is the value to be agreed upon, then a correct transmitter does not broadcast any message. By *unforgeability* of authenticated broadcasts, no correct ever process accepts any message originating from the transmitter. Hence, no correct process sets *value* to 1, and all correct processes decide on the default value 0.

Theorem 3.1: Byzantine Agreement can be achieved in $t+1$ rounds with $O(n^2 t \log n)$ message bits using the authenticated algorithm in Figure 3.1.

```
process p:
 if p is the transmitter     → value:= m;          /*m ∈ {0,1} */
  [] p is not the transmitter → value:= 0 ;
 fi
for r := 1 to t+1 do
  if value=1 and p has not broadcast a message in earlier rounds
       → broadcast (p,1,r);
         relay the r−1 messages accepted in previous rounds
         that caused value to be set to 1   fi
  if in rounds r'≤r, accepted (p_k,1,k) for all k, 1≤k≤r,
     where p_1 is the transmitter, and all the p_k's are distinct
       → value:= 1   fi
od
 decide value.
```

Figure 3.1. A Byzantine Agreement algorithm using authenticated broadcasts

---

Proof: The proof of correctness of the algorithm of Figure 3.1 is similar to that in [Dole83]. We first show that *Validity* is achieved. That is, if the transmitter p_1 is correct, then every correct process decides on the transmitter's value.

(i) If agreement is to be reached on the value 1, the transmitter initially sets *value* to 1 and broadcasts $(p_1,1,1)$ in round 1. By *correctness*, every correct process accepts $(p_1,1,1)$ in round 1. Therefore, every correct process sets *value* to 1. Once a process sets *value* to 1, it does not change it and hence decides on 1.

(ii) If agreement is to be reached on 0, the transmitter initially sets *value* to 0 and does not broadcast any message in round 1. By *unforgeability*, no correct process ever accepts the message $(p_1, 1,1)$. Therefore, every correct process retains $value=0$ throughout the algorithm, and when it terminates, decides on 0.

To show that *Agreement* is reached, we consider the following two cases:

(i) If some correct process p sets *value* to 1 at the end of round $r<t+1$, it must have accepted messages $(p_k,1,k)$ from at least r distinct processes p_k, $1 \le k \le r$. Note that $p \ne p_k$ for $1 \le k \le r$. In round $r+1$, it broadcasts $(p,1,r+1)$ and relays $(p_k, 1,k)$, $1 \le k \le r$. By the *correctness* and *relay* properties, every correct process accepts these $r+1$ messages and sets *value* to 1 in round $r+1$

(ii) If a correct process sets *value* to 1 in round $t+1$, it must have accepted messages $(p_k,1,k)$ from $t+1$ distinct processes p_k. At least one of these processes must be correct. This correct process, say p_r, broadcast $(p_r,1,r)$ in round $r \le t+1$. Therefore, p_r set *value* to 1 in round $r-1<t+1$. By case (i), every correct process sets *value* to 1 by the end of round $t+1$.

Thus, if a correct process sets *value* to 1, every correct process sets *value* to 1 and decides on 1. Otherwise, every correct process has $value=0$ and decides on 0. Thus, *Agreement* is satisfied.

The algorithm requires $t+1$ rounds. Each process broadcasts at most one message and relays up to $O(t)$ signed messages. Thus, correct processes send a total of $O(n^2 t)$ messages. Assuming that each signature requires $O(\log n)$ bits, the algorithm requires $O(n^2 t \log n)$ bits of information exchange. □

3.2.3. A simple non-authenticated algorithm for Byzantine Agreement

The proof of the authenticated algorithm of Figure 3.1 shows that the only properties of authenticated broadcasts it relies on are *correctness*, *unforgeability*, and *relay*. Cryptographic techniques that provide digital signatures can be used for message authentication, and hence for the implementation of authenticated broadcasts. However, the correctness of the authenticated algorithm does not depend on this particular implementation, and any other implementation of authenticated broadcasts providing these three properties can be used instead. In Section 2.4, we described a broadcast primitive providing these three properties. Replacing signed communication in the authenticated algorithm with this primitive, directly yields an equivalent non-authenticated algorithm. In addition, the *relay* property of the primitive ensures that messages that are accepted by correct processes are automatically relayed to all other processes. Hence, processes need not explicitly relay accepted messages.

In Figure 3.2, we present the non-authenticated algorithm derived by replacing signed communication in the algorithm of Figure 3.1 with the broadcast primitive of Figure 2.1. Each logical round of the algorithm corresponds to two phases of the underlying primitive.

Theorem 3.2: Byzantine Agreement can be achieved in $2t+2$ phases with $O(n^2 t \log n)$ message bits using the non-authenticated algorithm in Figure 3.2.

Proof: The proof of correctness is identical to that of the authenticated algorithm in Theorem 3.1.

The algorithm requires $2t+2$ phases since each round of the algorithm is implemented by two phases of the underlying primitive. Since each correct process executes at most one broadcast, correct processes send a total of $O(n^2 t)$ messages if the primitive described in Section 3.1.4 is used. Assuming that each message is $O(\log n)$ bits long, correct processes exchange a total of $O(n^2 t \log n)$ bits. □

```
process p:
 if p is the transmitter      → value:= m;        /*m ∈ {0,1} */
 [] p is not the transmitter → value:= 0 ;
 fi
for r := 1 to t+1 do
  if value=1 and p has not broadcast a message in earlier rounds
       → broadcast (p,1,r)   fi
  if in rounds r'≤r, accepted (p_k,1,k) for all k, 1≤k≤r,
      where p_1 is the transmitter, and all the p_k's are distinct
    → value:= 1  fi
od
 decide value.
```

Figure 3.2. A non-authenticated algorithm for Byzantine Agreement using the broadcast primitive of Figure 2.1

The broadcast primitive requires $n>3t$ processes. Therefore, although the authenticated algorithm (in Figure 3.1) can tolerate any number of faulty processes, the equivalent non-authenticated algorithm in Figure 3.2 requires $n>3t$ processes. It is well-known that no non-authenticated Byzantine Agreement algorithm exists for $n \leq 3t$ [Lamp82a]. Therefore, if $n \leq 3t$, no broadcast primitive can provide the properties of *correctness*, *unforgeability*, and *relay* without using signatures. If such a primitive existed, it could be used to translate the authenticated algorithm of Figure 3.1 into a non-authenticated algorithm for $n<3t$ processes, contradicting the lower bound on the number of processes required to reach Byzantine Agreement. Hence, our broadcast primitive is optimal with respect to the number of faulty processes that can be tolerated.

We now outline some simple optimizations to reduce the message and time complexity of the non-authenticated algorithm. The first optimization is to reduce the message complexity by isolating a set of $3t+1$ *active* processes [Dole82b, Dole83]. The remaining processes are denoted *passive*. Only active processes broadcast messages according to the algorithm. Passive processes only accept messages, and decide on a value 1 only if they accept this message from at least $t+1$ distinct processes. Otherwise, they decide on 0. The proof of Theorem 3.2 can be easily extended to show the correctness of this algorithm. Since there are $O(t)$ active processes, each broadcasting at most once, we have the following result:

Corollary 3.3: Byzantine Agreement can be achieved in $2t+2$ phases using $O(nt^2 \log n)$ bits.

It is clear that in the *authenticated* algorithm, messages signed and broadcast in the last round will not be relayed further. Hence, processes need not sign these messages. Consider a modified authenticated algorithm where processes broadcast *unsigned* messages in round $t+1$. Translating this algorithm into a non-authenticated one saves one phase: in round $t+1$, our broadcast primitive is not needed and a one-phase unsigned broadcast suffices. However, we must guarantee that a message accepted by a correct process in round t (i.e., by phase $2t$) is accepted by all correct processes by phase $2t+1$, i.e., within one additional phase. This is achieved by using a simple version of the primitive in round t, for example, that of Figure 2.1. We now have the following result:

Corollary 3.4: Byzantine Agreement can be reached in $2t+1$ phases using $O(nt^2 \log n)$ bits.

The communication complexity can be further reduced, at the cost of an extra phase, with another well-known technique [Dole82b, Dole83]. The $3t+1$ active processes described earlier run the agreement algorithm strictly among themselves, and broadcast their decision directly to all the passive processes only at the end. The passive processes decide on the majority value. Since there are $O(t)$ active processes which reach agreement in $2t+1$ phases, this algorithm terminates in $2t+2$ phases and requires $O(nt+t^3 \log t)$ message bits.

3.2.4. A message efficient broadcast primitive

In some algorithms, correct processes are required to broadcast at most R broadcasts, for some R. For example, in the authenticated algorithm of Figure 3.1, $R=1$. In this section, we modify the primitive of Section 2.4 to ensure that faulty processes do not execute more broadcasts than that required by the algorithm.

We introduce additional types of messages: *init'* sent by processes in the third phase of a broadcast, and *echo'* sent by processes in the remaining phases of the broadcast. The primitive is described in Figure 3.3. As before, each round r corresponds to phases $2r-1$ and $2r$ of the primitive. As with the primitive of Figure 2.1, broadcasts are not restricted to binary valued messages. The value m to be broadcast can be drawn from some arbitrary set **M**.

Lemma 3.5: If a correct process sends $(echo',p,m,k)$, then p must have sent $(init,p,m,k)$ to at least one correct process in phase $2k-1$.

Proof: Let l be the earliest phase in which any correct process q sends $(echo',p,m,k)$. Suppose $l \geq 2k+3$. Process q must have received $(echo',p,m,k)$ messages from at least $n-2t$ distinct processes, i.e., it must have received $(echo',p,m,k)$ from at least one correct process in phase $l-1$ or earlier. Hence, some correct process sends $(echo',p,m,k)$ before phase l, a contradiction. Therefore, $l=2k+2$, and q must have received $(init',p,m,k)$ from at least $n-t$ processes in phase $2k+1$. At least $n-2t$ of these processes are correct and each of them must have received at least $n-2t$ $(echo,p,m,k)$ messages in phase $2k$. Hence, at least one correct process must have sent $(echo,p,m,k)$ in phase $2k$ and it must have received $(init,p,m,k)$ from p in phase $2k-1$. □

Theorem 3.6: The broadcast primitive in Figure 3.3 has the properties of *correctness*, *unforgeability*, and *relay*.

Proof: *Correctness*: Since p is correct, every correct process receives $(init,p,m,k)$ in phase $2k-1$ and sends $(echo,p,m,k)$ in phase $2k$. Hence, every

Algorithm for broadcasting and accepting (p,m,k):

/* Processes execute at most R broadcasts in the algorithm
in which this primitive is applied. */

Round k:

Phase $2k-1$: process p sends $(init,p,m,k)$ to all processes.

Each process executes the following for each $m \in \mathbf{M}$:
Phase $2k$:

if received $(init,p,m,k)$ from p in phase $2k-1$
and received at most R $(init,p,\_,\_)$ messages in all previous phases
$\rightarrow$ send $(echo,p,m,k)$ to all **fi**

if received $(echo,p,m,k)$ from at least $n-t$ distinct processes in phase $2k$
$\rightarrow$ *accept* (p,m,k) **fi**

Round $k+1$:

Phase $2k+1$:

if received $(echo,p,m,k)$ from at least $n-2t$ distinct processes q in phase $2k$
and received at most R $(echo,p,\_,\_)$ messages from q in all previous phases
$\rightarrow$ send $(init',p,m,k)$ to all **fi**

Phase $2k+2$:

if received $(init',p,m,k)$ from at least $n-t$ distinct processes in phase $2k+1$
$\rightarrow$ send $(echo',p,m,k)$ to all **fi**

if received $(echo',p,m,k)$ from at least $n-t$ distinct processes in phase $2k+2$
$\rightarrow$ *accept* (p,m,k) **fi**

Round $r \geq k+2$:

Phase $2r-1,2r$:

if received $(echo',p,m,k)$ from at least $n-2t$ distinct processes in previous phases
and not sent $(echo',p,m,k)$
$\rightarrow$ send $(echo',p,m,k)$ to all **fi**

if received $(echo',p,m,k)$ from at least $n-t$ distinct processes
in this and previous phases
$\rightarrow$ *accept* (p,m,k) **fi**

Figure 3.3. A primitive that permits up to R authenticated broadcasts per process.

correct process receives $(echo,p,m,k)$ from at least $n-t$ correct processes in phase $2k$ and accepts (p,m,k) at the end of phase $2k$, i.e. at the end of round k.

Unforgeability: If p is correct and does not execute broadcast(p,m,k), it does not send any message $(init,p,m,k)$ in phase $2k-1$, and no correct process sends $(echo,p,m,k)$ in phase $2k$. Hence, no correct process can accept (p,m,k) in phase $2k$. If some correct process accepts (p,m,k) at the end of phase $2k+2$ or later, it must have received $(echo',p,m,k)$ messages from at least $n-t$ distinct processes, i.e., it must have received $(echo',p,m,k)$ from at least $n-2t$ correct processes. By Lemma 3, p must have sent $(init,p,m,k)$ to at least one correct process in phase $2k-1$, a contradiction. Thus, no correct process ever accepts (p,m,k).

Relay: If $r=k$, then q must have received $(echo,p,m,k)$ from at least $n-t$ distinct processes in phase $2k$. Hence, in the same phase, every correct process receives $(echo,p,m,k)$ from at least $n-2t$ distinct processes and sends $(init',p,m,k)$ in phase $2k+1$. Therefore, every correct process sends $(echo',p,m,k)$ during phase $2k+2$ and every correct process accepts (p,m,k) at the end of phase $2k+2$, i.e., round $r+1$. If $r \geq k+1$, let q accept (p,m,k) in phase i where $i=2r-1$ or $2r$. q must have received $(echo',p,m,k)$ from at least $n-t$ distinct processes by phase i. Hence, every correct process receives $(echo',p,m,k)$ from at least $n-2t$ distinct processes by phase i, and therefore sends $(echo',p,m,k)$ at or before phase $i+1$. Every correct process receives $(echo',p,m,k)$ from at least $n-t$ distinct processes by phase $i+1$ and accepts (p,m,k) by the end of phase $i+1$, i.e., in round $r+1$ or earlier. $\square$

As before, in computing the message complexity of the primitive, we only consider messages sent by correct processes. In what follows, message fields marked by an "_" (underscore) can contain arbitrary values. For example, $(\_,p,\_,\_)$ refers to all messages sent by process p, and $(echo,p,\_,\_)$ refers to all messages of type *echo* sent by p.

Lemma 3.7: In an algorithm in which the primitive of Figure 3.3 is applied, each correct process sends a total of $O(Rn)$ $(\_,p,\_,\_)$ messages for each process p.

Proof: Consider a given process p. Each correct process p_i considers at most R $(init,p,\_,\_)$ messages from p and thus sends at most R $(echo,p,\_,\_)$ messages. A process p_i considers at most R $(echo,p,\_,\_)$ message from each other process p_j throughout the algorithm. Since p_i sends $(init',p,m,k)$ only on receiving at least $n-2t$ $(echo,p,m,k)$ messages, and since $n \geq 3t+1$, each correct process sends at most $\lfloor Rn/(n-2t) \rfloor \leq 3R$ $(init',p,\_,\_)$ messages. For a correct process to send a $(echo',p,m,k)$ message in phase $2k+2$, it must receive $(init',p,m,k)$ messages from at least $n-t$ processes, i.e., from at least $n-2t$ correct processes. Since each correct process sends at most $3R$ $(init',p,\_,\_)$ messages, there can be at most $\lfloor 3R(n-t)/(n-2t) \rfloor \leq 6R$ distinct $(echo',p,\_,\_)$ messages sent by each correct process. Hence, each correct process sends at most $10R$ $(\_,p,\_,\_)$ messages to each process. Note that a correct process p need send $(init',p,m,\_)$, and $(echo',p,m,\_)$ messages only for those values of m for which it sent $(init,p,m,\_)$ messages. Since p sends only R $(init,p,\_,\_)$ messages, it sends at most $4R$ $(\_,p,\_,)$ messages to each process. Thus, each correct process sends $O(Rn)$ $(\_,p,\_,\_)$ messages. $\square$

Corollary 3.8: The total number of $(\_,p,\_,\_)$ messages sent by all correct processes throughout an algorithm for each process p is $O(Rn^2)$.

