Impact of Communication Networks on Fault-Tolerant Distributed Computing

Rogério Drummond
Ph. D. Thesis

TR 86-748
April 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

IMPACT OF COMMUNICATION NETWORKS
ON
FAULT-TOLERANT DISTRIBUTED COMPUTING

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

Rogério Drummond

June 1986

IMPACT OF COMMUNICATION NETWORKS

ON

FAULT-TOLERANT DISTRIBUTED COMPUTING

Rogério Drummond, Ph.D.

Cornell University 1986

When the desired reliability of a computing system exceeds that of its individual hardware components the need for fault-tolerant systems arise. While distributed systems have the potential to achieve highly reliable computing, programming them is a challenging task. Several paradigms have been identified that can simplify the conceptual design of fault-tolerant distributed computing systems.

Properties of a distributed system have profound implications on the solvability and efficiency of implementations of these paradigms. In this thesis we study the effect that different communication models have on the efficiency of fault-tolerant computing. As an instance of a fundamental operation we examine protocols for *reliable broadcast* in distributed systems. Our main contribution is the characterization of the time complexity of reliable broadcast with respect to communication models. A practical consequence of our results is the development of efficient reliable broadcast protocols in systems where communication between processors is through multicasts. A variety of common networks are shown to support this style of communication. In fact, by parameterizing the minimum multicast size and the diameter of these networks, we are able to characterize all known network architectures.

Distributed systems where processors perceive the same approximate time makes programming them much easier. *Clock synchronization* protocols implement this abstraction given only clocks that have bounded drift rates with respect to real time. We show how a primitive which is normally used only for communication in a distributed system can also be used for synchronizing clocks. If this primitive occurs naturally with a sufficient frequency, clock synchronization can be achieved at no additional message cost.

Our results reveal hardware/software tradeoffs between performance, resiliency and network cost. Thus, they offer many new alternatives previously not considered in designing fault-tolerant systems.

BIOGRAPHICAL SKETCH

Rogério Drummond Burnier Pessôa de Mello Filho was born on July 29^{th}, 1955 in Campinas, Brazil. He is the eldest of four boys. His first notable achievement was a "philosophical" one. At the age of five, disliking the title "primogeniture", he stated, "primogeniture is God." His second achievement was to learn how to write his own name. After this large and arduous endeavor he received the B.Sc. and M.Sc. degrees in Computer Science from the University of Campinas, Brazil in 1978 and 1980 respectively. His interest in academia and teaching sprouted very early. As an undergraduate he was always engaged in extracurricular activities. He worked as an undergraduate teaching assistant and was the recepient of an undergraduate research scholarship for two years. He joined the Department of Computer Science at University of Campinas in 1980 and in the Spring of 1981 he enrolled in the graduate program in Computer Science at Cornell University.

ACKNOWLEDGMENTS

Some moments in our lives seem eternal, whether in time, or in depth. The years I spent in Ithaca were filled with such moments. It is not within these limited lines (of "Rog's English" as my friends say) that I will be able to do justice to all of them.

Özalp Babaoglu has been the perfect research partner — firstly he taught me how. I thank him for being such a thorough advisor. I could not have hoped for anyone better, nor could I have completed this work without his guidance. He has also been present and solidarious in the moments I needed him. That is the kind of friend we all search for.

For Sam Toueg, doing research is, above all, a fun activity. Despite his being a "point-to-point network guy," I thank him for our many discussions which were always very inspiring. He was one of my minor advisors and provided helpful comments on this thesis.

My growing interest in the cognitive development area found solid resonance in Barbara Lust's enthusiasm and competence in this field. As a result of our interaction, I added some technical background to support my interest on the subject. She was my other minor advisor.

Gil Neiger has been a "companheiro" in many ways. I thank him for the many discussions we had in the early stages of this work. He, Hilary and Pat were also kind enough to read and improve the drafts of what became this thesis. Fred Schneider's comments on early drafts of the material on the RB problem improved substantially its presentation. Raimund Seidel kindly showed me how much I didn't know about graph theory. Our conversations were very helpful and prevented me from reinventing the wheel. Rick Palmer and Kate MacGregor provided a Macintosh during the final two month of thesis rush. Preparing for the qualifying exam with Solveig Torgersen was almost fun; together we were able to raise each other's spirits and work efficiently. Geri and Joanna were for me always much more than departmental secretaries.

From my point of view I shall leave Ithaca taking (in my heart) more than I leave behind through this thesis. I have no words to express the solidarity that my friends had shown to me in my last days of thesis rush. These five years were very important for me and many people share the responsibility of making them so meaningful: Aliza, Amr, Andrez, Bia, Gil, Hilary, Ingrid, Irene, Isabela, Kate, Luiz Fernando, Jim, Penny, Raimund, Rick, Sally, Solveig, Sonia, Tommy, and the CUSLAR crowd.

Finally, I want to thank the financial support provided by the Universidade de Campinas, CAPES (agency of the Government of Brazil), and the Fundação de Amparo à Pesquiza do Estado de São Paulo.

TABLE OF CONTENTS

List of Tables

LIST OF FIGURES

CHAPTER 1

Introduction

In this thesis we study the influence of communication networks on the performance of some fundamental fault-tolerant operations in distributed systems. In particular, we study solutions to the Reliable Broadcast and Clock Synchronization problems. The main contribution of our work is to formally characterize the performance of solutions to these problems with respect to the model of communication. We show that many common networks support a model of communication that achieves efficiency of solutions previously thought not possible for these problems. In this chapter, we describe the system under investigation, define the problems of interest, introduce terminology and outline the rest of the thesis.

1.1. Fault-Tolerant Distributed Systems

Continued and correct operation are desirable attributes of any system. They are particularly important for computer systems engaged in "process control" [Sp84] and "database applications" [Ki84, Sk82]. Unfortunately, even the most reliable hardware is prone to failures. If the reliability expected from a system is higher than the reliability of its individual hardware components, then the need for fault tolerance techniques arises. The key idea behind fault tolerance is that of replicating the system's components. However, given a finite amount of hardware, even a fault-tolerant system has a finite set of failure scenarios under which it operates correctly. Consequently, continued and correct operation of a fault-tolerant system is probabilistic. The correctness of the system depends on the probability that actual failures conform to the tolerable failure scenarios. Fault-tolerant systems that rely on replication are only justified if the reliability they provide is greater than the reliability of a single, non-replicated component. Babaoglu shows that given a stochastic model of processor failure times, the replicated structures we propose for fault tolerance indeed provide higher reliability than their non-replicated counterparts [Ba86].

Distributed computing systems provide a convenient environment for the development of fault-tolerant systems. Informally, a *distributed system* is a collection of autonomous processors that share neither memory nor a common clock and communicate only through a network. Such a system can tolerate failures if the programs to run on it are replicated and executed by several processors. These replicas must cooperate, thus requiring distributed "coordination" algorithms.

The design of fault-tolerant distributed algorithms is difficult for several reasons. The lack of shared memory prevents the processors from realizing a common view of the current system state. Random communication delays and the lack of a common clock limit the processors' ability to synchronize their actions. In addition to these difficulties, the algorithms have to tolerate unforeseen failures of processors and communications network components. These failures are either undetectable or at best are only detected after some processor has already taken actions based on incorrect information it has received.

1.2. Failure Modes

We say that a processor is *non-faulty* if its behavior conforms to an abstract specification (typically in the form of a protocol). Any deviation from this specification is called a *failure*. A processor is said to be *faulty* after its first failure. Note that during the execution of a protocol a faulty processor can exhibit many failures.