As in Section 2.4, we reduce the message complexity by isolating a set of $3t+1$ processes called the *reflectors*, and execute the primitive with $n=3t+1$ as follows: only reflectors send *init'*, *echo* and *echo'* messages to all the other processes and other processes only receive these messages and accept messages according to the primitive. It can be verified that the properties of Theorem 3.6 and Lemma 3.7 still hold. Since there are $O(t)$ reflectors, by Lemma 3.7, the total number of $(\_,p,\_,\_)$ messages sent by all correct processes for each process p is $O(Rnt)$.

3.2.5. Multivalued Byzantine Agreement

Binary valued Byzantine Agreement algorithms can be extended to the multivalued case using standard techniques [Turp84, Perr85]. These schemes add a phase to the running time of the binary valued algorithms. Algorithms such as that in [Dole83] have been directly developed for the multivalued case and are no more expensive in time than the corresponding binary valued algorithms. Our goal is to develop a non-authenticated multivalued algorithm that requires no more phases than the binary algorithm in Figure 3.2. The approach used here is to first develop a multivalued authenticated Byzantine Agreement algorithm similar to that in [Dole83]. We then translate it to a non-authenticated algorithm.

This translation can be achieved by using the broadcast primitive of Figure 2.1. However, even though each correct process broadcasts at most twice throughout the authenticated algorithm, each faulty process can broadcast more than twice. This would result in a high message complexity for the translated algorithm, if the primitive of Figure 2.1 is used. Hence, to efficiently translate the authenticated algorithm to a non-authenticated algorithm, we need a broadcast primitive that restricts the number of messages sent by correct processes, in addition to providing the properties of *correctness*, *unforgeability*, and *relay*. The primitive of Section 3.2.4 was developed to satisfy these requirements. Replacing signed communication in the authenticated algorithm with this primitive directly translates it to an efficient non-authenticated algorithm.

The proofs of the authenticated and non-authenticated algorithms, written in terms of the properties of authenticated broadcasts, are identical. We present only the non-authenticated algorithm (Figure 3.4) and its proof of correctness.

The multivalued algorithm is a straightforward modification of the binary algorithm presented in Section 3.2.3. Each process maintains a set *values* of potential decision values. A process p adds m to this set in round r if it has accepted messages containing m from r distinct processes including the transmitter p_1. In the next round, p broadcasts m if it has not yet broadcast 2 distinct values. On termination, if the set *values* of process p contains exactly one element, it decides on that value. Otherwise it decides that the sender is faulty.

Theorem 3.9: Multivalued Byzantine Agreement can be achieved in $2t+2$ phases with $O(n^2 t(\log n + \log|\mathbf{M}|))$ message bits using the non-authenticated algorithm in Figure 3.4.

Proof: The proof of correctness is similar to that of Theorem 3.1 and that in [Dole83]. We first prove *Validity*.

```
process p:
 if p is the transmitter      → values:= {m};        /*m ∈ M */
  [] p is not the transmitter → values:= ∅ ;
 fi
for r := 1 to t+1 do
  if |values|≠0
    → for each m∈values that p has not broadcast in previous rounds do
          if p has not yet broadcast 2 distinct values
            → broadcast (p,m,r)   fi  od
  fi
  for each m∈M:
    if in rounds r'≤r, accepted (p_k,m,k) for all k, 1≤k≤r,
       where p_1 is the transmitter, and all the p_k's are distinct
        → values:= values∪{m}  fi
od
if |values|=1   → decide on the value m ∈ values
[] |values|≠1   → decide "sender faulty"
fi
```

Figure 3.4. A non-authenticated algorithm for multivalued Byzantine Agreement using the broadcast primitive of Figure 3.3.

If the transmitter p_1 is correct and agreement is to be reached on the value m, then p_1 broadcasts $(p_1, m, 1)$ in round 1. By *correctness*, every correct process accepts this message in round 1, and adds m to its set *values*. The transmitter does not broadcast any other value in round 1, and hence, by *unforgeability*, no correct process accepts any message $(p,m',1)$ for $m \neq m'$. Thus, every correct process has $values = \{m\}$ when it stops and decides on m.

To show that *Agreement* is satisfied, we consider two cases:

(i) If a correct process p first adds m to *values* in round $r<t+1$, it must have accepted messages (p_k, m, k) from r distinct processes p_k, $1 \leq k \leq r$. Note that

$p \neq p_k$, $1 \leq k \leq r$. In round $r+1$, p broadcasts $(p,m,r+1)$ if it has not already broadcast two distinct values. In that case, by *correctness* and *relay*, every correct process accepts these $r+1$ messages and adds m to *values*.

(ii) If a correct process p first adds m to its set *values* in round $t+1$, it must have accepted messages (p_k, m,k) from $t+1$ distinct processes p_k, $1 \leq k \leq t+1$. At least one of these processes, say p_r, must be correct. This correct process must have have first added m to its set *values* in round $r-1<t+1$. By case (i), every correct process adds m to its set *values*.

Let p be the correct process whose set *values* has the maximum number of elements at the termination of the algorithm. If process p has $|values| \leq 2$, it follows from (i) and (ii) that the set *values* of every correct process contains at least the elements that p has added to its set. Since the set *values* of p is the largest, it follows that every correct process has the same elements in *values*. Therefore, all correct processes reach identical decisions. Specifically, if process p has $|values| = 1$, then all correct processes have the same element m in *values* and decide on m. If process p has $|values| = 0$ or $|values| = 2$, then all correct processes decide that the sender is faulty.

If process p has $|values| > 2$, it follows from cases (i) and (ii) that the set *values* of every correct process contains at least two of the elements that p has added to its set. Therefore, every correct process has $|values| \geq 2$ and decides that the sender is faulty. This proves that *Agreement* is satisfied.

The algorithm requires $t+1$ rounds, i.e., $2t+2$ phases of communication. Each correct process broadcasts at most two messages. From Lemma 3.10, it follows that the total number of messages sent by correct processes is $O(n^2t)$. As in Section 3.2.3, this message complexity can be improved to $O(nt^2)$. Since messages are $O(\log n + \log|\mathbf{M}|)$ bits long, the bit complexity of the algorithm is $O(nt^2(\log n + \log|\mathbf{M}|))$ □

3.3. Early stopping Byzantine Agreement

3.3.1. Introduction

The lower bound of $t+1$ on the number of rounds required to reach Byzantine Agreement has been used to argue that solutions to this problem are too expensive to be practical. However, by relaxing certain requirements of the problem, agreement can be reached in fewer than t phases, if the actual number of failures that occur during execution of the algorithm, f, is less than t, the maximum number of failures that the algorithm can overcome.

Dolev et al. [Dole82d] identify two types of Byzantine Agreement algorithms, depending on whether the action resulting from the agreement is to be synchronized or not. These are referred to as *Immediate* Byzantine Agreement and *Eventual* Byzantine Agreement.

Byzantine Agreement is *immediate* if correct processes can also agree on the phase number at which they reach agreement. Thus, once immediate agreement is reached, processes not only agree on the action to be performed, but can also be sure that all correct processes perform the action in the same phase. If this condition is relaxed, agreement is *eventual*, for we are only guaranteed that if a correct process decides on some value, then every correct process eventually decides on the same value.

The algorithms discussed in Section 3.2 reach immediate agreement at the end of round $t+1$. In this section, we consider algorithms for eventual agreement. In [Dole82d], it is shown that, in the worst case, no algorithm can reach immediate Byzantine Agreement in fewer than $t+1$ phases, even if only $f<t$ failures actually occur during execution of the algorithm. However, eventual Byzantine Agreement can be reached in time proportional to f rather than t. Algorithms with this property are said to exhibit *early stopping* [Dole82d]. With these algorithms, one pays for the fault-tolerance actually needed, rather than always paying the worst-case time complexity. However, Dolev et al. [Dole82a] showed that at least $f+2$ phases are required to reach eventual Byzantine Agreement in the worst case.

Algorithms with early stopping have previously been developed for systems with crash faults [Lamp82b], omission faults [Hadz83a], *sr*-omission faults [Perr84a, Perr85], and malicious faults when authentication is available [Perr84b].

Early-stopping algorithms for malicious failures without message authentication were given in [Dole82d, Reis82]. These algorithms are generally regarded as complicated and difficult to understand. Furthermore, the algorithm in [Reis82] cannot tolerate more than $n/20$ simultaneous failures. One algorithm in [Dole82d] reaches eventual Byzantine Agreement in $\min(2f+5, 2t+3)$ phases and requires $O(nt^2f)$ messages. Thus, when t failures occur, this algorithm is less efficient, in time and message complexity, than the one without early stopping in Section 3.2. The second algorithm in [Dole82d] meets the lower bound on the number of phases, but requires $n>2t^2+3t+5$ processes.

In this section, we propose a simple early-stopping algorithm that overcomes up to $n/3$ malicious processes. It is simpler, terminates earlier and has a lower communication complexity than the first algorithm in [Dole82d]. Surprisingly, even if t failures occur, this algorithm is as efficient, in time and message complexity, as the best previously known algorithms that do not exhibit early-stopping.

The algorithm was inspired by an early-stopping algorithm described in [Perr84b]. This algorithm assumes that messages are authenticated and terminates in $\pi = \min(2f+4, 2t+2)$ phases. By replacing signed communication in this algorithm by our broadcast primitive, we can directly translate this into a non-authenticated algorithm that terminates in 2π phases. However, this is expensive.

The algorithm in [Perr84b] uses a mechanism that enables a process to detect broadcasts by other processes, even if it does not directly receive the broadcast message. To achieve this property, this mechanism requires two phases of communication per round of the algorithm. We note that the broadcast primitive of Figure 3.3 can be easily modified to provide a similar property without any additonal cost in the number of phases. Using this modified primitive allows us to derive a non-authenticated early-stopping algorithm that does not require any more phases than the authenticated algorithm.

In the following sections, we first formally describe this additional property that allows processes to detect broadcasts. We then derive an early-stopping algorithm assuming we have a broadcast primitive that provides this property in addition to those of authenticated broadcasts. Finally, we extend the broadcast primitive of Figure 3.3 to achieve this additional property.

3.3.2. Extending the primitive to detect broadcasts

In Chapter 2, we saw that authenticated broadcasts provide the properties of *correctness*, *unforgeability*, and *relay*. The primitive of Section 2.4 was extended in Section 3.2.4 to restrict the number of messages that are sent by correct processes. In Section 3.3.4, we further extend this primitive to exhibit an additional property. This property allows each correct process to maintain a set, *broadcasters*, that contains the names of processes that broadcast messages using the primitive. The set *broadcasters* is initially empty. The primitive updates *broadcasters* as follows:

5. *Detection of broadcasts*: If a correct process accepts (p,m,k) in round k or later, then every correct process has $p \in$ *broadcasters* at the end of round $k+1$. Furthermore, if correct process p does not broadcast any message, then a correct process can never have $p \in$ *broadcasters*.

Informally, if p broadcasts a message in round k, this property provides that the initiation of this broadcast is detected by all correct processes by round $k+1$, even if the first correct process to accept p's message does so in a later round.

3.3.3. A binary Byzantine Agreement algorithm with early stopping

In this section, we present an early-stopping Byzantine Agreement algorithm for systems of $n>3t$ processes (Figure 3.5). We first consider the case where processes reach agreement on a value $m \in \{0,1\}$. This *binary* algorithm can be extended to the case where m is drawn from an arbitrary universe **M** [Toue84a].

3.3.3.1. Informal description

We start from the algorithm for Byzantine Agreement that does not exhibit early-stopping (Figure 3.2). In Section 3.2.2, we showed that this algorithm satisfies *Agreement* and *Validity*. We now examine the conditions under which a correct process can stop before the end of round $t+1$. It will be seen that adding two conditions to the algorithm of Figure 3.2 guarantees early-stopping.

We first note that once a correct process broadcasts the value 1, it can safely stop and decide 1: its broadcast ensures that every correct process also sets value to 1 at the end of that round.

To decide 0 and stop in round $r<t+1$, a process must be guaranteed that no correct process will later decide 1. It may do so if it has proof that, in some previous round (say round $k<r$), no process broadcast a message. (Note that to decide 1 after round k, a process must accept a message broadcast in round k.) In order to prove that there was no broadcaster in some round, each process uses the set *broadcasters* provided by the primitive. If process p discovers that there are fewer than $r-1$ broadcasters by round r, this constitutes proof that there were no broadcasters in some previous round k, since each process can broadcast at most once. Hence, no correct process *ever* accepts a message originating in round k. Therefore, no correct process can later decide 1. Hence, p can safely decide 0 and stop.

The algorithm terminates by round $f+2$, where f is the number of processes that actually fail. This can be seen as follows. If a correct process decides 0, then *no* correct process can decide 1 and broadcast a message. Therefore the set *broadcasters* contains only faulty processes, and $|broadcasters| \le f$. Hence, by round

$r=f+2$ every process will have $|broadcasters|<r-1$, and, as explained above, can safely decide 0 and stop.

Consider the case when correct processes decide 1. Assume that the first correct process to decide 1, say process p_r, does so in round r. If $r<t+1$, process p_r must have received messages $(p_k,1,k)$ from $r-1$ distinct processes. Since all these $r-1$ processes must be faulty, we see that $r-1 \le f$, i.e., $r \le f+1$. Since p_r decides 1 in round r, every correct process then decides 1 by the next round, i.e., by round $f+2$. Details of the algorithm are shown in Figure 3.5.

3.3.3.2. Proof of correctness

We begin with some definitions. A process *stops at round k* if $r = k$ when it stops. A process *decides m at round k* if it stops at round k with $value = m$. We

---

```
process p: broadcasters := φ;
 if p is the transmitter      → value:= m;        /*m ∈ {0,1} */
 [] p is not the transmitter → value:= 0 ;
 fi
for r := 1 to t+1 do
  if value=1  → broadcast (p,1,r);
                stop  fi
  consider broadcasters as updated in round r:
  if | broadcasters | < r − 1   → stop   fi
  if in rounds r'≤r accepted (p_k, 1, k) for all k, 1 ≤ k ≤ r,
         where p_1 is the transmitter, and the p_k's are distinct,
    → value:= 1   fi
od
decide value
```

Figure 3.5. The early-stopping Byzantine Agreement algorithm

---

now show that the algorithm satisfies the *Validity* property of the Byzantine Agreement problem.

Lemma 3.10: If the *transmitter* is correct and broadcasts m, then all correct processes decide m.

Proof: Follows from the proof of *Validity* in Theorem 3.1. □

To prove that the *Agreement* property is achieved, we need the following two lemmas.

Lemma 3.11: If a correct process decides 1 at round r, then every correct process that executes round r (i.e., did not stop in an earlier round) decides 1 at round r or $r + 1$.

Proof: Let p be a correct process that decides 1 at round r, where $1 \leq r \leq t + 1$. Consider when p sets *value* to 1. If it is at the beginning of round 1, then p must be the *transmitter* and the proof of *Validity* can be applied. There are two other possible cases:

1. Process p sets *value* to 1 in round $r - 1$. Therefore, by round $r - 1$, p must have accepted $(p_k, 1, k)$ messages for all k, $1 \leq k \leq r - 1$, where p_1 is the *transmitter*, and the p_k's are distinct. Note that $p \neq p_k$ for all k, $1 \leq k \leq r - 1$. In round r, p broadcasts $(p, 1, r)$. From the property of *detection of broadcasts*, since p accepted $(p_k, 1, k)$ for all k, $1 \leq k \leq r - 1$, then every correct process that executes round r must have $p_k \in$ *broadcasters* for all k, $1 \leq k \leq r - 1$, at the end of round r. So at this point every correct process has $|$ *broadcasters* $| \geq r - 1$, and therefore cannot stop and decide 0 in round r.

By properties of *correctness* and *relay* of our broadcast primitive, all correct processes that execute round r must accept $(p, 1, r)$ and $(p_k, 1, k)$ for all k, $1 \le k \le r - 1$, by the end of round r. Hence, they all set *value* to 1 in round r, and decide 1 in round r or $r + 1$.

2. Process p sets *value* to 1 in round r. Since p also stops in round r it must be that $r = t + 1$. So p must have accepted some messages $(p_k, 1, k)$ for $1 \le k \le t + 1$, with distinct p_k's. One of these $t+1$ processes, say p_j, must be correct. By *unforgeability* of our broadcast primitive, p_j broadcast $(p_j, 1, j)$ in round j, decided 1 in round j, and set *value* to 1 in round $j - 1 < j$. The proof now follows by applying case 1 to p_j. □

From Lemma 3.11 it is clear that if a correct process decides 1 at round r, then no correct process decides 0 at round $r' \ge r$. We now show the analogous result if a correct process decides 0.

Lemma 3.12: If a correct process decides 0 at round r then no correct process decides 1 at round $r' \ge r$.

Proof: Let p be a correct process that decides 0 at round r. By Lemma 3.11, no correct process decides 1 at round $r' = r$. Assume, for contradiction, that a correct process q decides 1 at round $r' > r$. We must have $r < t + 1$, and therefore p decided 0 by stopping in round r with $|\mathit{broadcasters}| < r - 1$. Since q decides 1 at round $r' > r$ it must have accepted $(p_k, 1, k)$ messages for all k, $1 \le k \le r' - 1$. Hence, by *detection of broadcasts*, p must have $p_k \in \mathit{broadcasters}$ for all k, $1 \le k \le r - 1$, at the end of round r. So p has $|\mathit{broadcasters}| \ge r - 1$ at round r, a contradiction.

Satisfaction of the *Agreement* property is now proved.

Lemma 3.13: All correct processes decide the same value.

Proof: Let r be the earliest round in which a correct process decides, and p a correct process that decides in round r. Thus, no correct process decides in round $r' < r$. If p decides 1, then by Lemma 3.11, no correct process decides 0 in round $r' \geq r$. If p decides 0 then, by Lemma 3.12, no correct process decides 1 in round $r' \geq r$. Note that all correct processes stop, and therefore decide. □

Theorem 3.14: The algorithm in Figure 3.5 solves the Byzantine Agreement problem.

Proof: Immediate from Lemma 3.10 and Lemma 3.13. □

We now show the algorithm demonstrates early-stopping.

Theorem 3.15: If there are f faulty processes during execution of the algorithm, then all correct processes decide by round $\rho = \min(f + 2, t + 1)$.

Proof: All correct processes stop and therefore decide by round $t + 1$. Let r be the earliest round in which a correct process decides, and p a correct process that decides in round r. There are two possible cases.

1. Suppose p decides 1. If $r = 1$ then p must be the *transmitter*, and the proof of Lemma 3.10 shows that all other correct processes decide 1 at round 2. We now assume $r > 1$. Since p decides 1 at round $r > 1$, then p must have accepted $(p_k, 1, k)$ messages for $1 \leq k \leq r - 1$. Since r is the earliest round in which a correct process broadcasts a message, all p_k's, $1 \leq k \leq r - 1$, must be faulty, and

therefore $r - 1 \leq f$. From Lemma 3.11, every correct process decides 1 by round $r + 1 \leq f + 2$.

2. If p decides 0, then, by Lemma 3.13, all correct processes also decide 0. Therefore, no correct process ever broadcasts a message. The broadcast primitive guarantees that the set *broadcasters* maintained by a correct process contains only names of faulty processes. Hence $|\, broadcasters \,| < f + 1$, and any correct process that executes round $f + 2$ must stop at this round. □

Suppose the *transmitter* is correct. From the proof above, if $m = 1$ then all correct processes decide 1 in round 2. However, if $m = 0$ and there are f processes that fail during execution of the algorithm, it could take $f+2$ rounds for the correct processes to decide 0. This undesirable characteristic is easily eliminated by allowing only the *transmitter* to be inserted in the set *broadcasters* in round 2. With this modification, if $m = 0$, then the correct processes have $|\, broadcasters \,| = 0 < r - 1$ in round 2, and stop.

3.3.4. The broadcast primitive and its communication complexity

In this section, we extend the broadcast primitive of Figure 3.3 to exhibit the *detection of broadcasts* property. The extension involves inserting a single statement in phase $2k+1$ that updates the set *broadcasters*.

The primitive, presented in Figure 3.6, can be applied to algorithms in which correct processes broadcast at most one message throughout the algorithm (as in the algorithm of Figure 3.5). We first show that the primitive also provides the *detection of broadcasts* property. A complexity analysis is postponed to a subsequent section.

Rules for broadcasting and accepting (p,m,k):

Round k:

Phase $2k-1$: process p sends $(init,p,m,k)$ to all processes.

Each process executes the following for each $m \in \mathbf{M}$:
Phase $2k$:

if received $(init,p,m,k)$ from p in phase $2k-1$
and received only one $(init,p,\_,\_)$ messages in all previous phases
$\rightarrow$ send $(echo,p,m,k)$ to all **fi**

if received $(echo,p,m,k)$ from at least $n-t$ distinct processes in phase $2k$
$\rightarrow$ *accept* (p,m,k) **fi**

Round $k+1$:

Phase $2k+1$:

if received $(echo,p,m,k)$ from $\geq n-2t$ distinct processes q in phase $2k$
and received only one $(echo,p,\_,\_)$ messages from q in all previous phases
$\rightarrow$ send $(init',p,m,k)$ to all **fi**
if received $(init',p,m,k)$ from $\geq n-2t$ distinct processes in phase $2k+1$
$\rightarrow$ $broadcasters := broadcasters \cup \{ p \}$ **fi**

Phase $2k+2$:

if received $(init',p,m,k)$ from $\geq n-t$ distinct processes in phase $2k+1$
$\rightarrow$ send $(echo',p,m,k)$ to all **fi**

if received $(echo',p,m,k)$ from $\geq n-t$ distinct processes in phase $2k+2$
$\rightarrow$ *accept* (p,m,k) **fi**

Round $r \geq k+2$:

Phase $2r-1,2r$:

if received $(echo',p,m,k)$ from at least $n-2t$ distinct processes in previous phases
and not sent $(echo',p,m,k)$
$\rightarrow$ send $(echo',p,m,k)$ to all **fi**

if received $(echo',p,m,k)$ from at least $n-t$ distinct processes
in this and previous phases
$\rightarrow$ *accept* (p,m,k) **fi**

Figure 3.6. The broadcast primitive that updates the set *broadcasters*

Theorem 3.16: The broadcast primitive in Figure 3.6 achieves *correctness*, *unforgeability*, *relay* and *detection of broadcasts*.

Proof: The proof that the primitive achieves the first three properties follows from Theorem 3.6. We now consider *detection of broadcasts*.

We first show the second part of this property. If a correct process adds p to *broadcasters*, then it received $(init',p,\_,\_)$ messages from at least $n-2t$ distinct processes in phase $2k+1$. Therefore, at least one correct process sent an $(init',p,\_,\_)$ message in phase $2k+1$, and it must have received $(echo,p,\_,\_)$ messages from at least $n-2t$ distinct processes in phase $2k$. Hence, at least one correct process sent an $(echo,p,\_,\_)$ message in phase $2k$, and it must have received an $(init,p,\_,\_)$ message directly from p in phase $2k-1$. Therefore p executed the broadcast primitive.

We now prove the first part of *detection of broadcasts*. If a correct process accepts (p,m,k) in phase $2k$ (round k), then it must have received $(echo,p,m,k)$ messages from at least $n-t$ distinct processes in phase $2k$. Therefore, every correct process received $(echo,p,m,k)$ messages from at least $n-2t$ distinct correct processes in phase $2k$, and these must be the first $(echo,p,\_,\_)$ message that each one sent. Hence, every correct process sends $(init',p,m,k)$ messages in phase $2k+1$. So, every correct process receives $(init',p,m,k)$ messages from at least $n-t$ distinct processes in phase $2k+1$, and adds p to *broadcasters* in this phase, i.e., in round $k+1$.

If a correct process accepts (p,m,k) in phase $k' \geq 2k+2$, then it must have received $(echo',p,m,k)$ messages from at least $n-t$ distinct processes by phase k'.

Let l be the earliest phase in which any correct process q sends an $(echo',p,m,k)$ message. The proof of Lemma 3.5 shows that $l = 2k+2$, and q must have received $(init',p,m,k)$ messages from at least $n-t$ distinct processes in phase $2k+1$. Therefore all correct processes received $(init',p,m,k)$ messages from at least $n-2t$ distinct processes in phase $2k+1$, and they all add p to *broadcasters* in this phase. □

Note that the broadcast primitive need not terminate if the broadcaster is faulty, i.e., there is no a priori bound on the number of phases needed to establish the four properties. However, when a process stops in round r of an agreement algorithm, then it also stops its execution of the underlying broadcast primitive at the end of round r, i.e., at the end of phase $2r$. We now informally show that processes stopping their participation in the underlying broadcast primitive in the same round that they decide does not affect the early-stopping Byzantine Agreement algorithm presented in the previous section.

Let r be the earliest round in which a correct process decides, and let p be a process that decides in round r. If p stops and decides 1 in round r then it continues its execution of the broadcast primitive until the end of round r. From the proof of Lemma 3.11, every correct process will set $value = 1$ by the end of round r, and therefore decides 1 when it stops, regardless of p's participation in the broadcast primitive after round r. If p decides 0, stopping its participation in the broadcast primitive can only prevent a message from being accepted by some process, or prevent the insertion of a process's name into the set *broadcasters* of some other process (so the size of *broadcasters* can only *decrease* relative to its size had p never stopped executing the primitive). This cannot cause any correct process to decide 1 or to continue executing the algorithm for more rounds than if p never stopped its execution of the broadcast primitive.

We now consider the message complexity of this broadcast primitive.

Lemma 3.17: In an algorithm in which the primitive of Figure 3.6 is applied, each correct process sends a total of $O(n)$ messages of type $(\_,p,\_,\_)$ for each process p.

Proof: The proof is the same as that of Lemma 3.7, with $R=1$. □

3.3.5. Complexity of the binary Byzantine Agreement algorithm

By Lemma 3.17, each correct process sends $O(n)$ $(\_,p,\_,\_)$ messages throughout the algorithm for each process p. There are n processes, and therefore each correct process sends $O(n^2)$ messages throughout the algorithm. Hence, all correct processes send a total of $O(n^3)$ messages. However, standard techniques, such as those in Sections 3.2.2, can be used to reduce the number of messages to $O(nt^2)$. The details of this extension are omitted here and can be found in [Toue84a]. We assume it takes $\log n$ bits to encode the name of a process. Therefore, correct processes exchange $O(nt^2 \log n)$ bits during the algorithm. From Theorem 3.15, correct processes decide by round $\rho=\min(f+2,t+1)$. The broadcast primitive requires two phases of communication exchange for each round of the algorithm in which it is applied. Therefore, it follows that correct processes decide and stop within $\min(2f+4,\ 2t+2)$ phases.

3.3.6. Multivalued Byzantine Agreement with early stopping

In [Toue84a], this algorithm is extended to the case where the *transmitter's* value m is drawn from an arbitrary universe $\mathbf{M}$. In that paper, we show that multivalued Byzantine Agreement algorithm can be achieved with $O(nt^2)$ messages and a total of $O(nt \cdot \log|\mathbf{M}| + nt^2 \cdot \log n)$ message bits. If there are f faulty processes during the execution of the algorithm, then all correct processes decide and stop within $\min(2f + 4, 2t + 2)$ phases. By comparison, the algorithm in [Dole82d] terminates in $\min(2f+5, 2t+3)$ phases and requires $O(nt^2 f(\log n + \log|\mathbf{M}|)|\mathbf{M}|)$ message bits.

3.4. Byzantine Elections

The problem of Byzantine Elections [Merr84] involves the forecasting of election results in a synchronous network of unreliable processes. If an election is not close, these algorithms allow accurate forecasting of results in less than $t+1$ rounds. Thus, although processes might not agree on the votes of individual processes until the algorithm terminates, they obtain enough information to predict the outcome of the election in fewer than $t+1$ rounds. Merritt has proposed a solution to this problem that assumes message authentication. No non-authenticated algorithm for Byzantine Elections was known. In this section, we derive one from the authenticated algorithm proposed by Merritt [Merr84] using our broadcast primitive.

Consider a system in which the processes have to agree on the votes of a set of v processes called *voters*. We only consider the algorithm for Notarized Elections [Merr84], where a set of w processes are assigned to be *witnesses*, with $w \geq 2t$.

Witnesses do not themselves vote, they just sign and forward messages from the voters and other witnesses. During each round of the algorithm, each process (voter or witness) choses a value as the vote of each voter. We assume there are a total of $n = v + 2t$ processes.

The requirements of an algorithm for Byzantine Elections are :

(1) During any round j of the algorithm, there is never any disagreement on the vote of a correct process.

(2) All correct processes reach Byzantine Agreement on every vote when the algorithm terminates.

(3) After round j, $1 \le j \le t+1$, values chosen as the votes of at most $t-j+1$ processes are different from those eventually chosen. This allows processes to arrive at a decision earlier than round $t+1$ if the election is not close.

An authenticated algorithm for Notarized Byzantine Elections from [Merr84] is presented in Figure 3.7. The following notation is used: a value signed by a voter i is an $i-vote$. The signature of a witness of an $i-vote$ is an *affidavit* for that $i-vote$.

The complete proof of correctness of this algorithm is found in [Merr84]. The proof is outlined below, highlighting those portions that identify the properties of authentication required by the algorithm.

(1) If a process accepts an affidavit from a correct witness p for an $i-vote$ in round j, then every process accepts the $i-vote$ and at least t affidavits for it by round $j+1$. This follows from the fact that if a correct witness finds an

Round 1:
  Each voter i **broadcasts** $i-vote$.

Round j , $2 \le j \le t+1$:
  Each witness w does the following:
    For every voter i:
      if witness w has **accepted** an $i-vote$ with at least $j-2$ distinct affidavits
        → the $i-vote$ is valid **fi**
      Sign any new valid $i-votes$, producing a new affidavit.
      broadcast every valid $i-vote$ or affidavit for a valid vote that
      was not **broadcast** by w in earlier rounds.

Decision procedure for round j, $1 \le j \le t+1$:
  if process p has **accepted** exactly one $i-vote$ with at least $j-1$
    distinct affidavits (including its own, if p is a witness),
    → decide on the value signed as process i's vote,
  [] otherwise
    → decide *error* as i's vote
  fi

Figure 3.7. Authenticated algorithm for Notarized Byzantine Elections [Merr84].