In a distributed system, the only externally visible behavior of a processor is the sequence of messages it sends. We consider the following failure modes of processors (with respect to a given protocol):

Crash Failure. A faulty processor stops sending messages after some arbitrary time, never to resume again.

Omission Failure. A faulty processor omits sending some of the messages prescribed by its protocol. The messages it does send are always as prescribed.

Malicious Failure. A faulty processor exhibits arbitrary behavior. In particular, such a processor can withhold messages that it is required to send, can send messages not prescribed by its protocol, and can even collude with other (faulty) processors to disrupt the system.

A non-faulty processor sends exactly those messages that are prescribed by its protocol — no more and no less. Clearly, a protocol that can tolerate malicious failures can also tolerate either omission or crash failures. Similarly, a protocol that can tolerate omission failures will tolerate crash failures.

We also consider failures of the components of the communication network. These shall also be classified as crash, omission or malicious failures with the above definitions applied to communication components.

1.3. Restricting the Influence of Faults

Arbitrary processor failures in a distributed system using a point-to-point communication network are troublesome for several reasons. First, a faulty processor can send conflicting information along disjoint routes to other processors. Secondly, unless the communication graph is fully connected, faulty processors on the route of a message can tamper with it. Finally, faulty processors can generate spurious messages and pretend to be forwarding them on behalf of other processors.

The externally visible behavior that faulty processors may exhibit can be effectively restricted through the use of certain mechanisms. For example, if the system supports an authentication mechanism such as digital signatures [Di76], faulty processors are prevented from altering messages and from spontaneously generating spurious messages. This restriction on the behavior of a faulty processor results in algorithms that are simpler and that can tolerate more faults than their counterparts without authentication [Do83b]. Srikanth and Toueg have recently devised a communication primitive that achieves certain properties of authentication in the absence of digital signatures [Sr85c]. In what follows, we propose communication architectures as mechanisms to further restrict the behavior of faulty processors.

Our motivation is based on the following observation. If we can design a communication network architecture such that

(i) there exists a single route between any given processor and all other processors, and

(ii) messages never need to be forwarded (i.e. all processors are effectively immediate neighbors of each other),

then the behavior of a faulty processor can be limited. In particular, designs based on such architectures need not worry about message tampering and confounding of processors through conflicting messages. Consequently, a malicious processor can only lie about its internal state by sending incorrect messages or by not sending a message at all. Any other faulty behavior can be detected by non-faulty processors.

Broadcast networks, such as ALOHA [Ab73], DCS [Fa73] and Ethernet [Me76] come very close to realizing the above properties. In Chapter 2 we introduce communication structures called *R-redundant broadcast networks* and characterize their properties. These networks severely limit the effect of faulty processors and allow great simplification of the design of fault-tolerant distributed algorithms.

1.4. The Reliable Broadcast Problem

Recall that a distributed computing system consists of a set of autonomous processors that can communicate through a network. To cooperate, these processors must periodically coordinate their activities. One way to do this is through *broadcasts* — an operation that allows a processor to communicate a local value to all other processors in the system. Since, in the systems we consider, both processors and communication components can fail, the semantics of a broadcast operation have to accommodate failures. We define the *reliable broadcast* problem in an unreliable environment as follows.

> **Reliable Broadcast (RB).** A distinguished processor, called the *transmitter*, broadcasts a local value (drawn from a domain of at least two elements) such that the following requirements are satisfied:
>
> > *Agreement.* All non-faulty processors accept the same value.
> >
> > *Non-triviality.* If the transmitter is non-faulty, then all non-faulty processors accept the value that was broadcast by the transmitter.
> >
> > *Simultaneity.* All non-faulty processors accept a value at the same time.

We distinguish between a processor *receiving* a message and *accepting* a message. "Receive" is the general communication primitive provided by the network, whereas "accept" is an operation implemented at the protocol level. Therefore, it is possible for a processor to receive some broadcast message but not to accept it.

The abstract formulation of this problem was first studied by Pease *et al.* [Pe80] under the name *Interactive Consistency* (later called *Byzantine Agreement* [La82b]) and subsequently studied by numerous other researchers (see [Fi83a] for an early survey).

The reliable broadcast has been identified as the appropriate primitive in solutions to a large number of problems. These include distributed consensus [Fi83a], distributed synchronization [Sc82], replicated data update (e.g., distributing updates to routing tables in a packet switching network) [Ga84], and transaction management in database systems [Mo83]. For certain applications, non-faulty processors may be required to accept messages in the same order. The resulting primitive is called an *atomic broadcast* [Cr84] and can be implemented on top of a reliable broadcast protocol by using timestamps.

Given a reliable broadcast protocol, it becomes simple to convert any application into an equivalent fault-tolerant application: The program and data associated with the application are replicated at several processors, and whenever one of the processors modifies its copy of the data, it broadcasts this information to all other processors through the RB protocol [La84a, Sc84]. The fault-tolerant version of the application derived using this technique can exhibit reasonable performance only if the RB protocol implementation is efficient.

Properties of the environment in which RB is formulated have profound implications on the nature of the solution. In this thesis, we will be concerned only with deterministic protocols in synchronous systems. We define such systems in Section 1.5.

Most of the previous formulations of the RB problem assume a reliable, point-to-point network as the communication architecture. In such a system, let F denote the maximum number of processors that can be faulty during the execution of a protocol solving RB. Lower bounds for the total number of processors participating in the protocol [La82b], time complexity of the protocol [Fi82], and message complexity [Do82a] have been obtained as functions of F. It is customary to measure time complexity in units called *rounds* consisting of message exchange and computation steps, such that during a round every processor can communicate with all others in the system. The messages sent by a processor in a given round cannot depend upon the messages it receives during the same round. A particularly important result in this formulation is that at least F+1 rounds are required by any solution to the RB under even the most restrictive failure modes and unconstrained amounts of information exchange [Fi82, Do83b, Ha84a]. We note that if simultaneity is not required, than there are solutions to the RB problem that may stop earlier than F+1 rounds [Do82b]. However, in the worst case scenarios these protocols also require F+1 rounds.

In Chapter 3, we dispel the belief that RB is an inherently expensive primitive by breaking the F+1 round barrier. We show that in R-redundant broadcast networks, RB can be implemented in two rounds. With respect to the RB problem, a system using an R-redundant broadcast network can tolerate more failures than one using a point-to-point network. In a point-to-point network system without authentication, the RB problem is solvable only if the total number of faulty processors is less than a third of the total number of processors. In contrast, R-redundant broadcast network systems can tolerate a number of faulty processors that is almost half the total number of processors under the same failure assumptions. We argue that our assumptions about the behavior of the communication network are reasonable in the environments where RB is applicable as a primitive in the first place — highly integrated distributed systems built around local area networks. We note that other researchers have also proposed using broadcast networks to implement reliable broadcast, however under much stricter failure assumptions [Ch84, Le81].

In Chapter 4, we study the effect of communication network properties on the execution time required to solve the RB problem when processor failures are restricted to omission faults. We consider the class of network structures characterized by their diameter and ability to support partial broadcasts. This class includes fully-connected point-to-point networks, linear chains, rings, broadcast channels and all other familiar communication structures. We derive a single protocol that implements RB for any member within this class. This protocol has been shown to be optimal with respect to its execution time as measured in rounds [Ba85b]. The tradeoffs that are revealed between performance, resiliency and network cost offer many new alternatives previously not considered in designing fault-tolerant systems.

1.5. Models of Systems and Protocols

A protocol that permits non-deterministic steps during its computation is called a *randomized protocol*. In contrast, *deterministic protocols* do not permit non-deterministic steps and, consequently, their input alone determines a unique execution sequence.