$i-vote$ valid, it broadcasts an affidavit for that vote and also relays the vote and all the affidavits that caused it to find the $i-vote$ valid. Hence, by *correctness* and *relay*, every correct witness finds the $i-vote$ valid in round j, and broadcasts an affidavit.

(2) If a correct process changes its decision for some process i after round j, then at least $j-1$ witnesses are faulty. This is shown by considering the various situations in which a correct process could change its decision. In each case, either the $j-1$ witnesses that caused it to decide on a value at round j or the witnesses that cause it to later change its decision are faulty.

(3) There is never any disagreement on the vote of correct processes. A correct process i broadcasts its vote to all other processes in round 1. By *correctness*, every correct process accepts this message in round 1, and decides on this value. Since process i does not broadcast any other vote, by *unforgeability*, no correct process finds any other $i-vote$ valid.

(4) At the end of round $t+1$, correct processes agree on every vote. By (3), agreement is guaranteed on votes of correct processes. If a correct witness broadcasts an affidavit for an $i-vote$, then it also relays the $i-vote$ and all the affidavits that caused it to find the $i-vote$ valid. Hence, by *correctness* and *relay*, every process finds the $i-vote$ valid in the next round and further, every correct witness also broadcasts an affidavit for this $i-vote$. If a process finds an $i-vote$ valid at the end of round $t+1$, it must have accepted the $i-vote$ and accepted affidavits from at least t witnesses. If voter i is faulty, at least one of these witnesses is correct and hence every other correct process finds the $i-vote$ valid. From this, it can be seen that correct process either agree on an $i-vote$ or decide on *error* as the vote.

(5) In (2), we saw that if any value is changed after round j, there are at least $j-1$ faulty witnesses. Hence, at most $t-j+1$ voters are faulty, and hence correct processes always agree on the votes of the remaining (correct) voters. That is, agreement is reached on $max(0, v+j-t-1)$ votes.

Each witness relays a vote and $O(t)$ affidavits for each voter. Thus, the total number of messages exchanged is $O(nvt^2)$.

Round 1:
  Each voter i **broadcasts** $i-vote$.

Round j , $2 \le j \le t+1$:
  Each witness w does the following:
    For every voter i:
      if witness w has **accepted** an $i-vote$ with at least $j-2$ distinct affidavits
        $\rightarrow$ the $i-vote$ is valid **fi**
      broadcast every valid $i-vote$ or affidavit for a valid vote that
      was not **broadcast** by w in earlier rounds.

Decision procedure for round j, $1 \le j \le t+1$:
  if process p has **accepted** exactly one $i-vote$ with at least $j-1$
    distinct affidavits (including its own, if p is a witness),
    $\rightarrow$ decide on the value signed as process i's vote,
  $[]$ otherwise
    $\rightarrow$ decide *error* as i's vote
  fi

Figure 3.8. A non-authenticated algorithm for Notarized Byzantine Elections.

From the proof outlined above, we see that the properties of authentication required by the algorithm are the three properties described in Section 2.2. Hence, the broadcast primitive of Figure 2.1 can be used in place of authenticated communication, resulting in an equivalent non-authenticated algorithm for Byzantine Elections. Once again, processes need not relay accepted votes or affidavits since the underlying primitive achieves this. The non-authenticated algorithm is described in Figure 3.8. This algorithm requires a total of $n > 3t$ processes, of which at least $2t$ are designated as witnesses.

Theorem 3.18: Byzantine Elections without authentication can be solved in $t+1$ rounds or $2t+2$ phases with $O(nvt^2)$ messages with the algorithm of Figure 3.8.

Proof: The proof of correctness is exactly the same as that of the authenticated algorithm [Merr84], stated in terms of the three properties of authenticated systems.

Each witness broadcasts an affidavit for each vote of each voter, hence the total number of messages broadcast is $O(nvt^2)$, the same as that in the original authenticated algorithm. $\square$

3.5. Discussion

The broadcast primitive of Section 2.4 was extended in Section 3.2.4 to restrict the number of messages broadcast by each process. In some applications, it might be desirable to restrict processes to effectively broadcast only one message per round of the algorithm. That is, in addition to the properties of *correctness*, *unforgeability*, and *relay*, we might be interested in the following property:

4. (*Uniqueness*) If a correct process accepts (p,m,k) in round k, no correct process accepts (p,m',k) in round k where $m \neq m'$.

The broadcast primitive of Figure 2.1 can be easily extended to achieve this property with the following modification: in phase $2k$, a correct process sends $(echo,p,m,k)$ only for the first *init* message it receives from process p in phase $2k-1$. All additional *init* messages from p in the same phase are ignored. We now show that *uniqueness* is achieved with this modification.

Assume that two correct processes accept (p,m,k) and (p,m',k) respectively in round k, with $m \neq m'$. Then, at least $n-t$ processes sent $(echo,p,m,k)$ and at least $n-t$ processes sent $(echo,p,m',k)$ in phase $2k$. This implies that at least one correct process sent both $(echo,p,m,k)$ and $(echo,p,m',k)$ in phase $2k$, a contradiction.

The Crusader Agreement problem, a weaker version of the Byzantine Agreement problem, has been defined in [Dole82c]. The problem requires that, when a *transmitter* sends a message to a set of processes,

(1) If the transmitter is correct, then all correct processes agree on its message.

(2) All correct processes that do not explicitly know that the transmitter is faulty agree on its message.

It can be seen that solutions to the Crusader Agreement algorithm achieve the properties of *correctness*, *unforgeability* and *uniqueness*. However, the solution proposed in [Dole82c] does not have the *relay* property, and therefore it cannot model the relaying of signed messages, which is crucial in simulating authentication. The first two phases of our primitive achieve Crusader Agreement.

In some authenticated algorithms such as those in [Dole83, Halp84], a process accepting a message appends its signature to this message and then broadcasts it. When a process receives a message with a list of signatures, it can verify from these signatures that each process on the list actually broadcast the message. Non-authenticated algorithms can be derived from such algorithms in one of two ways. The first approach involves modifying the authenticated algorithm so that when a process accepts a message, it signs the message, and also relays all the messages that caused it to accept the message. Thus, messages contain exactly one signature each. We used this approach for the authenticated Byzantine Agreement algorithm in Section 3.1.

Another approach is to require that when a process q accepts a message of the form $m{:}p_1{:}p_2 \cdots {:}p_k$, where each p_i is the signature of a process, q considers the message to be valid only if has also accepted each prefix of the message, i.e., it has also accepted each of $m{:}p_1$, $m{:}p_1{:}p_2, \ldots, m{:}p_1{:}p_2 \cdots {:}p_{k-1}$. This provides a simple and direct, although inefficient, method for deriving a non-authenticated algorithm from the authenticated one.

3.6. Summary

In this chapter, we illustrated the use of broadcast primitives to methodologically derive non-authenticated algorithms for agreement in distributed systems. Informally, one starts by developing an authenticated algorithm. Replacing authenticated communication with the broadcast primitive gives an equivalent non-authenticated algorithm.

Using this approach, we derived a non-authenticated Byzantine Agreement algorithm that is simpler and more efficient than those previously known. We also used this approach to derive the first known non-authenticated algorithm for Byzantine Elections.

The broadcast primitive was extended to provide properties beyond those of authentication. This extended primitive was used to derive a simple and efficient algorithm for early-stopping Byzantine Agreement. It is interesting to note that the communication complexity of the best known Byzantine Agreement algorithm *without* early stopping is no lower than the early-stopping algorithm in this section. This is somewhat surprising, since one would expect to pay some price in communication complexity for the property of stopping early.

In the next chapter, we consider authentication in asynchronous systems. We describe a primitive that provides properties of authentication in such systems. We apply this algorithm to derive non-authenticated algorithms for randomized Byzantine Agreement and for clock synchronization by translating them from authenticated ones.

CHAPTER 4

Authentication in asynchronous systems

4.1. Introduction

In this chapter, we consider systems where communication is asynchronous. In such systems, there is no bound on the time it takes for messages to be delivered. Messages sent by correct processes are eventually received by all correct processes, but this could take an arbitrarily long time. Hence, there can be no fixed bound on the duration of a phase, and the phases of processes are not synchronized.

It has been shown that in asynchronous systems, there is no deterministic solution to the Byzantine Agreement problem in the presence of even one crash failure [Fisc85]. To overcome this limitation, randomized and probabilistic algorithms have been proposed [BenO83, Brac85, Rabi83, Toue84b]. With these algorithms, correct processes reach agreement with probability 1. The algorithms in [Rabi83, Toue84b] require an authenticated message system. In this chapter, we identify properties of authenticated broadcasts in asynchronous systems and derive a primitive that achieves these properties. We describe an application of this primitive by replacing the authenticated broadcast used in the randomized algorithm in [Toue84b] with this primitive.

4.2. Asynchronous authenticated broadcasts

In asynchronous systems there is no fixed bound on the time it takes for messages to be delivered. Hence, although we can design algorithms that proceed in phases at each process, the phases of processes are not synchronized. The properties that authenticated broadcasts in asynchronous systems satisfy are therefore weaker versions of those described in Chapter 2, and can be stated as follows:

1. *(Correctness)* If correct process p broadcasts (p,m,k), then every correct process accepts (p,m,k).
2. *(Unforgeability)* If process p is correct and does not broadcast (p,m,k), then no correct process ever accepts (p,m,k).
3. *(Relay)* If a correct process accepts (p,m,k), then every other correct process accepts (p,m,k).

Rules for broadcasting and accepting (p,m,k):

Round k (*phase* k): process p sends $(init,p,m,k)$ to all processes.

Each process executes the following for any $m \in \mathbf{M}$:

if received $(init,p,m,k)$ from p **and** received only one $(init,p,\_,k)$ from p
  → send $(echo,p,m,k)$ to all **fi**

if received $(echo,p,m,k)$ from at least $n-t$ distinct processes in previous phases
  → *accept* (p,m,k) **fi**

if received $(echo,p,m,k)$ from at least $n-2t$ distinct processes in previous phases
  and not sent $(echo,p,m,k)$
  → send $(echo,p,m,k)$ to all **fi**

Figure 4.1. A primitive to simulate asynchronous authenticated broadcasts

---

The synchronous broadcast primitive of Figure 2.1 can be easily modified to derive an asynchronous primitive with the three properties described above (Figure 4.1).

Theorem 4.1: The primitive of Figure 4.1 achieves the properties of *correctness*, *unforgeability*, and *relay* in asynchronous systems.

Proof: The proof closely follows that of Theorem 3.2 for synchronous systems. We use our assumption that messages sent out by correct processes are eventually received by all correct processes.

Correctness: To broadcast (p,m,k), a correct process p sends $(init,p,m,k)$ to all. Every correct process eventually receives this and sends $(echo,p,m,k)$ to all. Every correct process therefore receives $(echo,p,m,k)$ from at least the $n-t$ correct processes and accepts (p,m,k).

Unforgeability: If correct process p does not broadcast (p,m,k), it does not send $(init,p,m,k)$ to any process. Thus, a correct process can receive $(echo,p,m,k)$ from at most the t faulty processes, and hence no correct process sends $(echo,p,m,k)$. Thus, no correct process accepts (p,m,k).

Relay: If a correct process accepts (p,m,k), it must have received $n-t$ $(echo,p,m,k)$ messages. Every correct process receives at least $n-2t$ of these and sends $(echo,p,m,k)$ to all. Hence, every correct process accepts (p,m,k). □

As in the synchronous system, we can show that a broadcast by a correct process using the primitive of Figure 4.1 requires $O(n^2)$ messages. However, in each round, each faulty process can again cause each correct process to send $O(n \cdot min(t,|\mathbf{M}|))$ messages. It is possible to modify this asynchronous primitive so that each correct process sends only $O(n)$ messages for each process in each round.

In general, the asynchronous primitive of Figure 4.1 can be used in developing non-authenticated algorithms that do not proceed in synchronized phases. An example of this is clock synchronization, as shown in Chapter 5. Another example is presented in the next section.

4.3. Randomized Byzantine Agreement

A randomized algorithm for Byzantine Agreement was presented by Rabin [Rabi83]. This was improved by Toueg [Toue84b] to overcome malicious failures of up to a third of the processes. The latter algorithm consists of iterations with two rounds. The first round uses authenticated broadcasts and the second round uses a non-authenticated broadcast primitive described in that paper. The algorithm is presented in Figure 4.2. The algorithm reaches agreement in an expected number of iterations that is a small constant independent of n and t.

In the original algorithm, a process p justifies the message it sends in the second round by broadcasting the list of signed messages it receives in the first round. From the proof of correctness of the algorithm [Toue84b], we see that the only properties of authentication needed by this algorithm are the three properties that are described in Section 4.2 and are provided by our asynchronous broadcast primitive. Therefore, we can translate this authenticated algorithm to a non-authenticated one by simply replacing the direct authenticated broadcast used in the first round with the primitive of Figure 4.1.

With our primitive, signed messages accepted by process p in the first round are automatically relayed to all, and therefore p does not have to explicitly relay

```
Process P_i: M := M_i
for k =1 to k =R do
(* Round 1 *)
  broadcast M;
  wait to accept M-messages from n-t distinct processes;
  proof := set of accepted messages;
  count(1) := number of accepted messages with M = 1;
  if count(1) ≥ n-2t  → M := 1
  [] count(1) ≤ n-2t  → M := 0;
  fi

(* Round 2 *)
  echo_broadcast [M, proof];
  wait to accept [M, proof]-messages, with correct proofs, from n-t distinct processes;
  count(1) := number of accepted messages with M =1;
  s_k := compute_secret(k);
  M := 0;
  if (s_k = 0 and count(1) ≥1) or (s_k = 1 and count(1) ≥2t+1)   → M := 1;  fi

od
```

Figure 4.2. An authenticated asynchronous binary agreement protocol [Toue84b]

(broadcast) this list of accepted messages. Hence, the translation *reduces* the original communication complexity by a factor of n.

As in Rabin's algorithm [Rabi83], the resulting algorithm still requires that a correct dealer use encryption to distribute shares of the secret random bits to each process individually, *before* the agreement algorithm starts.

CHAPTER 5

Optimal Clock Synchronization

5.1. Introduction

An important problem in distributed computing is that of synchronizing clocks in spite of faults. Given "hardware" clocks whose rate of drift from real time is within known bounds, synchronization consists of maintaining logical clocks that are never too far apart. Processes maintain these logical clocks by computing periodic adjustments to their hardware clocks.

Clock synchronization is an interesting and important problem since in many distributed systems it is necessary for processes to perform certain actions at approximately the same time. Also, once synchronization is achieved, processes can use their logical clocks to run synchronized algorithms such as those described in earlier chapters. For these reasons, clock synchronization in the presence of failures has received wide attention [Lamp85, Lund84, Dole84, Halp84, Halp85, Maha85].

The periodic adjustments made to the hardware clocks in order to synchronize logical clocks are potentially the source of a problem. Although the underlying hardware clocks have a bounded rate of drift from real time, the drift of adjusted logical clocks can exceed this bound. In other words, while synchronization ensures that logical clocks are close together, the accuracy of these clocks (with respect to real time) can be lower than that specified for hardware clocks. This reduction in accuracy might appear to be an inherent consequence of synchronization in the presence of failures for the following reasons. The rate of drift of faulty hardware clocks can be beyond the specified bounds, and correct logical clocks can be forced to drift with them. Furthermore, variation in message delivery times introduces uncertainty in evaluating values of clocks of other processes. All previous synchronization algorithms exhibit this reduction in accuracy [Lamp85, Halp84, Lund84, Dole84].