We say that a distributed protocol exhibits *deterministic termination* if its execution time can be bounded. For most applications this is a desirable, if not an essential, property. In *asynchronous distributed systems,* where message delivery times and relative processor speeds cannot be bounded, it is impossible to implement any protocol that relies upon message exchange between processors and also exhibits deterministic termination. Unfortunately, most interesting protocols rely on message exchange. Fischer *et al.* have shown that the *distributed consensus problem*[1] has no deterministic solution in systems where failures cannot be detected and message delivery times cannot be bounded even if the communication media is perfectly reliable and processor failures are restricted to crash faults [Fi83b]. In such a system, the RB problem has no solution at all, whether deterministic or randomized [Br85]. If the transmitter simply fails before sending any message, then the remaining processors can never know whether the transmitter has failed or whether its message has not yet arrived. We note that there are randomized protocols that solve the distributed consensus problem in asynchronous environments [Br85]. These solutions, however, do not (and can not) exhibit deterministic termination.

A distributed system is said to be *synchronous* if the message delivery times and the relative speed of processors are bounded. These systems not only allow deterministic

[1] This problem is equivalent to solving n concurrent RB problems where each processor plays the role of the transmitter.

termination protocols but are also much simpler to program. All the systems considered in this thesis are synchronous; the protocols are deterministic and exhibit deterministic termination.

1.6. Communication Complexity

From a practical point of view, we must be careful in comparing two different protocols based on their time complexity as measured in rounds. While the round concept clearly and formally abstracts the information exchange and the logical ordering needed to reason about distributed protocols, it hides the communication effort required for its implementation. The actual time necessary to implement a round may vary substantially from one communication architecture to another.

In Chapter 5, we adopt the end-to-end message delay as the time unit measurement and study the complexity of communication primitives. We identify those primitives that are used in fault-tolerant distributed systems and classify communication networks according to their cost. We then exhibit lower bounds on the time needed to implement each of these communication primitives in the most cost-effective networks. An analysis of the time complexity of RB protocols under this new time complexity measurement concludes the chapter.

1.7. The Clock Synchronization Problem

As defined above, the synchronous property of a system is still not sufficient for many applications. Even in a synchronous system, a fast processor may get arbitrarily ahead of a slow processor after a sufficiently long time. In a synchronous system, bounding the relative speeds of processors is equivalent to the requirement that hardware clocks associated with processors have bounded drift rates. In distributed systems, processors often need to execute actions at about the same time and, therefore, their clock differences must be bounded. Protocols that can achieve this bounded difference in clocks are known as *Clock Synchronization Protocols* [La82a].

In Chapters 3 through 5 we present our results assuming a *perfectly synchronized* system. That is, a system where the local clocks of processors are perfectly synchronized and the message delivery time is the same for all messages. With this assumption we can reason about distributed computations as proceeding in lockstep. Then, in Chapter 6, we show how any such computation can be adapted to synchronous systems. This separation of concerns allows for a much clearer presentation of our protocols and communication primitives. Finally, we present a clock synchronization protocol and analyze its performance in a variety of network architectures. We show that a communication primitive widely used to implement distributed algorithms is also a synchronization tool in the sense that it can be used to synchronize clocks. If this operation is executed frequently enough, the system is kept synchronized at no additional cost in terms of messages.

CHAPTER 2

Models and Assumptions

2.1. Assumptions

We make the following assumptions about the system:

CA: (*Connectivity Assumption*) Despite processor and communication component failures the network remains connected.

SA: (*Synchronism Assumption*) The network guarantees bounded delivery time for all messages and the clocks of non-faulty processors have bounded drift rate with respect to (unobservable) real time.

IA: (*Identification Assumption*) Processors can reliably identify the immediate sender of any message they receive.

CA requires that the network have redundant paths. This assumption is relaxed in Chapter 4 when we deal with network partitions. Assumption SA requires that the systems under consideration be synchronous. The IA assumption is usually achieved by standard network protocols. In the next section we show how IA can be guaranteed when the communication components are unreliable.

2.2. Broadcast Networks

Broadcast networks are a popular topology for local area networks. Some of the reasons for this popularity are simplicity, high reliability, high performance and very low cost. Perhaps the best known example of this class of networks is the Ethernet [Me76]. Figure 2.1 shows a typical distributed system configuration based on a broadcast network.

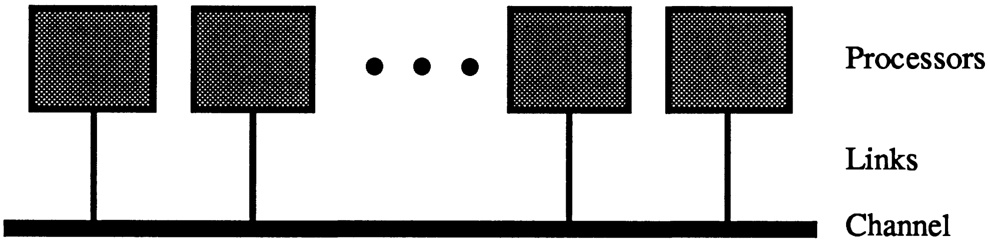


Figure 2.1. Broadcast Network.

The central component of a broadcast network is the *channel.* Typically, this is a passive transmission medium such as a coaxial cable, fiber-optic cable, or simply space (in the case of a radio broadcast network). Processors are connected to the channel through *links.* Links contain active components such as signal transceivers, network interface units, and buffer memory. Access to the channel by the links is controlled through a multiple-access transmission protocol [Ta81]. We note that collision-free transmission protocols exist for multiple-access channels that can guarantee the Synchronism Assumption [Ta81]. Even with transmission protocols where collisions may occur (such as in Ethernet), the message delivery time can be bounded with high probability by keeping the channel lightly loaded.

The fundamental communication operation implemented by a broadcast network is **broadcast**(v), where v is the message being communicated. We abstract the operation of the channel and the links as follows:

> *Non-faulty Channel Semantics.* In the absence of failures, the channel arbitrarily selects one of the links to listen to and simply echoes the incoming message (if any) to all of the other links.
>
> *Faulty Channel Semantics.* A fault in the channel causes some of the incoming messages to be lost (i.e., echoed to none of the other links).

Non-faulty Link Semantics. In the absence of failures, the link simply transfers messages between the channel and its associated processor.

Faulty Link Semantics. A fault in the link causes some arbitrary incoming messages, either from the processor or the channel, to be lost.

We further require that neither a faulty channel nor a faulty link can spontaneously generate (valid) messages. This requirement is usually achieved by standard transmission protocols [Ta81]. Networks that exhibit the above properties can be readily implemented. Even in systems where messages can be altered by a faulty communication component such that some processors receive corrupted messages instead of the original message, the above "omission fault behavior" of the communication media can be simulated as follows. Depending on the nature of the communication media and faults, the messages can be augmented with checksums, cyclic redundancy checks (CRC), or digital signatures based on public-key cryptography techniques [Di76, Ri78]. The receiving processors can then verify each incoming message and, with high probability, detect and discard invalid messages.

The above techniques can also be applied when the communication components are not reliable enough to guarantee the IA assumption. In this case, the identity of the sender of a message can be retrieved from the sender's signature. To simplify our presentation, we will assume that message authentication and validation (if present) are performed at a level below our protocols. Messages that are invalid are simply discarded and are never presented to the protocol level. If no message is delivered to the protocol level after a time interval equal to the network delivery time bound, we say that a null message (denoted as ϕ) is received.

As defined, our channel semantics ensure that either all (in case the channel is non-faulty) or none (in case the channel is faulty) of the processors connected to a given channel through non-faulty links receive the broadcast message. There are rare channel failures that result in the partitioning of the links (and thus processors) such that some of them receive a broadcast message and others do not. These types of failures are addressed in Chapters 3 and 4.

Given that processors communicate through a network with the above properties, we note that there is only one externally visible system behavior for any possible failure scenario.

> *Broadcast Network Property (BNP):* In response to a **broadcast**, those processors that receive a (non-null) message receive the same message.

This behavior is ensured even if processor failures are malicious, since it is physically impossible for a processor to send conflicting messages to different processors in the same broadcast. In other words, we have succeeded in restricting the visible affects of processor failures.

We now discuss the power of this property when applied to solve the RB problem in the presence of processor, link and channel failures.

2.3. Redundant Broadcast Networks

Even with the Broadcast Network Property, the **broadcast**(v) operation alone does not achieve the agreement requirement for RB since a single broadcast network cannot guarantee 1-connectivity[1] among processors in the presence of communication faults.

[1] We say that a network has *k-connectivity* if there exist at least k disjoint paths consisting of links and channels between every pair of processors.

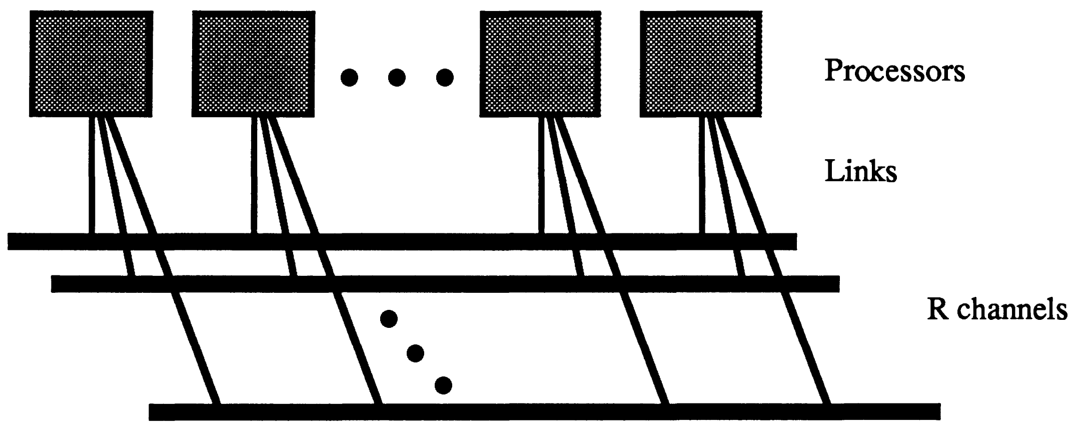


Figure 2.2. R-Redundant Broadcast Network.

To increase connectivity, we propose an *R-Redundant Broadcast Network* generated by replicating each communication component R-fold. As shown in Figure 2.2, the construction requires R independent broadcast networks, with each of the processors connected to each of the R channels through an independent link. In such a network we account separately for the various components that exhibit failures. During some execution of a RB protocol, let

f be the actual number of faulty processors,

λ be the actual number of faulty links, and

γ be the actual number of faulty channels.

Note that f is bounded by F. Let L and G be upper bounds for λ and γ, respectively.

We say that a processor is *properly connected* to a channel if its link to that channel is non-faulty. By construction, it is trivial to show the following property of R-redundant broadcast networks.

Property 2.1. *In an R-redundant broadcast network, the condition* $R>\lambda+\gamma$ *is sufficient to guarantee that any pair of processors will remain at least 1-connected for the duration of the protocol.*

In the subsequent discussions, unless otherwise stated, we will assume that this property holds.

Since every processor in an R-redundant broadcast network is connected to R channels, we augment the broadcast primitive syntax to **broadcast**(v,i); where v is the message to be communicated, and $i \in \{1,2,...,R\}$ selects a channel. We will use the following shorthand for notational convenience:

```
broadcast(v,*)  ≡  foreach i in {1,2,...,R} do
                        broadcast (v, i)
                   od;
```

Note that we do not require this operation to be atomic nor do we make any assumptions about the relative order or timing of the broadcasts over the various channels. Furthermore, no assumptions are made about the behavior of faulty processors with respect to messages broadcast over different channels. In particular, a faulty processor can broadcast conflicting messages over different channels. All that we rely on is that the Broadcast Network Property holds for each of the broadcast networks.

CHAPTER 3

Reliable Broadcast Protocols for Redundant Broadcast Networks

3.1. Introduction

While crash failures are more restrictive than omission failures, it is interesting to note that considering processors and network components that exhibit crash failures does not render the RB problem any easier. In fact, Hadzilacos has shown that for point-to-point networks, there exists a transformation procedure for converting any solution to the RB for crash failures into an equivalent solution (in the sense of having the same number of rounds and messages) for omission failures [Ha84a]. Therefore, we will initially consider processors that fail by omission Then, in Section 3.3 we solve the RB problem for malicious processor failures. In Section 3.4, we allow failures of the communication components that may partition a broadcast channel. Malicious failures of the communication components are considered in Section 3.5. In each case we present a two-round solution for the RB problem. As the failure constraints are relaxed, the protocols require an increase in the redundancy of the network components and processors. Section 3.6 discusses possible optimizations and Section 3.7 concludes the chapter.

3.2. Reliable Broadcast Protocol for Omission Processor Failures

Let p_1, p_2, ..., p_n denote processors and assume that p_1 is the transmitter. Let δ denote a default value known *a priori* to all of the processors such that all of the non-faulty ones will accept δ in case the transmitter is deemed faulty. Let V denote the domain of all possible valid message values not including ϕ. We say that a RB protocol is *symmetric* if the computation carried out by each processor is independent of its name, with the exception of

```
Round 1: For p1 (the transmitter with local value v ∈ V)
        broadcast(v,*);

Round 2: For all pi
        if |Ci^1| ≠ 0 then
                mi is the (unique) message received in round 1
                foreach ki in {1,2,...,R} - Ci^1 do
                        broadcast(mi,ki);
                od;
                accept mi ;
        fi;

At the conclusion of round 2: For all pi
        if not yet accepted then
                if |Ci^2| ≠ 0 then
                        mi is the (unique) message received in round 2
                        accept mi ;
                else
                        accept δ;
                fi;
        fi;
```

Figure 3.1. Protocol P1:
Reliable Broadcast for Omission Failures.

the transmitter which is distinguished from the other processors. We do not consider the message authentication and validation steps (if present), which can be processor name dependent, to be part of the RB protocol. Each processor executes the protocol of Figure 3.1. Note that all variables are subscripted with the processor name only to denote that they are local variables. Consequently, the protocol is symmetric according to our definition. Recall that the values broadcast in a given round cannot depend on messages received in that round. In the protocol, the notation C_i^r denotes the set of channels from which processor p_i receives the (non-null) message $m_i \in V$ at round r. Note that message m_i is unique since processor failures are restricted to omission.

For simplicity, we will present protocols in a form that does not satisfy the simultaneity requirement. A trivial modification will be sufficient to achieve simultaneity. The following theorem establishes the correctness of protocol P1.

Theorem 3.1. *If* $n \geq \lambda + f$, *then protocol P1 implements RB for processors that may fail by omission.*

Proof: We proceed by case analysis.

If the transmitter is non-faulty, all processors receive the broadcast in round 1 (and know that it is from the transmitter) and accept v at round 2. Thus, non-triviality and agreement are assured. The following ensures agreement in cases of transmitter failure.

If the transmitter is faulty and broadcasts no messages over non-faulty channels to which it is properly connected in the first round, by the end of the second round only ϕ messages have been received and all non-faulty processors accept δ.