In this chapter we show that accuracy need not be sacrificed in order to achieve synchronization, even for systems with malicious failures and no authentication mechanism. We present the first synchronization algorithm that guarantees that logical clocks have the same accuracy as the underlying physical clocks. We show that no synchronization algorithm can achieve a better accuracy, and therefore our algorithm is optimal in this respect.

In contrast to previous results, we present a unified solution to the different versions of the problem: systems that exhibit crash, omission, or arbitrary failures with and without message authentication. With simple modifications, the solution also provides for initial clock synchronization and for the integration of new clocks.

We first present an algorithm for systems with arbitrary failures assuming that the system provides authentication. We then develop a broadcast primitive that achieves those properties of authentication required by the algorithm. Replacing authenticated communication with this primitive results in an equivalent non-authenticated algorithm. This solution is then simplified for crash and omission failures.

We show that to achieve optimal accuracy, fewer than half the clocks in the system can be faulty. With arbitrary failures, and in the absence of authentication, synchronization can be achieved only if fewer than a third of the clocks in the system are faulty [Dole84]. Our algorithm is optimal with respect to the number of faulty clocks it can tolerate for all the models of failure that we consider.

The solution presented in this paper is simple and efficient, and its message complexity is comparable to those previously published. Further comparisons with previous results are presented in Section 5.9.

5.2. The model

We consider a system of distributed processes that communicate through a reliable, error-free and fully connected message system (the connectivity condition is relaxed later). Each process has a physical "hardware" clock and computes its logical time by adding a locally determined adjustment to this physical clock.

The notation used here closely follows that in [Halp84]. Variables and constants associated with *real* time are in lower case and those corresponding to the *logical* time of a process are in upper case. The following assumptions are made about the system:

A1. The rate of drift of physical clocks from real time is bounded by a known constant $\rho > 0$. That is, if $R_i(t)$ is the reading of the physical clock of process i at time t, then for all $t_2 \geq t_1$,

$$(1+\rho)^{-1}(t_2-t_1) \leq R_i(t_2)-R_i(t_1) \leq (1+\rho)(t_2-t_1)$$

Thus, correct physical clocks are within a linear envelope of real time. We also see that the rate of drift between clocks is bounded by $dr=\rho(2+\rho)/(1 + \rho)$.

A2. There is a known upper bound t_{del} on the time required for a message to be prepared by a process, sent to all processes and processed by the correct processes receiving it.

A process is *faulty* if it deviates from its algorithm or if its physical clock violates assumption A1, otherwise it is said to be *correct*. Faulty processes may also collude to prevent correct processes from achieving synchronization. We use the term "correct clock" to refer to the logical clock of a correct process.

Resynchronization proceeds in rounds, a period of time in which processes exchange messages and reset their clocks. A process i starts a new logical clock C_i^k after the k^{th} resynchronization. Define beg^k and end^k to be the real time at which the first and last correct process respectively start their k^{th} clocks. The period $[beg^k, end^k]$ is the k^{th} *resynchronization period.*

Given the above assumptions, a *synchronization algorithm* is one that satisfies the following conditions for all correct clocks i and j, all $k \geq 1$, and $t \in [end^k, end^{k+1}]$:

1. *Agreement*: There exists a constant $D_{\max}$ such that

$$\left| C_i^k(t) - C_j^k(t) \right| \leq D_{\max}$$

2. *Accuracy*: There exists a constant γ such that for any execution of the algorithm,

$$(1 + \gamma)^{-1}t + a \le C_i^k(t) \le (1 + \gamma)t + b$$

for some constants a and b which depend on the initial conditions of this execution.

The *agreement* condition asserts that the maximum deviation between correct logical clocks is bounded. The *accuracy* condition states that correct logical clocks are within a linear envelope of real time.

Note that γ is a bound on the rate of drift of logical clocks from real time and hence is a measure of their accuracy with respect to real time. We are interested in synchronization algorithms that minimize γ. In Theorem 5.12, we show that γ cannot be smaller than ρ, the bound on the accuracy of physical clocks. Therefore, we are interested in algorithms satisfying:

3. *Optimal accuracy*: For any execution of the algorithm, for all correct clocks i, all $k \ge 1$, and $t \in [end^k, end^{k+1}]$

$$(1 + \rho)^{-1}t + a \le C_i^k(t) \le (1 + \rho)t + b$$

for some constants a and b which depend on the initial conditions of this execution.

5.3. The authenticated algorithm

The following is an informal description of a synchronization algorithm for systems with n processes of which at most f are faulty. The algorithm requires that $n \ge 2f+1$ and that messages are authenticated.

Let P be the logical time between resynchronizations. A process expects the k^{th} resynchronization, for $k \ge 1$, at time kP on its logical clock. When $C^{k-1}(t) = kP$, it broadcasts a signed message of the form (*round* k), indicating that it is ready to resynchronize. When a process receives such a message from $f+1$ distinct processes, it knows that at least one correct process is ready to resynchronize. It is then said to *accept* the message, and decides to resynchronize, even if its logical clock has not yet reached kP. A process resynchronizes by starting its k^{th} clock, setting it to $kP+\alpha$, where α is a constant. To ensure that clocks are never set back, α is chosen to be greater than the increase in C^{k-1} since the process sent a (*round* k) message. After resynchronizing, the process also relays the $f+1$ signed (*round* k) messages to all other processes to ensure that they also resynchronize. The algorithm is described in Figure 5.1. We show that it achieves *agreement* and *accuracy*. We later modify it to achieve *optimal accuracy*.

5.3.1. Proof of correctness: Agreement

We first show that the algorithm achieves the *agreement* property. Define $ready^k$ to be the earliest (real) time at which any correct process sends a (*round* k) message. We assume that the clocks C^0 of correct processes are synchronized, i.e., at $ready^1$ all correct processes are using clock C^0 and for all correct processes i and

---

```
cobegin
  if C^{k-1}(t)=kP                    /* ready to start C^k */
       → sign and broadcast (round k) fi
//
  if accepted the message (round k)   /* received f+1 signed (round k) messages */
       → C^k(t):= kP+α;               /* start C^k */
         relay all f+1 signed messages to all fi
coend
```

Figure 5.1. An authenticated algorithm for clock synchronization for process p for round k

---

j, $\left|C_i^0(ready^1) - C_j^0(ready^1)\right| \le D_{\max}$. In Section 5.7, we describe an algorithm for achieving this initial synchronization. For ease of presentation we assume that the maximum permitted deviation between correct logical clocks, $D_{\max}$, is a given constraint.

Lemma 5.1: The k^{th} resynchronization period is bounded in size. That is, there exists a constant $d_{\min}$ such that for $k \ge 1$, $end^k - beg^k \le d_{\min}$.

Proof: Let p be the first correct process to start its k^{th} clock. By definition, this occurs at beg^k. Process p must have received $f+1$ signed $(round\ k)$ messages. Since it relays all these messages, every correct process receives them and accepts the message $(round\ k)$ by time $beg^k + t_{del}$. Hence, every correct process starts its k^{th} clock by time $beg^k + t_{del}$. By setting $d_{\min} = t_{del}$, we get $end^k \le beg^k + d_{\min}$.

□

Lemma 5.2: At the end of the k^{th} resynchronization period, correct clocks differ by at most $d_{\min}(1 + \rho)$. That is, for $k \ge 1$, and for all correct processes i and j, $\left|C_i^k(end^k) - C_j^k(end^k)\right| \le d_{\min}(1 + \rho)$.

Proof: By Lemma 5.1, $end^k - beg^k \le d_{\min}$. Therefore the last correct process to start its k^{th} clock does so within $d_{\min}$ of the first correct clock doing so, and in this period, the first clock could have drifted by at most $\rho d_{\min}$. Thus, at end^k, the difference between correct clocks is at most $d_{\min}(1 + \rho)$. □

Lemma 5.3: No correct process starts its k^{th} clock until at least one correct process is ready to do so, i.e., $beg^k \ge ready^k$, for $k \ge 1$.

Proof: The first correct process to start its k^{th} clock does so only when it accepts a (*round k*) message, i.e., only when it receives (*round k*) messages from at least $f+1$ processes. Since at least one correct process must have sent a (*round k*) message, $beg^k \geq ready^k$. □

Assume that the following conditions hold for some $k \geq 1$:

S1. At $ready^k$ all correct processes are using C^{k-1}.

S2. For correct processes i and j, $\left| C_i^{k-1}(ready^k) - C_j^{k-1}(ready^k) \right| \leq D_{\max}$.

With these assumptions, we prove the following lemmas.

Lemma 5.4: All correct processes start their k^{th} clocks soon after one correct process is ready to do so. Specifically, $end^k - ready^k \leq (1+\rho)D_{\max} + t_{del}$.

Proof: The first correct process to send a (*round k*) message does so at $ready^k$. By S2, the slowest correct clock is no more than $D_{\max}$ behind. Hence, every correct process sends a (*round k*) message no later than $(1+\rho)D_{\max}$ after $ready^k$, and therefore every correct process starts its k^{th} clock within a further t_{del}. Thus, $end^k - ready^k \leq (1+\rho)D_{\max} + t_{del}$. □

By Lemma 5.4, the real time that elapses from the time a correct process sends a (*round k*) message (when C^{k-1} reads kP) to the time it starts C^k (setting it to $kP+\alpha$) is at most $(1+\rho)D_{\max}+t_{del}$. Therefore, if $\alpha \geq \left[(1+\rho)D_{\max}+t_{del}\right](1+\rho)$ then no correct process sets its logical clock backwards. Henceforth, we assume that α satisfies this relation.

Lemma 5.5: There is a bound on the period for which the k^{th} logical clock is used. That is, $end^{k+1} - end^k \leq (P-\alpha)(1+\rho) + t_{del}$.

Proof: Every correct process that sends a (*round* $k+1$) message does so no later than the time $(k+1)P$ on its clock, i.e., no later than $(P-\alpha)(1+\rho)$ after end^k. Every process starts its $k+1^{st}$ clock within a further t_{del}, thus proving the lemma. $\square$

Lemma 5.6: The maximum deviation between the k^{th} logical clocks of correct processes i and j is bounded. That is, for $t \in [end^k, end^{k+1}]$, $\left|C_i^k(t) - C_j^k(t)\right| \le D_{\max}$.

Proof: By Lemma 5.2, correct logical clocks are at most $d_{\min}(1+\rho)$ apart at end^k. By Lemma 5.5, $end^{k+1} - end^k \le (P-\alpha)(1+\rho) + t_{del}$, and clocks of correct processes can drift apart at a rate dr in this interval. Thus, in the interval $[end^k, end^{k+1}]$,

$$\left|C_i^k(t) - C_j^k(t)\right|$$

$$\le [(P-\alpha)(1+\rho) + t_{del}]dr + d_{\min}(1+\rho)$$

$$\le [P(1+\rho) + t_{del}]dr + d_{\min}(1+\rho)$$

P is chosen to satisfy the relation $D_{\max} \ge [P(1+\rho) + t_{del}]dr + d_{\min}(1+\rho)$. $\square$

Lemma 5.7: Synchronization periods do not overlap. That is, $end^k < ready^{k+1} \le beg^{k+1}$.

Proof: The first correct process to send a (*round* $k+1$) message does so no earlier than at real time $beg^k + (P-\alpha)/(1+\rho)$. Therefore, $ready^{k+1} \ge beg^k + (P-\alpha)/(1+\rho)$. Hence, by Lemma 5.1, $ready^{k+1} \ge end^k - d_{\min} + (P-\alpha)/(1+\rho)$. By Lemma 5.3, $ready^{k+1} \le beg^{k+1}$. Thus, $end^k < ready^{k+1} \le beg^{k+1}$, if P satisfies the relation $P > d_{\min}(1+\rho) + \alpha$. $\square$

From the proof of Lemmas 5.6 and 5.7, we see that D_{max} cannot be made arbitrarily small. The proof of Lemma 5.6 shows that $D_{max} \geq [(P-\alpha)(1+\rho)+t_{del}]dr + d_{min}(1+\rho)$. From Lemma 5.7, we see that $P-\alpha \geq d_{min}(1+\rho)$. Therefore, the smallest possible D_{max} that this algorithm can achieve is given by $D_{max} \geq d_{min}(1+\rho)^3+t_{del}dr$. It has been shown that, for any algorithm, D_{max} must be at least $t_{del}/2$ [Dole84].

Lemma 5.8: The algorithm in Figure 5.1 achieves *agreement*.

Proof: If assumptions S1 and S2 hold for some $k \geq 1$, then Lemma 5.6 states that the *agreement* condition is satisfied for k. We now show, by induction on k, that S1 and S2 hold for all $k \geq 1$ and therefore *agreement* is satisfied for all $k \geq 1$. As stated earlier, our initialization algorithm will guarantee that S1 and S2 are true for the base case, $k=1$.

Assume that S1 and S2 are true for some k. By Lemma 5.7, $end^k < ready^{k+1} \leq beg^{k+1}$. Thus, at $ready^{k+1}$, all correct processes use their k^{th} clocks. From Lemma 5.6 it follows that at $t=ready^{k+1}$, and for correct processes i and j, $\left|C_i^k(t)-C_j^k(t)\right| \leq D_{max}$. Thus S1 and S2 are true for $k+1$. □

5.3.2. Proof of correctness: Accuracy

We now show that the algorithm achieves *accuracy*.

Lemma 5.9: For any execution of the algorithm of Figure 5.1, there exists a constant b, such that for all correct processes i, all $k \geq 1$ and for $t \in [end^k, end^{k+1}]$:

$$C_i^k(t) \leq \frac{P}{P-\alpha}(1+\rho)t+b$$

Proof: Let $E(t_0)$ be the set of executions of the algorithm in which $ready^1 = t_0$. Consider an execution $e \in E(t_0)$ in which for all $k \geq 1$, $ready^k = beg^k$, and the clock of correct process j, C_j^k, is started at beg^k. In execution e, the physical clock of process j runs at the maximum possible rate, i.e., $(1 + \rho)$ with respect to real time. It is clear that execution e is possible.

Since C_j^k is started at beg^k for each k, it is started at least as early as any other correct C_i^k in execution e. Furthermore, between beg^k and beg^{k+1}, C_j^k increases at the maximum possible rate. Hence, C_j^k is an upper bound on the k^{th} logical clocks of all correct processes in execution e. That is, for $t \in [end^k, end^{k+1}]$, $C_i^k(t) \leq C_j^k(t)$, for any other correct process i.

We now show that C_j^k is an upper bound on the k^{th} logical clock of any correct process in any execution in $E(t_0)$. To prove this, we first show that for any $k \geq 1$, $ready^k$ in execution e is at least as early as $ready^k$ in any other execution $e' \in E(t_0)$. The proof is by induction on k.

For $k = 1$, $ready^1 = t_0$ for all executions in $E(t_0)$. Assume, for some $k > 1$, that $ready^k$ in execution e is no later than $ready^k$ in execution e'. In execution e, $beg^k = ready^k$, the k^{th} logical clock of process j is started at beg^k, and process j runs at the maximum possible rate. Therefore, $ready^{k+1} = ready^k + (P - \alpha)/(1 + \rho)$. It is easy to show that in any execution, $ready^{k+1} \geq ready^k + (P - \alpha)/(1 + \rho)$. Therefore, $ready^{k+1}$ in execution e is at least as early as that in execution e'.