Next consider the case where the transmitter is faulty and some non-faulty processor p_i receives the initial broadcast. Since processor failures are restricted to omission, the non-null messages m_i received by p_i must all be equal to v. The protocol requires p_i to echo m_i through all channels not in $\mathbf{C}_i{}^1$. Therefore, by the end of the second round, v-messages will have reached all non-faulty channels to which p_i is properly connected. By the 1-connectivity property (Property 2.1), it is guaranteed that all processors receive v in one of the two rounds. As receiving a non-null message in either of the two rounds implies accepting, all processors accept v.

Finally, we consider what happens if the transmitter is faulty and only other faulty processors receive the initial broadcast. This can only occur when all non-faulty processors have faulty links to the non-faulty channels that p_1 broadcast over in the first round. The minimum number of faulty links necessary by the end of round 1 for this scenario is n–f and comes about when p_1 broadcasts over a single non-faulty channel to which it is properly connected and all of the non-faulty processors have faulty links to this channel. Since $n \geq \lambda+f$ by hypothesis and $\lambda=n-f$ by the above argument, there can be no further faulty links. If one of the faulty processors echoes v during round 2 over a channel other than the one p_1 broadcast over, then all non-faulty processors are guaranteed to receive it and accept it. Otherwise, all non-faulty processors receive only ϕ messages by the end of round 2 and accept δ. ∇

From the protocol description, it is obvious that P1 results in at most nR messages from being exchanged and requires two rounds to implement RB. The next result shows the optimality of P1 with respect to time complexity within the class of redundant broadcast networks.

Theorem 3.2. *For the class of R-redundant broadcast networks, there exists no symmetric one-round protocol to implement RB in the presence of even non-faulty channels, a single faulty processor, and a single faulty link.*

Proof: Under any (one-round) protocol, there must be a step where p_1 broadcasts its local value. Assume that p_1 fails by crashing after having broadcast v over k channels ($0<k\leq R$). As before, $\mathbf{C}_i^1$ denotes the set of channels over which processor p_i receives non-null messages from the transmitter by the end of round 1. Note that during the single round, any two processors can also exchange information through broadcasts. However, this information cannot depend on what each one received from the transmitter since this occurs in the same (single) round. Furthermore, since we require that the protocol be deterministic and symmetric, the information content (with respect to RB) of this communication between the processors is zero. Note that each channel could have a different name with respect to different processors. This fact, along with the requirement that the protocol be symmetric, prevents the decision to be based on the actual elements of

the set C_i^1. This leaves the number of non-null messages received by each processor from the transmitter by the end of the round as the only relevant information for accepting. Therefore, for any symmetric one-round protocol, there must exist a common function Ψ such that each processor p_i accepts $\Psi(|C_i^1|)$ at the end of the round. From the non-triviality requirement of RB, we must have $\Psi(R)=v$. Furthermore, $\Psi(0)=\delta$ simply due to complete lack of global information. Consequently, there must exist some threshold value $z>0$ such that $\Psi(z-1)=\delta$ and $\Psi(z)= v$. Now, in the above failure scenario, let $k=z$ whereby all non-faulty processors accept v except for the processor with the single faulty link to one of the k channels, which has to accept δ. We conclude that the agreement requirement for RB cannot be assured under any symmetric one-round protocol. ∇

We next consider the RB problem when processor failures are unconstrained.

3.3. Reliable Broadcast Protocol for Malicious Processor Failures

By far the most important property of a broadcast network is that it ensures agreement among the receivers of a broadcast in the absence of network failures. It is impossible for a (maliciously) faulty processor to send conflicting values to different processors in a single broadcast over the same channel. Our motivation for the construction of R-redundant broadcast networks was to increase connectivity so that network failures could be tolerated in an effective manner. However, in so doing, we have left the system vulnerable to malicious failures whereby conflicting messages can be sent by a faulty processor over each of the R components of the redundant broadcast network during a single **broadcast**(v,*) operation.

In this section, we present an efficient protocol that implements RB in R-redundant broadcast networks in the presence of malicious processor failures. Our assumptions about the behavior of the communication system continue to apply. In terms of fault-tolerant system design, malicious failures represent a "worst case" design strategy. Even in this case, our protocol requires only two rounds to implement RB. The only additional cost with respect to protocol P1 is an increase in the total number of processors that must participate in the protocol for a given level of fault tolerance.

Before discussing the protocol, we introduce some notation and definitions. Let $W=\{\phi,\perp\}\cup V$ denote the domain of message values including the null message and the special error symbol $\perp$. The set of all possible bags (multisets) of size k constructed from the elements of a set S is denoted as S^k. In the protocol, every non-faulty processor broadcasts at most one message over a channel in a single round. Let $B_i^r(j) \in W^R$ be the *in-bag* of R messages (one from each channel) that processor p_i receives from processor p_j during round r. If a processor p_i receives more than one message over the same channel from some (faulty) processor p_j during some round r, it sets its in-bag $B_i^r(j)=\{\perp\}^R$ immediately regardless of what messages it received (or will receive) from p_j over other channels in the same round.

We say that a bag B is *consistent in v* if and only if $B=\{\phi,v\}^R$ for some unique $v \in W$ and $v \neq \perp$. In other words, the bag contains copies of either a single non-error message or two messages, one of which has to be the null message. The *filter function*, $\sigma: W^R \rightarrow W$, that maps bags of size R to the valid message domain is defined as follows:

$$\sigma(B)= \begin{cases} v & \text{if B is consistent in v} \\ \delta & \text{otherwise.} \end{cases}$$

Finally, we define the *decision function* with threshold z, $\Psi^z: W^n \rightarrow V$, that maps bags of n messages to a value to be accepted as follows:

$$\Psi^z(M) = \begin{cases} v & \text{if } \varphi(M)=v \text{ and } \theta(M,v)>z \\ \delta & \text{otherwise} \end{cases}$$

where $\varphi(M)$ selects the most frequent non-ϕ element in the bag M and $\theta(M,x)$ counts the number of times element x appears in M.

The protocol to achieve RB in the presence of malicious processor failures is shown in Figure 3.2.

Round 1: For p_1 (the transmitter with local value $v \in V$)
broadcast(v,*);

Round 2: For all p_i
$m_i \leftarrow \sigma(B_i^1(1))$;
broadcast(m_i,*);

At the conclusion of round 2: For all p_i
$M_i \leftarrow \{j=1,2,\ldots,n : \sigma(B_i^2(j))\}$;
accept $\Psi^z(M_i)$;

Figure 3.2. Protocol P2 with Threshold z:
Reliable Broadcast for Malicious Processor Failures.

Let us consider the result of a processor failure during the **broadcast**(v,*) operation in this system. Note that if the failure is such that a processor broadcasts conflicting messages over faulty channels or through faulty links to non-faulty channels, the failure is hidden from all other processors and the transmitter appears to be non-faulty. It is only when the conflicting messages are visible to other processors (because they reach non-faulty channels) that agreement can be compromised. With this in mind, we say that a sending processor is *overtly malicious* if the bag of message values as observed on the channels due to a broadcast is not consistent in some value. A receiving processor p_i is called a *witness to an overtly malicious failure* of processor p_j in round r if and only if $B_i^r(j)$ is not consistent in any value.

Similarly, a processor exhibits an *overt omission* failure if it is not overtly malicious but broadcasts a null message over at least one non-faulty channel to which it is properly connected. In an R-redundant broadcast network, processor p_i *witnessing an overt omission failure* of processor p_j in round r requires that $B_i^r(j)=\{\phi\}^R$.

As we have defined it, the filter function σ is designed to detect exactly these failures. It results in ϕ for the witnesses of overt omission failures and in δ for the witnesses of overt malicious failures. We will next present results that bound the number of processors that can be witnesses to the failure of the transmitter and show that under no circumstance can other faulty processors collude to prevent agreement. This is accomplished by ensuring that a sufficiently large number of non-faulty processors are biased towards the same decision by the end of the first round.

Lemma 3.1. *In an R-redundant broadcast network, the number of witnesses to an overt omission failure of the transmitter by the end of the first round in Protocol P2 is either* $n-1$ *or in the range* $[0,\lambda]$.

Proof: Let x denote the number of non-faulty channels to which p_1 is properly connected and sends non-null messages. Clearly $0 \leq x < R$ since $x=R$ would imply that the transmitter is non-faulty. Since 1-connectivity is ensured, when $x=0$, all $n-1$ remaining processors witness the failure. If $x>0$, then for a processor to witness the failure it must have faulty links to all x channels. As there are at most λ actual faulty links, the number of such witness processors is $\lfloor \lambda/x \rfloor$. This results in a number of witnesses in the range $[0,\lambda]$. ∇

Lemma 3.2. *In an R-redundant broadcast network, the number of witnesses to an overtly malicious transmitter by the end of the first round in Protocol P2 is in the range* $[n-\lambda-1, n-1]$.

Proof: Let p_1 fail by broadcasting two different (non-null) messages over two non-faulty channels to which it is properly connected. For a processor not to witness this failure, it must not be properly connected to one of these channels. Since there are λ possible faulty links, there can be at most λ such processors. Consequently, the number of witnesses (not counting the faulty transmitter) is at least $n-\lambda-1$ and at most $n-1$. ∇

Recall that F and L denote the upper bounds for the actual number of faulty processors (f) and links (λ), respectively.

Theorem 3.3. *If* $n>F+L+f+\lambda$, *then protocol P2 with threshold* $z=F+L$ *implements RB for processors that may exhibit malicious failures.*

Proof: We proceed by case analysis.

If the transmitter is non-faulty, all of the in-bags will be consistent in v by the end of the first round. For all non-faulty processors, the filter function yields v, which is echoed in round 2. The $n-f$ such echoes will be filtered by the end of the second round and added as v values to the M_i bag for each processor. For each non-faulty p_i, the bag M_i will contain at least $n-f$ occurrences of v and at most f occurrences of some other value. Even if all of the f faulty processors collude and try to force a decision on this other value, it will fail since $f \leq F+L$ and the threshold of the decision function will not be exceeded. By rewriting the hypothesis of the theorem as $n-f \geq F+L+\lambda+1$ and noting that $\lambda \geq 0$, we have $n-f \geq F+L+1$ so that v will be selected by all of the decision functions of the non-faulty processors. Thus we have shown non-triviality for Protocol P2. We next argue that it also ensures agreement if the transmitter is faulty.

Assume that the transmitter fails by overt omission. By Lemma 3.1, there will be $k \in \{0,1,\ldots,\lambda,n-1\}$ witnesses. If $k=n-1$, then at least $n-f$ processors will broadcast ϕ in the second round. By an argument identical to that of the previous paragraph, all non-faulty processors accept δ. If $k \leq \lambda$, then $n-f-k$ non-faulty processors do not witness the failure, and for each, the filter function yields the value v since each has an in-bag consistent in v. In the second round, $n-f-k$ messages consistent in v are broadcast such that for each non-faulty processor p_i, the bag M_i will contain $n-f-k$ occurrences of v. Since $k \leq \lambda$, the inequality $n-k-f \geq F+L+1$ is assured by the hypothesis $n>F+L+f+\lambda$ and there are enough v values in M_i to exceed the threshold function. The bag M_i can contain at most $f+\lambda$ occurrences of values other than v. But as $f+\lambda \leq F+L$ the threshold function is not exceeded and the decision function yields v for all non-faulty processors.

Finally, assume that the transmitter is overtly malicious. By Lemma 3.2, k processors witness the failure where $n-\lambda-1 \leq k \leq n-1$. In the second round, at least $k-(f-1)$ processors will echo δ messages. By the end of the second round, M_i for each non-faulty p_i will contain at least $k-f+1$ occurrences of δ. Since for all possible values of k we have $k-f+1 \geq F+L+1$ and the quantity $n-(k-f+1)$, which is at most $f+\lambda$, does not exceed the threshold, the decision function yields δ for each non-faulty processor. ∇

3.4. Coping with Channel Partitions

Consider the failure mode of an individual broadcast network that partitions the set of links into two disjoint groups—those that receive a message and those that do not. For example, in an Ethernet, the coaxial cable physically breaking, the failure of a repeater unit between two channel segments and an interface unit causing collisions shorter in duration than the end-to-end propagation delay of the channel are rare but possible instances of partition failures. We now relax the faulty channel semantics to allow such partition failures.

> *Faulty Channel Semantics:* A faulty channel fails to deliver a broadcast message to a subset of the links connected to it. Those links that do receive a message receive the same message.

With respect to a given broadcast, the group of links receiving the message is called the *in-partition*, and the group not receiving it is called the *out-partition*. Note that our definition allows the composition of these partitions to change dynamically from one message to another and from one broadcaster to another. Let us observe the behavior of the communication system from a processor's point of view for a given broadcast. The case where the processor's link is non-faulty but is in an out-partition and the case where its link has failed and the processor is in an in-partition are indistinguishable—they both result in the processor receiving a ϕ message. This observation suggests a duality between link and channel failures. We can either assume that channels are failure free and simulate a partition failure by pretending that all of the links in the out-partition have failed regardless of their actual state, or we can assume that links are failure free and simulate their failure by constructing the appropriate out-partitions for channel failures. We will pursue the first alternative.

When we simulate channel partitions as multiple link failures, links can be "faulty" either because they have failed or because they belong to the out-partition of a faulty channel. With this new interpretation, all of our previous results continue to hold for this more general faulty channel semantics. All that is required is that we be able to account properly for the total number of faulty links. We characterize each channel according to the configuration of link failures for that channel. For channel i, let $\lambda_i(m)$ denote the number of processors that do not receive a given broadcast message m over that channel. If $\lambda_i(m)=0$ then channel i has not failed. If $\lambda_i(m)=n$ then channel i is said to have *failed totally*. For any other value, channel i is said to have *failed partially*. Throughout the execution of a protocol, a channel is non-faulty if it has not failed for any broadcast message. If all of the channel failures are total, then it is said to exhibit a *total fault.*[1] Any other channel behavior is called a *partition fault.* We let γ' count the number of faulty channels (either total or partition).

We will first examine the implications of the new semantics for faulty channels on the replication factor of our R-redundant broadcast networks. Recall that in such networks, $R>\lambda+\gamma$ is a sufficient condition for 1-connectivity. Clearly, the network can sustain more than $R-\gamma$ faulty links and still preserve 1-connectivity if there are multiple faulty links to the

[1]Note that this behavior corresponds to the earlier semantics for faulty channels whereby some messages are lost completely.

same channel. Since we do not allow any message forwarding by processors, 1-connectivity can be ensured if there is at least one non-faulty channel with all non-faulty links. Therefore, according to our new notation, 1-connectivity is guaranteed if $R>\gamma'$.

For each channel, we are interested in the maximum number of processors that do not receive some message after a broadcast. Let λ_i be the maximum of $\lambda_i(m)$ over all messages m broadcast over channel i during the execution of the protocol such that $\lambda_i(m)\neq n$. If the channel exhibits a total fault, then we define λ_i to be zero for that channel. Note that this definition results in $\lambda_i=0$ for both channels that fail totally and that are non-faulty. Under both circumstances, all processors receive the same message (either ϕ or what was broadcast) and the n "faulty" links need not be accounted (except for connectivity reasons). Let λ' be the sum of λ_i over all channels. With this new interpretation, all of our previous results can be extended to channel partition failures simply by replacing all occurrences of λ with λ'. In essence, λ' is an upper bound on the total number of link failures that can occur in each round. Note that the total number of faulty links for the duration of protocol execution can be much larger than λ'.