In execution e, $beg^k = ready^k$ for all $k \geq 1$. By Lemma 5.3, in any execution, $beg^k \geq ready^k$ for all $k \geq 1$. Therefore, the k^{th} logical clock of process j is started no later than that of any other correct process in any execution in $E(t_0)$. Since process j also runs at the maximum possible rate, C_j^k is an upper bound on the k^{th} logical clocks of all correct processes in all executions in $E(t_0)$.

We now estimate an upper bound for C_j^k. For process j, the interval of real time between consecutive resynchronizations is $(P-\alpha)/(1+\rho)$. In this period its logical time increases by P. Therefore, for all $k \geq 1$, and for $t \in [end^k, end^{k+1}]$:

$$\frac{C_j^k(t) - C_j^1(t_0)}{t - t_0} \leq \frac{P}{P-\alpha}(1+\rho)$$

Since $C_j^1(t_0) = P + \alpha$, a constant, $C_j^k(t) \leq \dfrac{P}{P-\alpha}(1+\rho)t + b$, where b is a constant that depends on t_0. □

Lemma 5.10: For any execution of the algorithm of Figure 5.1, there exists a constant a, such that for all correct processes i, all $k \geq 1$ and for $t \in [end^k, end^{k+1}]$:

$$\frac{P}{P - \alpha + \dfrac{t_{del}}{(1+\rho)}}(1+\rho)^{-1}t + a \leq C_i^k(t)$$

Proof: Let $E(t_0)$ be the set of executions of the algorithm in which $end^1 = t_0$. Consider an execution $e \in E(t_0)$ where, for all $k \geq 1$, correct process j accepts the (*round* k) message t_{del} in real time after C_j^{k-1} reads kP. Also, C_j^k is started at end^k for all $k \geq 1$. In e, the physical clock of process j runs at the minimum possible rate, i.e., at $(1+\rho)^{-1}$ with respect to real time. Such an execution is clearly possible. It is easy to show that C_j^k is a lower bound on the k^{th} logical clocks of all correct processes in execution e. That is, for $t \in [end^k, end^{k+1}]$, $C_j^k(t) \leq C_i^k(t)$, for any other correct process i.

C_j^k is also a lower bound on the k^{th} logical clocks of all correct processes in any execution in $E(t_0)$. In execution e we have $end^{k+1} = end^k + (P-\alpha)(1+\rho) + t_{del}$.

The proof follows by Lemma 5.5 and an easy induction on k.

We now estimate a lower bound for C_j^k. For process j, $(P-\alpha)(1+\rho)+t_{del}$ is the interval of real time between consecutive resynchronizations. In this period, its logical time increases by P. Therefore, as in Lemma 5.9, for all $k \geq 1$ and $t \in [end^k, end^{k+1}]$

$$C_j^k(t) \geq \frac{P}{(P-\alpha)(1+\rho)+t_{del}}\, t + a$$

for some constant a which depends on t_0. □

Theorem 5.11: The algorithm in Figure 5.1 is a synchronization algorithm. With this algorithm, correct processes send a total of $O(n^2 f)$ signed messages per resynchronization.

Proof: By Lemma 5.8, the algorithm achieves *agreement*. Lemmas 5.9 and 10 imply that *accuracy* is achieved with $(1+\gamma)=\frac{P}{P-\alpha}(1+\rho)$. In each resynchronization round, each correct process broadcasts at most one signed message and relays at most $f+1$ signed messages to every other process. Thus, correct processes send a total of $O(n^2 f)$ signed messages per resynchronization. □

The number of bits exchanged for each resynchronization is comparable to that in [Halp84].

5.4. Achieving optimal accuracy

5.4.1. A bound on accuracy

We first show that for any synchronization algorithm, the accuracy of synchronized logical clocks cannot exceed that of the underlying hardware clocks. In what follows, define $C_i(t)=C_i^k(t)$ for $t\in[end^k, end^{k+1})$ and all $k \geq 1$.

Theorem 5.12: For any synchronization algorithm, the bound on the rate of drift of logical clocks from real time is at least as large as the bound on the rate of drift of physical clocks.

Proof: Consider an algorithm that satisfies *agreement* and *accuracy*. For simplicity, assume that all physical clocks are set to 0 at time $t=0$, i.e., $R_i(0)=0$ for all i. Then, all correct physical clocks satisfy the relation

$$(1 + \rho)^{-1}t \leq R_i(t) \leq (1 + \rho)t$$

Consider an execution of the algorithm in which all processes in the system are correct and the physical clock of each process runs at the maximum possible rate. That is, for all processes i, $R_i^{(1)}(t)=(1 + \rho)t$, where superscripts denote execution numbers. Further, assume the transmission delay for each message is exactly d, with $d \leq t_{del}/(1 + \rho)^2$. By *accuracy*, in this execution, for all correct processes i and for some constant $b^{(1)}$:

$$C_i^{(1)}(t) \leq (1 + \gamma)t + b^{(1)} \tag{1}$$

Now consider a second execution in which all processes are still correct, but have their physical clocks running at the minimum possible rate. That is, for all processes i, $R_i^{(2)}(t)=(1 + \rho)^{-1}t$. Let the transmission delay for each message be $d(1 + \rho)^2$. Again, by *accuracy*, in this execution for all correct processes i and for some constant $a^{(2)}$:

$$(1 + \gamma)^{-1}t + a^{(2)} \leq C_i^{(2)}(t) \tag{2}$$

Assume that for each process i, the initial state is the same in both executions. That is, in both executions, a process starts executing the algorithm at the same reading of its physical clock. In the second execution, physical clocks and the speed at which messages are delivered are slowed down by the same factor, $(1 + \rho)^2$, with respect to the first execution. Therefore, from within the system both executions appear identical to every process. Hence, considering a particular process i, the rate at which its logical time advances with respect to its physical time must be the same in both executions. In particular, if $R_i^{(1)}(t_1) = R_i^{(2)}(t_2)$ for some t_1 and t_2, then $C_i^{(1)}(t_1) = C_i^{(2)}(t_2)$.

Since $R_i^{(1)}(t) = (1 + \rho)t$ and $R_i^{(2)}(t) = (1 + \rho)^{-1}t$, it follows that if $t_2 = (1 + \rho)^2 t_1$, then $R_i^{(1)}(t_1) = R_i^{(2)}(t_2)$ and therefore $C_i^{(1)}(t_1) = C_i^{(2)}(t_2)$. Therefore, from equations (1) and (2), $(1 + \gamma)t_1 + b^{(1)} \geq (1 + \gamma)^{-1}(1 + \rho)^2 t_1 + a^{(2)}$ for all t_1. This implies that $\gamma \geq \rho$. □

5.4.2. An algorithm for optimal accuracy

We now describe a modification to our algorithm to achieve *optimal accuracy*. In the algorithm of Figure 5.1, correct processes start their k^{th} clocks as soon as they accept a (*round k*) message. However, there is an uncertainty of t_{del} in the time it takes for correct processes to accept a message. It is this uncertainty that introduces a difference in the logical time between resynchronizations. For the fastest clock, the logical time between resynchronizations is $P - \alpha$ (Lemma 5.9), and for the slowest clock, this interval is $P - \alpha + \dfrac{t_{del}}{(1 + \rho)}$ (Lemma 5.10). Informally, we can compensate for this as follows: if a process accepts a (*round k*) message early, it delays the starting of the k^{th} clock by $\dfrac{t_{del}}{2(1 + \rho)}$. If it accepts the message late, it advances the starting of the k^{th} clock by $\dfrac{t_{del}}{2(1 + \rho)}$. Thus, in the cases described in both Lemmas 5.9 and 5.10, the logical time between resynchronizations becomes $P-\alpha+\frac{t_{del}}{2(1+\rho)}$, as will be shown later. This will then be used to achieve *optimal accuracy*.

Suppose process i accepts (*round* k) at time t, and let $T=C_i^{k-1}(t)$. Define $\beta=\frac{t_{del}}{2(1+\rho)}$.

If $T \leq kP+\beta$, we say the (*round* k) message was accepted *early*. Process i delays the starting of C_i^k by setting it to $kP+\alpha$ when C_i^{k-1} reads $\min(T+\beta,\ kP+\beta)$. In this case, the start of C_i^k is delayed by at most β but never beyond the time when C_i^{k-1} reads $kP+\beta$.

If $T>kP+\beta$, we say (*round* k) was accepted *late*. Process i advances the starting of C_i^k, by setting it to $kP+\alpha$ when C_i^{k-1} reads $\max(T-\beta,\ kP+\beta)$. Note that C_i^k must be started when C_i^{k-1} reads $T'<T$, that is, "in the past". This is achieved by setting C_i^k to $kP+\alpha+(T-T')$ when C_i^{k-1} reads T. That is, C_i^k is set to $\min(C_i^{k-1}(t)+\alpha-\beta, kP+\alpha+\beta)$ at time t. In this case, the start of C_i^k is advanced by at most β, but is never started before C_i^{k-1} reads $kP+\beta$.

The definitions of $ready^k$, beg^k and end^k are the same as before: $ready^k$ is the earliest time at which a correct process sends a (*round* k) message; beg^k and end^k are the earliest and latest times at which some correct process starts its k^{th} clock (setting it to $kP+\alpha$).

We first show that this modified algorithm achieves *agreement* by showing that Lemmas 5.1 to 5.8 still hold.

Proof of Lemma 5.1: The first correct process to start its k^{th} clock can start it β in logical time (or $\beta(1 + \rho)$ in real time) before it accepts a *(round k)* message. Every correct process accepts *(round k)* within t_{del} of the first correct process accepting it, and starts its k^{th} clock within a further $\beta(1 + \rho)$. Therefore, $end^k - beg^k \leq t_{del} + 2\beta(1 + \rho) = 2t_{del}$. Therefore, Lemma 5.1 is satisfied with $d_{\min} = 2t_{del}$.

Proof of Lemma 5.2: As in Section 5.3.

Proof of Lemma 5.3: Consider any correct process i. By definition, $C_i^{k-1}(ready^k) \leq kP$. Let process i accept the *(round k)* message at real time t. Note that $t \geq ready^k$. If $C_i^{k-1}(t) \leq kP + \beta$, then process i delays the starting of the k^{th} clock. If $C_i^{k-1}(t) > kP + \beta$, process i starts its k^{th} clock no earlier than at real time t' such that $C_i^{k-1}(t') = kP + \beta$. Clearly, $t' \geq ready^k$. Hence, no correct process starts its k^{th} clock before $ready^k$.

Proof of Lemma 5.4: Every correct process that broadcasts a *(round k)* message does so by real time $t_1 = ready^k + (1 + \rho)D_{\max}$. Therefore, every correct process accepts *(round k)* by $t_2 = t_1 + t_{del}$. For any correct process i, $C_i^{k-1}(t_1) \geq kP$ and hence $C_i^{k-1}(t_2) \geq kP + t_{del}/(1 + \rho) = kP + 2\beta$. Thus, with the modified algorithm, every correct process starts its k^{th} clock at real time $t < t_2$. Therefore, $end^k - ready^k \leq (1 + \rho)D_{\max} + t_{del}$.

Proof of Lemma 5.5: Consider any correct process i. Process i accepts a *(round k+1)* message by real time $t = end^k + (P - \alpha)(1 + \rho) + t_{del}$. Also, $C_i^k(t) \geq kP + t_{del}/(1 + \rho)$. Therefore, process i starts its $k+1^{st}$ clock by real time t, proving the lemma.

Proofs of Lemmas 5.6, 5.7 and 5.8: As in Section 5.3.

Thus, the modified algorithm achieves *agreement*. To show that the modified algorithm achieves *optimal accuracy*, we first evaluate the bounds on the drift of logical clocks from real time.

Lemma 5.9': For any execution of the modified algorithm, there exists a constant d, such that for all correct processes i, all $k \geq 1$ and $t \in [end^k, end^{k+1}]$:

$$C_i^k(t) \leq \frac{P}{P-\alpha+\beta}(1 + \rho)t+d$$

Proof: Let $E(t_0)$ be the set of executions of the algorithm in which $ready^1 = t_0$. Consider an execution $e \in E(t_0)$ in which for all $k \geq 1$, correct process j broadcasts and accepts (*round* k) at $ready^k$. In execution e, the physical clock of process j runs at the maximum possible rate, i.e., $(1 + \rho)$ with respect to real time.

Process j accepts (*round* k) at $ready^k$, when C_j^{k-1} reads kP (i.e., early). Therefore, C_j^k is started at real time t such that $C_j^{k-1}(t) = kP+\beta$, i.e., when $t = ready^k + \beta/(1 + \rho)$. Note that no correct physical clock increases by more than β between $ready^k$ and t.

Consider another correct process i. By definition of $ready^k$, $C_i^{k-1}(ready^k) \leq kP$, and therefore, $C_i^{k-1}(t) \leq kP+\beta$. Suppose process i accepts (*round* k) when C_i^{k-1} reads T_i. This must occur after $ready^k$, and therefore, at time t, $C_i^{k-1}(t) \leq T_i+\beta$. We consider two cases:

1. If $T_i \leq kP+\beta$, then process i starts C_i^k at real time t' when $C_i^{k-1}(t') = \min(T_i+\beta, kP+\beta)$. Since both $C_i^{k-1}(t) \leq T_i+\beta$ and $C_i^{k-1}(t) \leq kP+\beta$, then $t \leq t'$.

2. If $T_i > kP+\beta$, process i starts C_i^k at real time t' when $C_i^{k-1}(t') = \max(T_i - \beta, kP+\beta)$. Therefore, $C_i^{k-1}(t') \geq kP+\beta \geq C_i^{k-1}(t)$ and $t' \geq t$.

Thus, in execution e, for any $k \geq 1$, the k^{th} clock of process i is started no earlier than that of process j. Between resynchronizations, C_j^k runs at the maximum possible rate. Therefore, C_j^k is an upper bound on the k^{th} logical clock of all correct processes in execution e. As in Lemma 5.9, we can also show that C_j^k is an upper bound on the k^{th} clock of all correct processes in any execution in $E(t_0)$.

Between every two successive resynchronizations, the logical clock of process j is advanced by P, and the time that elapses on the logical clock of j is $P-\alpha+\beta$. (For example, at the k^{th} resynchronization, the clock is set to $kP+\alpha$, the $k+1^{st}$ resynchronization occurs when this clock reads $(k+1)P+\beta$, and the new clock is set to $(k+1)P+\alpha$.) Since the clock of process j runs at $(1 + \rho)$ with respect to real time, the real time that elapses between two resynchronizations is $(P-\alpha+\beta)/(1 + \rho)$. Hence, for all $k \geq 1$ and $t \in [end^k, end^{k+1}]$:

$$C_j^k(t) \leq \frac{P}{P-\alpha+\beta}(1 + \rho)t+d$$

for some constant d that depends on t_0. □

Lemma 5.10': For any execution of the modified algorithm, there exists a constant c, such that for all correct processes i, all $k \geq 1$ and for $t \in [end^k, end^{k+1}]$:

$$\frac{P}{P-\alpha+\beta}(1 + \rho)^{-1}t+c \;\leq\; C_i^k(t)$$

Proof: Let $E(t_0)$ be the set of executions of the algorithm in which $end^1 = t_0$. Define $last^k$ to be the latest real time at which a correct process accepts $(round\ k)$. Consider an execution $e \in E(t_0)$ in which the first logical clock of a correct process j,

C_j^1 is started at end^1, and for all $k \geq 1$, process j accepts $(round\ k)$ at $last^k$, and t_{del} (in real time) after its logical clock reads kP. The physical clock of process j runs at the minimum possible rate, i.e., at $(1 + \rho)^{-1}$ with respect to real time. In the modified algorithm, since $C_j^{k-1}(last^k) = kP + 2\beta$, process j sets its k^{th} clock to $kP + \alpha + \beta$ at $last^k$.