For certain network configurations, we can tolerate even more faulty links. These configurations are characterized as permitting a bound on the λ_i for each channel. For example, consider a broadcast network consisting of several channel segments connected through repeaters. If repeater failures are the only sources of channel partitions, and the number of processors connected to each segment is known, we can easily express the minimum number of processors that will receive every message broadcast over that channel. Since λ_i denotes the maximum number of processors that do not receive some message over channel i, $n-\lambda_i$ denotes the minimum number of processors that do receive every message over the same channel. If $n-\lambda_i>f$ can be ensured for each channel, we know that at least one non-faulty processor will receive each message broadcast over each channel. For omission failures, this condition is sufficient to guarantee that protocol P1 continues to implement RB in two rounds.

In summary, if the faulty links that simulate partitions are well-structured (as is the case with multiple Ethernet segments and repeaters), our protocols can sustain many more of them than what is allowed by the pessimistic accounting through λ'. As an example, Theorem 3.1 requires that the total number of faulty links not exceed n–f for protocol P1 to work. However, if $\lambda_i < n-f$ for each channel, the system can tolerate almost $R-1$ times as many faulty links.

3.5. Malicious Communication Component Failures

The failure mode for the links and channels we have examined so far can be characterized as omission failures. In this section we examine the consequences of network component failures that can undetectably alter the contents of messages.

Given that the communication media can alter messages, receiving a message over a single channel does not permit the processor to infer the actual message broadcast over that channel. Let $G > \gamma'$ be an upper bound on the number of faulty channels. If we ensure that $R > 2G$ and each processor communicates through **broadcast**(v,*), we can overcome the behavior of faulty communication components as follows. If a non-faulty processor broadcasts the message v, then all processors will receive at least G+1 copies of v since at most G copies could have been altered by faulty paths. Since there are more non-faulty paths between processors than faulty ones, receivers can always determine the actual broadcast message by simply selecting the majority element from the bag of received messages. Having guaranteed effective communication through this mechanism, the subsequent discussion will assume channels that are failure free. We can do this without loss of generality since malicious channel failure behavior can be simulated through the failure of links.

In the context of malicious communication failures, we modify slightly our previous notion of consistency for bags. Now, a bag B is said to be *consistent in v* if and only if v is a valid message and it is the majority element. A bag that is not consistent in any value is called *inconsistent.*

Let B^* denote the bag of messages as observed on the channels due to the execution of a broadcast by some processor over the R channels. We will now prove that protocol P2, with the new definition of consistency, achieves RB in the presence of malicious processor and link failures.

Lemma 3.3. *If* B^* *is consistent in* v, *then*

(i) *at most* λ' *processors may have inconsistent in-bags,*

(ii) *at most* λ' *processors may have in-bags consistent in some value* y *different from* v,

(iii) *at least* $n-\lambda'$ *processors have in-bags consistent in* v.

Proof: For the first two cases, a faulty link to a processor could turn a bag consistent in v as observed over the channels into an inconsistent bag or a bag consistent in some other value y as seen by the processor. In the worst case, there can be λ' such processors since there are at most λ' faulty links in each round. As a consequence, at least $n-\lambda'$ processors have non-faulty links and receive the bag of messages consistent in v. ∇

Lemma 3.4. *If* B^* *is inconsistent, then at most* λ' *processors receive a bag of messages consistent in some value.*

Proof: Trivially follows from the observation that a change to a single element of an inconsistent bag could make it consistent. ∇

Given these two Lemmata, we next show how protocol P2, with the new definition of consistency, can be adapted to support arbitrary communication failures. Let L' denote the upper bound for λ' for all executions of the protocol.

Theorem 3.4. *If* $n>F+L'+f+\lambda'$, *then protocol P2 with threshold* $z=F+L'$ *implements RB when processors and communication components may fail maliciously.*

Proof: If the transmitter is non-faulty, the bag of messages observed over the channels resulting from its initial broadcast is consistent in v. By Lemma 3.3 at least $n-\lambda'$ processors receive bags consistent in v. Among them, at least $n-\lambda'-f$ processors are non-faulty and echo v in the second round. Since by the statement of the theorem $n-\lambda'-f$ is greater than z and $\lambda'+f$ does not exceed z, the non-faulty processors accept v and non-triviality is assured.

If the transmitter is faulty, the bag of messages observed over the channels resulting from its initial broadcast can be either consistent in v or inconsistent. If it is consistent, we have decision on v as above. Otherwise, by Lemma 3.4 at least $n-\lambda'-f$ non-faulty processors receive inconsistent bags. In the second round these processors echo δ. By the same argument as in the previous paragraph, decision on δ is assured. ∇

We have found that when omission failures of processors are considered in systems with arbitrary communication failures, Theorem 3.4 continues to hold. It appears that the nature of the communication failures determines RB protocol requirements rather than processor failures. However, if we account communications failures for the duration of the protocol, as in Sections 3.2 and 3.3, and not during each round, a reduction in the total number of processors required is possible.

Let λ and γ count the actual numbers of (malicious) faulty links and channels respectively. Let L and G be upper bounds for λ and γ, respectively. We can guarantee effective communication by requiring $R>2(L+G)$ and by using the new definition of consistency. We redefine the decision function with threshold z, $\Psi^z:W^n \rightarrow V$, of protocol P2 such that

$$\Psi^z(M) = \begin{cases} v & \text{if } \varphi(M)=v \text{ and } \theta(M,v)>z \\ \delta & \text{otherwise} \end{cases}$$

where $\varphi(M)$ selects the most frequent non-ϕ, non-δ element in the bag M and $\theta(M,x)$ counts the number of times element x appears in M. Note that this new definition of Ψ favors decisions on non-δ values.

We say a bag B is *x-inconsistent* if B is not consistent and x is the minimum number of elements of B that need to be changed to make B consistent in some value. A bag B is *x-consistent in v* if B is consistent in v and x is the minimum number of elements of B that need to be changed to make B no longer consistent in v. Note that these definitions are exclusive.

Theorem 3.5. *If* $n>L+G+f+\lambda+\gamma$ *then the modified protocol P2 with threshold* $z=L+G$ *implements RB for omission processor failures and malicious communication component failures.*

Proof: Note that since processor failures are restricted to omission and $R>2(L+G)$, no processor can ever receive a message consistent in a value other than v in the first round. Also, in the second round, as processors can not generate spurious messages, there can only be consistent messages either in v or δ.

Let y be the number of link and channel failures that occur in the first round, ($y\leq\lambda+\gamma$). We proceed by case analysis.

If B^* is x-consistent in v then at least $n-\lfloor y/x \rfloor$ processors receive a message consistent in v in the first round. For all possible values of x and y at least $n-\lambda-\gamma$ processors receive a message consistent in v from the transmitter. At least $n-\lambda-\gamma-f$ v-messages are broadcast in the second round. Each bag M_i will contain at least $n-\lambda-\gamma-f$ v-values and since $n-\lambda-\gamma-f>z$ is ensured by the statement of the theorem, the decision function chooses v.

If B^* is x-inconsistent then at most $\lfloor y/x \rfloor$ processors receive a consistent message in the first round. For all possible values of x and y at most $\lambda+\gamma$ processors receive a consistent message from the transmitter. At most $\lambda+\gamma$ non-δ messages are broadcast in the second round. The remaining $n-\lambda-\gamma$ messages received in the second round are δ, ϕ or inconsistent. As the threshold function will not be exceeded by any non-δ value, a decision is reached on δ. ∇

We note that the total number of processors required is roughly twice the number of faulty communication components plus the number of faulty processors.