We now show that the logical clock of process j is as slow as that of any other correct process in any execution in $E(t_0)$. That is, we show that for all $k \geq 1$ and $t \in [last^k, last^{k+1}]$, $C_j^k(t) \leq C_i^k(t)$ for any correct process i in any execution in $E(t_0)$. The proof is by induction on k.

For $k=1$, note that C_j^1 is started at $end^1 = t_0$ and process j runs at the minimum possible rate. In any execution in $E(t_0)$, for any other correct process i, C_i^1 is started no later than at end^1. Therefore, for $t \geq end^1$, and specifically for $t \in [last^1, last^2]$, we see that $C_j^1(t) \leq C_i^1(t)$. For the inductive step, assume that for some $k > 1$ and $t \in [last^{k-1}, last^k]$, we have $C_j^{k-1}(t) \leq C_i^{k-1}(t)$ for any correct process i. Define s_i to be the real time such that $C_i^{k-1}(s_i) = kP + \beta$, for any process i. Let t_i and T_i be the real and the corresponding logical time at which a process i accepts $(round\ k)$.

Consider any correct process i in any execution in $E(t_0)$. From the induction hypothesis, it follows that $s_i \leq s_j$ for all correct i. Since $s_j = last^k - \beta(1 + \rho)$, $last^k - s_i \geq \beta(1 + \rho)$. By assumption, $t_j = last^k$ and $T_j = kP + 2\beta$. We consider two cases:

1. If $T_i \leq kP + \beta$ (i.e., $t_i \leq s_i \leq s_j$), then C_i^k is set to $kP + \alpha$ no later than s_i. Since $last^k - s_i \geq \beta(1 + \rho)$, C_i^k increases by at least β between s_i and $last^k$. Therefore,

$C_i^k(last^k) \geq kP+\alpha+\beta = C_j^k(last^k)$.

2. If $T_i > kP+\beta$, then process i sets its k^{th} clock to $C_i^k(t_i) = \min(C_i^{k-1}(t_i)+\alpha-\beta, kP+\alpha+\beta)$. Since C_i^k and C_i^{k-1} increase by the same amount between t_i and $last^k$, $C_i^k(last^k) = \min(C_i^{k-1}(t_i)+\alpha-\beta, kP+\alpha+\beta)+C_i^{k-1}(last^k)-C_i^{k-1}(t_i)$. Since $C_i^{k-1}(last^k) \geq kP+2\beta$, $C_i^k(last^k) \geq kP+\alpha+\beta = C_j^k(last^k)$.

Thus, $C_j^k(last^k) \leq C_i^k(last^k)$. The physical clock of process j runs at the minimum possible rate. Therefore, for $t \in [last^k, last^{k+1}]$, $C_j^k(t) \leq C_i^k(t)$ for any correct process i in any execution in $E(t_0)$.

The logical clock of process j is incremented by P over successive resynchronizations. The real time that elapses between successive resynchronizations of process j is $(P-\alpha+\beta)(1 + \rho)$. Thus, for any execution of the modified algorithm, there exists a constant c (that depends on t_0), such that for all correct processes i, all $k \geq 1$ and $t \in [last^k, last^{k+1}]$,

$$\frac{P}{P-\alpha+\beta}(1 + \rho)^{-1}t+c \leq C_j^k(t)$$

Since for $t \in [end^k, last^k]$ $C_j^k(t) \geq C_j^{k-1}(t)$, the above inequality also holds for $t \in [end^k, end^{k+1}]$. □

By Lemmas 5.9' and 5.10', in any execution of the algorithm, for $k \geq 1$ and for $t \in [end^k, end^{k+1}]$, the logical clock of any correct process i is within the envelope

$$\mu(1 + \rho)^{-1}t+c \leq C_i^k(t) \leq \mu(1 + \rho)t+d$$

where $\mu = \dfrac{P}{P-\alpha+\beta}$, c and d are constants depending on the initial conditions of this execution. Therefore,

$$(1 + \rho)^{-1}t + c/\mu \leq C_i(t)/\mu \leq (1 + \rho)t + d/\mu$$

Hence, if correct processes slow down their logical clocks by this factor of μ, i.e., process i uses $L_i(t) = C_i(t)/\mu$ as its logical time, *optimal accuracy* is achieved. Also, since $\mu > 1$, *agreement* is still guaranteed. Process i continues to use C_i for the synchronization algorithm.

Theorem 5.13: With the modification described above, the algorithm of Figure 5.1 achieves *optimal accuracy.*

Proof: Follows from the above discussion. □

5.5. Bounds on faults tolerated

We now consider the maximum number of faults that can be overcome by a synchronization algorithm that achieves *optimal accuracy.*

Theorem 5.14: Any synchronization algorithm that achieves *optimal accuracy* must have a majority of correct clocks.

Proof: Assume that there exists a synchronization algorithm that achieves *optimal accuracy* for systems with $n \leq 2f$. We show that this is impossible by first considering a system with two processors p_1 and p_2, one of which can be faulty (i.e., $n = 2$ and $f = 1$).

Since the algorithm achieves *optimal accuracy*, in any execution of the algorithm, the logical clock of correct process i satisfies the following relation for all $t \geq end^1$:

$$(1 + \rho)^{-1}t + a \leq C_i(t) \leq (1 + \rho)t + b$$

where a and b are constants. Also, since the algorithm achieves *agreement*, there exists a constant $D_{\max}$ such that if p_1 and p_2 are correct, then $|C_1(t)-C_2(t)| \le D_{\max}$ for all $t \ge end^1$.

We now consider three possible executions of the algorithm. In what follows, superscripts correspond to execution numbers. For simplicity, we assume that all physical clocks start at 0 at real time 0. Assume that the initial state of a given process is the same in all executions. That is, a given process starts executing the algorithm at the same reading of its physical clock.

Execution e_1: Both processes are correct. The physical clock of p_1 runs at the maximum rate possible and that of p_2 at the minimum rate possible. That is, $R_1^{(1)}(t)=(1+\rho)t$ and $R_2^{(1)}(t)=(1+\rho)^{-1}t$. The transmission time for each message is exactly d, where $d \le t_{del}/(1+\rho)^2$.

Execution e_2: Process p_1 is correct and the rate of its physical clock is given by $R_1^{(2)}(t)=(1+\rho)^{-1}t$. The clock of p_2 is faulty and runs at $R_2^{(2)}(t)=(1+\rho)^{-3}t$, but p_2 is otherwise correct and follows the algorithm. The transmission time of each message is $d(1+\rho)^2$.

Execution e_3: Process p_2 is correct and its physical clock is given by $R_2^{(3)}(t)=(1+\rho)t$. The clock of p_1 is faulty and runs at $R_1^{(3)}(t)=(1+\rho)^3t$, but p_1 is otherwise correct. All messages now take $d/(1+\rho)^2$ to be delivered.

We see that all three executions are possible. Since *optimal accuracy* is achieved, and since p_1 is correct in e_1, its logical clock satisfies the relation $C_1^{(1)}(t) \le (1+\rho)t+b^{(1)}$. Since $R_1^{(1)}(t)=(1+\rho)t$, we see that $C_1^{(1)}(t) \le R_1^{(1)}(t)+b^{(1)}$. Similarly, in execution e_2, we see that $R_1^{(2)}(t)+a^{(2)} \le C_1^{(2)}(t)$. But the two executions look identical to p_1, and hence the relation between its logical and physical clocks must be the same in both executions. Therefore, to satisfy the two relations above, we see that for $k = 1, 2$,

$$R_1^{(k)}(t) + a^{(2)} \leq C_1^{(k)}(t) \leq R_1^{(k)}(t) + b^{(1)}$$

Therefore, in execution e_1, there exists a time τ such that for all $t \geq \tau$

$$(1 + \rho)t + a^{(2)} \leq C_1^{(1)}(t) \leq (1 + \rho)t + b^{(1)}$$

Similarly, by considering executions e_1 and e_3, in both of which p_2 is correct, we see that there exists a time τ' such that for all $t \geq \tau'$

$$(1 + \rho)^{-1}t + a^{(1)} \leq C_2^{(1)}(t) \leq (1 + \rho)^{-1}t + b^{(3)}$$

From these two relations it follows that in execution e_1, for any given D_{max}, there is some time t' such that for all $t \geq t'$, the deviation between the two correct logical clocks is greater than D_{max}, which violates the *agreement* condition.

This can be generalized to any system of $n \geq 2$ processes, where $n \leq 2f$. Partition the processes into two sets P_1 and P_2, with not more than f processes in either set. By constructing executions similar to those above, we can prove that no synchronization algorithm can achieve *optimal accuracy* if $n \leq 2f$. □

The authenticated algorithm of Figure 5.1 requires $n > 2f$ processes. By Theorem 5.13, this algorithm can be modified to achieve *optimal accuracy*. From Theorem 5.14, it follows that the modified algorithm is also optimal in the number of faults tolerated.

5.6. Synchronization without authentication

5.6.1. Simulating authenticated broadcasts

The synchronization algorithm in Figure 5.1 assumed that messages are authenticated. The proof of correctness and the analysis of the authenticated algorithm rely on the following properties of the message system:

P1. *(Correctness)* If at least $f+1$ correct processes broadcast (*round* k) messages by time t, then every correct process accepts the message by time $t + t_{del}$.

P2. *(Unforgeability)* If no correct process broadcasts a (*round* k) message by time t, then no correct process accepts the message by time t or earlier.

P3. *(Relay)* If a correct process accepts the message (*round* k) at time t, then every correct process does so by time $t + t_{del}$.

As seen earlier, implementing authentication using digital signatures provides these three properties. However, the correctness of the algorithm does not depend on this particular implementation, and any other implementation providing these properties can be used instead. A broadcast primitive to simulate authentication in asynchronous systems is described in Section 4.2. By replacing authenticated broadcasts in the algorithm of Figure 5.1 with this primitive, we get a logically equivalent non-authenticated algorithm having the properties of the authenticated algorithm. However, the number of messages sent by correct processes is $O(n^3)$ per resynchronization.

We now modify this broadcast primitive to achieve the three properties described above at a cost of only $O(n^2)$ messages per resynchronization. The primitive is presented in Figure 5.2, and requires $n \geq 3f+1$. With this primitive, each broadcast now requires two phases of communication. Therefore, t_{del}, the upper bound on the time required for a message to be prepared by a process, sent to all processes and processed by the correct processes accepting it, must be re-evaluated. Let τ be the maximum transmission delay between any two processes. Then, $t_{del} \geq 2\tau$.

Theorem 5.15: The broadcast primitive achieves properties of *correctness, unforgeability* and *relay*. The number of messages sent by correct processes is $O(n^2)$ per resynchronization.

Proof:

(Correctness): Since at least $f+1$ correct processes broadcast (*round k*) by time t, every correct process receives at least $f+1$ (*init, round k*) messages by time $t+\tau$ and sends (*echo, round k*). Hence, by time $t+2\tau$, every correct process receives at least $2f+1$ (*echo, round k*) messages. That is, every correct process accepts

---

To broadcast a (*round k*) message, a correct process sends (*init, round k*) to all.
for each correct process:
  if received (*init, round k*) from at least $f+1$ distinct processes
    → send (*echo, round k*) to all;
  [] received (*echo, round k*) from at least $f+1$ distinct processes
    → send (*echo, round k*) to all;
  fi
  if received (*echo, round k*) from at least $2f+1$ distinct processes
    → accept (*round k*) **fi**

Figure 5.2. A broadcast primitive to achieve properties P1, P2 and P3.

(*round* k) by time $t+t_{del}$.

(Unforgeability): Since no correct process sends an (*init*, *round* k) message by time t, a correct process could have received (*init*, *round* k) messages from at most f processes and (*echo*, *round* k) messages from at most f processes. Thus, no correct process sends an (*echo*, *round* k) message by time t. Hence, no correct process accepts (*round* k) by time t.

(Relay): Since a correct process accepts (*round* k) at time t, it must have received at least $2f+1$ (*echo*, *round* k) messages. Every correct process receives at least $f+1$ of these within another τ and sends an (*echo*, *round* k) if it has not already done so. Hence, by $t+2\tau$ (i.e. by $t+t_{del}$), every correct process accepts a (*round* k) message.

Since each correct process sends at most 2 messages for each resynchronization round (an *init* and an *echo*), the total number of messages sent by correct processes is $O(n^2)$ per round. □

5.6.2. A non-authenticated algorithm for clock synchronization

Replacing signed communication with our broadcast primitive extends the synchronization algorithm of Figure 5.1 to one for systems without authentication. The relay property of the primitive implies that we need not explicitly relay messages since the primitive does this automatically. Since the primitive requires $n>3f$, the non-authenticated algorithm also has this limit on the number of faulty processes. It has been shown in [Dole84] that if authentication is not available, and if there are no bounds on the rate at which faulty processes can generate messages, then synchronization is impossible unless $n>3f$.

As in Section 5.2, we assume that clocks are initially synchronized such that at $ready^1$, all correct processes are using C^0, and these clocks are at most $D_{\max}$ apart. The non-authenticated algorithm is described in Figure 5.3.

Theorem 5.16: The non-authenticated algorithm in Figure 5.3 achieves *agreement* and *accuracy*. Correct processes send $O(n^2)$ messages per resynchronization.

Proof: By the properties of *correctness*, *unforgeability*, and *relay* of the primitive of Figure 5.2, it is easy to see that the proofs of Lemmas 5.1 to 5.10 hold. Therefore, the algorithm in Figure 5.3 achieves *agreement* and *accuracy*. Also, by Theorem 5.15, correct processes send $O(n^2)$ messages for each resynchronization round. □

Thus, the number of messages sent by correct processes for each resynchronization is comparable to that in [Lund84].

In Section 5.4.2, we showed how the authenticated algorithm could be modified to achieve *optimal accuracy*. Translating this modified algorithm with our broadcast primitive results in a non-authenticated algorithm that achieves *optimal*

```
cobegin
  if C^{k-1}(t)=kP                        /* ready to start C^k */
       → broadcast (round k)  fi          /* using the primitive in Figure 5.2 */
//
  if accepted the message (round k)       /* according to the primitive */
       → C^k(t) := kP+α  fi               /* start C^k */
coend
```

Figure 5.3. A non-authenticated algorithm for clock synchronization for process p for round k.

---

accuracy.

5.7. Initialization and integration

The algorithms presented in the previous sections can be used, with simple modifications, to achieve initial synchronization and to integrate new processes into the network.

Here we show how processes start their 0^{th} clocks close to each other. A process decides, independently, that it is time to start clock C^0 and broadcasts a *round* 0 message. On accepting a (*round* 0) message at real time t, it starts C^0 by setting $C^0(t)=\alpha$. The number of processes required, and the rules for accepting messages are as described in Sections 5.3 and 5.6, for the authenticated and non-authenticated systems, respectively. Since the authenticated and non-authenticated algorithms are equivalent, we illustrate only the non-authenticated version here (Figure 5.4).

It is easy to see that all processes start C^0 within t_{del} of each other. Also no correct process starts C^0 until at least one correct process is ready to do so. Once they have started C^0, processes run the resynchronization algorithm. At $ready^1$, which by definition is the time when a correct process first sends a (*round* 1) mes-

---

broadcast (*round* 0);	/* using the primitive in Figure 5.2 */
if accepted the message (*round* 0) $\rightarrow C^0(t) := \alpha$ **fi**	/* according to the primitive */ /* start C^0 */

Figure 5.4. A non-authenticated algorithm for achieving initial synchronization.

---

sage, every correct logical clock reads P or less. That is, every correct process is using C^0. By proofs similar to those in Lemmas 5.2 and 5.6, it can be seen that at *ready*^{1}, correct clocks are no more than $D_{\max}$ apart. Thus, this algorithm justifies assumptions S1 and S2 for $k=1$ in the proof of Lemma 5.8.

We now describe how a process joins a system of synchronized clocks. This could be used by new processes to enter the system, or by processes which have become unsynchronized (possibly due to failures) to re-establish synchronization with the rest of the system. The algorithms are based on the idea in [Lund84], modified to the context of our algorithms.

When a process p wishes to join the system, it sends a message (*joining*) to the processes already in the system. It then receives messages from these processes and determines the number i of the round being executed. Since p could have started this algorithm in the middle of a resynchronization period, it waits for resynchronization period $i+1$ and starts its logical clock C^{i+1} when it accepts a (*round* $i+1$) message. It is easy to prove that its clock is now synchronized with respect to the clocks already in the system. Process p now begins to run the resynchronization algorithm described earlier. We present only the non-authenticated version in Figure 5.5. This algorithm can also be modified as described in Section 5.4.2 to ensure that *optimal accuracy* is achieved.

This integration scheme prevents a (possibly faulty) process joining the system from affecting the correct processes already in the system. Hence, we prefer this "passive" scheme to that presented in [Halp84].

send (*joining*) to all processes;
accept a (*round* i) message for some i;

if accepted the message (*round* $i+1$) /* wait for round $i+1$ */
  $\rightarrow C^{i+1}(t) := (i+1)P+\alpha$ **fi** /* start C^{i+1} */

Figure 5.5. A non-authenticated algorithm used by a process to join the system.

5.8. Restricted models of failure

In the preceding sections, we have assumed that faulty processes can exhibit arbitrary behavior. Fault-tolerant algorithms have also been studied under simpler, more restrictive models of failure. It is likely that in certain applications, faults are not as arbitrary as we have assumed so far. In such cases, developing algorithms for the simpler model of failure could result in easier and less expensive solutions.

The most benign type of failure is that of *crash* faults, where processes fail by just stopping [Lamp82, Hadz84]. Less restrictive models are *omission*, where faulty processes occasionally fail to send messages [Hadz84], or *sr-omission*, where faulty processes fail to send or receive messages [Perr84]. In this section, we show how the algorithms developed so far can be adapted to these models.

The algorithm of Figure 5.1 was shown to overcome arbitrary failures. The proof relied on an authenticated message system providing the properties of *correctness*, *unforgeability*, and *relay*. Consider systems with sr-omission failures, where a process is faulty either because it occasionally fails to send or receive messages, or because its physical clock does not satisfy assumption A1. For such systems, we can achieve these three properties without requiring signatures, using the broadcast primitive of Figure 5.6. With this broadcast primitive, the algorithm of Figure 5.3 is a synchronization algorithm for systems with sr-omission faults. Since crash faults and omission faults are a proper subset of sr-omission faults, the algorithm of Figure 5.3 can also tolerate these faults. As explained in Section 5.4.2, this algorithm is easily modified to achieve *optimal accuracy.* The primitive in Figure 5.6 requires $n>2f$ processes and $t_{del}=\tau$. In contrast, the primitive of Figure 5.2 requires $n>3f$ processes and $t_{del}=2\tau$, but it overcomes arbitrary failures.

The lower bound proofs of Theorem 5.12 and Theorem 5.14 do not make any assumptions on the behavior of faulty processes. In fact, we only required that the clocks of faulty processes run at arbitrary rates with respect to real time. Therefore, both lower bounds hold even for crash faults. Thus, our synchronization algorithm is optimal in the number of faults that can be tolerated for all the models of failure we consider.

To broadcast a (*round* k) message, a correct process sends (*init*, *round* k) to all.
for each correct process:
  if received (*init*, *round* k) from at least $f+1$ distinct processes
    → accept (*round* k);
      send (*echo*, *round* k) to all;
  [] received (*echo*, *round* k) from any process
    → accept (*round* k);
      send (*echo*, *round* k) to all;
  fi

Figure 5.6. A broadcast primitive to achieve properties P1, P2 and P3 for a system with sr-omission failures.

Initial synchronization and integration of new clocks are achieved as in previous sections.

5.9. Discussion

The requirements of synchronization can also be stated as follows [Halp84, Dole84]: there exist constants $d_{\min}, P, D_{\max}$ and ADJ, such that clocks are resynchronized at logical times that are multiples of P, and for all correct clocks i and j and for all $k \geq 1$:

C1. $\forall t \in [end^k, end^{k+1}]$

$$\left| C_i^k(t) - C_j^k(t) \right| \leq D_{\max}$$

C2. If C_i^k is started at time t, then

$$0 \leq C_i^k(t) - C_i^{k-1}(t) \leq ADJ$$

C3. $0 \leq end^k - beg^k \leq d_{\min}$

These conditions assert that the maximum deviation between correct clocks is bounded, the amount by which clocks are re-adjusted is bounded, and the size of a resynchronization period is small. Our algorithms satisfy these conditions. Lemmas 5.1 and 5.6 show that conditions C1 and C3 are satisfied. From Lemma 5.4, we see that clocks are never set back. It is easy to show that the maximum adjustment made is $\alpha + D_{\max}$. Hence, by setting $ADJ = \alpha + D_{\max}$, condition C2 is also met.

A feature of our algorithm is that $d_{\min}, P$, and ADJ depend only on the system parameters ρ and t_{del}, and on the constraint $D_{\max}$. In the authenticated algorithm in [Halp84], the adjustment ADJ is proportional to the number of faulty processors. Our solution does not use averaging, and for the non-authenticated case, given

$D_{\max}$, the maximum permitted deviation between correct clocks, our algorithm needs about half as many resynchronizations as in the best previous result [Lund84]. The minimum value of $D_{\max}$ that our algorithm can achieve depends only on ρ and t_{del}. In [Lamp85], the minimum $D_{\max}$ possible is proportional to the number of processes in the system.

In the preceding sections, we have assumed a completely connected network. This assumption can be relaxed using well-known techniques. For an authenticated system, node connectivity of $f+1$ is sufficient. This ensures that there is at least one fault-free path between every pair of correct processes. As in [Halp84], by defining t_{del} to be the maximum time to transmit a message between correct processes along at least one fault-free path in the network, the results of Section 5.3 hold.

Similarly, a non-authenticated system with node connectivity of $2f+1$ provides at least $f+1$ distinct fault-free paths between each pair of correct processes. Define t_{del} to be twice the maximum time taken for a message to be relayed along $f+1$ fault-free paths. Again, the results proved earlier for the non-authenticated system hold.

5.10. Summary

In this chapter, we studied a simple, efficient and unified solution to the problems of synchronizing clocks, initializing these clocks, and integrating new clocks, in distributed systems. The solution can be applied to systems with different types of failures: crash, omission, and arbitrary failures, with and without message authentication.

This is the first known solution that achieves *optimal accuracy*, i.e., the accuracy of synchronized clocks (with respect to real time) is as good as that specified for the underlying hardware clocks. The algorithms presented are also optimal with respect to the number of faulty processes that can be tolerated to achieve this accuracy.

CHAPTER 6

Conclusions

Non-authenticated algorithms for systems with arbitrary failures are generally regarded as complicated, unintuitive, and difficult to derive and prove correct. On the other hand, authenticated algorithms are usually much simpler and easier to derive. However, there are disadvantages to known cryptographic implementations of authentication. Furthermore, developing authenticated solutions did not generally help in deriving non-authenticated solutions.

To take advantage of the simplicity of authenticated algorithms without incurring the disadvantages of digital signatures, we developed broadcast primitives that achieve properties of authenticated broadcasts and certain properties beyond those provided by authentication. Using these broadcast primitives, we presented a methodology for deriving non-authenticated algorithms. Informally, one starts by developing an authenticated algorithm. Replacing authenticated communication in this algorithm with the primitive gives an equivalent non-authenticated algorithm.

We first applied this approach to synchronous systems and derived simple and efficient solutions to the problems of Byzantine Agreement, early stopping Byzantine Agreement, and Byzantine Elections. We then considered asynchronous systems and derived a broadcast primitive that provides properties of authentication in such systems.

The approach described above was then applied to the problem of maintaining synchronized logical clocks in a distributed system. We derived the first known synchronization algorithm that achieves optimal accuracy. This solution can be applied to systems with failures ranging from simply halting to arbitrary behavior.

The results in this thesis can be extended in two directions. We have discussed techniques for translating algorithms that assume authentication to algorithms that overcome arbitrary failures without using signatures. Hadzilacos [Hadz83a] has shown that omission failures can be effectively reduced to crash faults. What is required are mechanisms to reduce malicious failures with authentication to omission failures, possibly through reductions to intermediate models of failure. With communication primitives achieving these reductions, we would have a sequence of mechanisms that span the entire range of failures. This would greatly simplify and unify the design of fault-tolerant algorithms.

Another extension is the investigation of more efficient translation techniques. In this thesis, each of the algorithms that used a broadcast primitive was derived and proved correct on the basis of certain properties. The properties, and therefore the algorithms and their proofs, are independent of the particular implementation of the primitive used in that algorithm. More efficient or simpler implementations than the ones proposed here may exist.

References

Brac85 G. Bracha and S. Toueg, Asynchronous consensus and broadcast protocols, *Journal of the ACM,* vol. 32, no. 4, October 1985.

Coan85a B.A. Coan, D. Dolev, C. Dwork and L. Stockmeyer, The distributed firing squad problem, *Proc. 17th ACM Symposium on Theory of Computation,* Providence, Rhode Island, pp. 335-345, May 1985.

Coan85b B. Coan, A communication-efficient canonical form for fault-tolerant distributed protocols, Manuscript in preparation.

Cris84 F. Cristian, H. Aghili, R. Strong and D. Dolev, Atomic broadcasts: From simple message diffusion to Byzantine Agreement, Tech. Rep. RJ4540, IBM Research Laboratory, San Jose, California, December 1984.

Diff76 W. Diffie and M. Hellman, New directions in cryptography, *IEEE Transactions on Inf. Theory,* vol. IT-22, pp. 644-654, 1976.

Dole82a D. Dolev and H. R. Strong, Requirements for agreement in a distributed system, Tech. Rep. RJ 3418, IBM Research Laboratory, San Jose, California, March 1982.

Dole82b D. Dolev, M.J. Fischer, R. Fowler, N.A. Lynch, H.R. Strong, An efficient Algorithm for Byzantine without authentication, *Information and Control,* 52(1982), pp. 257-274.

Dole82c D. Dolev, The Byzantine Generals Strike again, *Journal of Algorithms* 3,1 (1982), pp. 14-30.

Dole82d D. Dolev, R Reischuck, and H. R. Strong, 'Eventual' is earlier than 'immediate', *Proc. 23rd Symposium on Foundations of Computer Science,* Chicago, Illinois, pp. 196-203, November 1982.

Dole83 D. Dolev and H. R. Strong, Authenticated algorithms for Byzantine Agreement, *SIAM J. Comput.,* vol. 12, no. 4, pp. 656-666, Nov. 1983.

Dole84 D. Dolev, J.Y. Halpern, and R. Strong, On the possibility and impossibility of achieving clock synchronization, *Proc. 16th ACM Symposium on Theory of Computation,* Washington D.C., pp. 504-511, April 1984.

Fisc83 M.J. Fischer, The consensus problem in unreliable distributed systems (A Brief Survey), YALEU/DCS/RR-273, June 1983.

Fisc85 M.J. Fischer, N.A. Lynch and M.S. Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM,* vol. 32, no. 2, pp. 374-382, April 1985.

Garc84 H. Garcia-Molina, F. Pittelli, and S. Davidson, Applications of Byzantine Agreement in database systems, Tech. Rep. TR 316, Princeton University, June 1984.

Hadz83a V. Hadzilacos, Byzantine Agreement under restricted types of failures (not telling the truth is different from telling lies), Tech. Rep. 19-83, Aiken Computation Laboratory, Harvard University, June 1983.

Hadz83b V. Hadzilacos, A lower bound for Byzantine Agreement with fail-stop processors, Tech. Rep. 21-83, Aiken Computation Laboratory, Harvard University, June 1983.

Halp84 J.Y. Halpern, B. Simons, R. Strong, and D. Dolev, Fault-tolerant clock synchronization, *Proc. 3rd Symposium on Principles of Distributed Computing,* Vancouver, Canada, pp. 89-102, Aug. 1984.

Lamp82a L. Lamport, R. Shostak, and M. Pease, The Byzantine Generals problem, *ACM Transactions on Programming Languages and Systems 4,* pp. 382-401, 1982.

Lamp82b L. Lamport and M. Fischer, Byzantine Generals and Transaction Commit protocols, Opus 62, SRI International, April 1982.

Lamp85 L. Lamport and P.M. Melliar-Smith, Synchronizing clocks in the presence of faults, *Journal of the ACM,* vol. 32, No. 1, pp. 52-78, Jan. 1985.

Lund84 J. Lundelius and N. Lynch, A new fault-tolerant algorithm for clock synchronization, *Proc. 3rd Symposium on Principles of Distributed Computing,* Vancouver, Canada, pp. 75-88, Aug. 1984.

Maha85 S.R. Mahaney and F.B. Schneider, Inexact agreement: Accuracy, precision and graceful degradation, *Proc. 4th Symposium on the Principles of Distributed Computing,* Minaki, Canada, pp. 237-249, Aug. 1985.

Merr84 M. Merritt, Elections in the presence of faults, *Proc. 3rd Symposium on the Principles of Distributed Computing,* Vancouver, Canada, pp. 134-142, Aug. 1984.

Moha83 C. Mohan, H.R. Strong, and S. Filkenstein, Method for distributed transaction commit and recovery using Byzantine Agreement within clusters of processors, *Proc. 2nd Symposium on Principles of Distributed Comput-*

ing, Montreal, Canada, pp.89-103, Aug. 1983.

Peas80 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.

Perr84 K. J. Perry and S. Toueg, Distributed Agreement in the Presence of Processor and Communication Faults, Tech. Rep. 84-610, Department of Computer Science, Cornell University, Ithaca, New York, May 1984. To appear in the *IEEE Transactions on Software Engineering.*

Perr84b K. J. Perry and S. Toueg, An authenticated Byzantine Generals algorithm with early stopping, Tech. Rep. 84-620, Department of Computer Science, Cornell University, Ithaca, New York, June 1984.

Perr85 K. J. Perry, Early stopping protocols for fault-tolerant distributed agreement, Ph.D. thesis, Cornell University, Jan. 1985.

Rabi83 M. Rabin, Randomized Byzantine generals, *Proc. 24th Symposium on Foundations of Computer Science,* Tucson, Arizona, pp. 403-409, Nov. 1983.

Rive78 R.L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM, 21,* 2, pp. 120-126, Feb. 1978.

Tane81 A. S. Tanenbaum, Computer Networks, Prentice-Hall software series, 1981.

Toue84a S. Toueg, K.J. Perry and T.K. Srikanth, Fast Distributed Agreement, *Proc. 4th Symposium on the Principles of Distributed Computing,* Minaki, Canada, Aug. 1985.

Toue84b S. Toueg, Randomized asynchronous Byzantine Agreements, *Proc. 3rd Symposium on Principles of Distributed Computing,* Vancouver, Canada, Aug. 1984.

Turp84 R. Turpin and B. Coan, Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement, *Information Processing Letters,* vol. 18, pp. 73-76, February 1984.