3.6. Optimizations

For a given level of fault tolerance in systems composed of many processors, only the minimum number of processors required by the protocol need to actively participate in the protocol's execution. For example, in protocol P1, Theorem 3.1 only requires $f+\lambda$ processors to implement RB. If $n>>F+L$ we can arrange that only $F+L$ processors participate actively in the protocol execution (i.e., by broadcasting messages) while the remaining processors participate as passive listeners. All of the processors execute the same protocol, but the passive ones simply refrain from performing any of the prescribed broadcasts. Note that although only $f+\lambda$ processors are required by Theorem 3.1, this value may vary from one execution to the next since it depends on the actual number of faulty components. Therefore, $f+\lambda$ cannot be known *a priori* and we have to use the $F+L$ upper bound. Obviously, these same observations apply to the other protocols.

Note that whenever the threshold represents the majority of the active processors, a decision can be made as soon as the threshold is exceeded, and the remaining messages can simply be ignored.

3.7. Conclusions

Prior to the work described in this thesis the RB problem had only been studied in point-to-point networks where F+1 rounds are necessary to solve RB. We offered two-round solutions in R-redundant broadcast networks at the cost of network redundancy. Most of the existing results for the RB problem either require a perfectly reliable network or processor and link failures are lumped together under the same parameter F. In point-to-point networks without authentication, at least 3F processors are required to tolerate F faulty components. Under the same assumptions we only require that $n>F+L+f+\lambda$. Since $F+L+f+\lambda\leq 2(F+L)$, our protocols can tolerate 50% more faulty components with the same number of total processors.

It should be noted that the F+1 lower bound for point-to-point networks has not yet been achieved when processors may fail maliciously and the total number of messages required is polynomial in F. The best result to date [Sr85a] requires $n>3F$ and 2F+1 rounds. Coan and Lynch [Co85] proposed a family of protocols that solve RB in $(1+\varepsilon)(F+1)$ rounds. They require $O(n^6)$ messages for their 2F+1 round solution ($\varepsilon \cong 2$). As ε approachs zero the number of messages required approaches $O(n^{1/\varepsilon})$.

CHAPTER 4

Reliable Broadcast Protocols through Partial Broadcasts

4.1. Introduction

In Chapter 3, we demonstrated two-round protocols to implement RB that are based on communication architectures other than point-to-point networks. These results demonstrated that performance, previously not thought possible, could be realized practically. In this chapter, we explore general tradeoffs between execution time of RB protocols and properties of the underlying communication network when the processor failures are restricted to omission. The first network feature we consider is the *broadcast degree.* Informally, this feature captures the semantics of communication over broadcast channels. We derive and prove the correctness of a simple protocol that is able to implement RB in the continuum of communication network structures from full broadcast channels at one extreme to fully connected point-to-point graphs at the other. This protocol degenerates into the F+1-round solution for point-to-point graphs [Ha84a] and our two-round solution for broadcast networks. The execution time increases linearly as the broadcast degree decreases in going from full broadcast networks to point-to-point networks. We then include the *survivor diameter* in our characterization of the communication network. We show that this two-dimensional characterization includes all of the familiar network topologies as special cases.

The main contribution of the results in this chapter is that for a desired level of fault tolerance, a spectrum of protocol performances can be "bought" by investing in the appropriate communication network structure. We are able to achieve increases in performance by exploiting the properties of the communication architecture to effectively restrict the externally visible behavior of faulty processors.

4.2. Broadcast Degree

The Broadcast Network Property we discussed in the previous chapter ensures that under any failure scenario, a processor cannot send conflicting messages to other processors in a single broadcast. For a given broadcast network, the set of processors that receive the (same) message in response to a broadcast is called the *receiving set* for that broadcast. The *broadcast degree* of a network is defined to be the lower bound on the size of the receiving sets for all broadcasts in the network. The receiving sets can vary from one broadcast message to another as well as from one transmitter to another. For a network to have broadcast degree b, all we require is that each of these sets have at least b elements. We assume that every processor receives its own broadcasts regardless of failures. Note that all communication failures manifest themselves in defining a particular broadcast degree for the network. Given that the system satisfies BNP, the behavior of faulty processors is restricted in that all of their actions are witnessed by at least $b-1$ other processors.

A distributed system based on a broadcast network can be modeled as a hypergraph $G=(N,E)$ where the nodes $N=\{1,2,\ldots,n\}$ represent the processors and the single hyperedge $e=\{1,2,\ldots,n\}$ represents the broadcast channel. In a network with broadcast degree b, communication failures alter the composition of the hyperedge without violating the condition $b \leq |e| \leq n$. For this graph to remain connected despite communication failures, it is clear that the broadcast channel has to be replicated as in the R-redundant broadcast networks.

The redundant broadcast network interconnects processors through a sufficient number of channels, each exhibiting the Broadcast Network Property, such that the network remains connected (i.e., CA is satisfied). In the graph model, the edge set E now contains a number of hyperedges that are arbitrary nonempty subsets of N. The *broadcast degree* of the entire redundant network is defined to be the minimum of the broadcast degrees of its constituents. Multiple rings, Ethernets or buses can be used to construct redundant broadcast networks. In the rest of this chapter, we will consider only redundant broadcast networks and refer to them simply as broadcast networks.

We assume that **broadcast**(v,*), when invoked by a non-faulty processor, functions by broadcasting the message v over all the channels which the processor is connected to. As in the previous chapters, no assumptions are made regarding the order in which channels are selected or the possibility of omitting messages entirely for some channels by faulty processors.

4.3. From Broadcasts to Point-to-Point Communication

We will initially use the broadcast degree of a network as its characterization. Clearly, for a system with n processors, $1 \leq b \leq n$. Note that for b=n, every broadcast is guaranteed to be received by every processor. Therefore, a protocol in which the transmitter simply executes **broadcast**(v,*) and other processors accept any message received, suffices to achieve RB since BNP with b=n trivially guarantees agreement and non-triviality. At the other extreme, when b=1, there can be isolated processors and the network cannot be guaranteed to remain connected no matter how many times the channels are replicated. Consequently, the smallest value of b for which RB can be solved is 2.

Consider a broadcast network with b=2. In this case, the network can only guarantee that for some broadcast during the protocol execution, the message will be received by only one other processor. In other words, for certain communications, the broadcast channels degenerate into point-to-point links between two processors. As this can happen arbitrarily often, the net effect of having a broadcast degree of only 2 can be characterized as the broadcast network becoming logically equivalent to a point-to-point network. The "logical edges" that are formed in the network result in a fully connected system graph. The physical paths corresponding to these logical edges can dynamically switch from one channel to another.

Summarizing, in a broadcast network where no communication failures occur, such that b=n can be attained, there exists a trivial 1-round protocol that implements RB. On the other hand, a broadcast network with b=2 is logically equivalent to a point-to-point network where we know that F+1 rounds are necessary and sufficient to achieve RB [Fi82, Do83b, Ha84a]. What remains to be resolved is the execution time required for a RB protocol in networks where the broadcast degree is between these two extremes. We will call such a limited broadcast capability a *partial broadcast*.[1]

4.4. Reliable Broadcast Protocols Through Partial Broadcasts

In this section, our RB protocol is based on requiring that a **broadcast**(v,*) by a non-faulty processor be received by all other processors, regardless of the broadcast degree of the network. Consequently, any protocol where processors simply echo the messages received in the previous round, will guarantee that all non-faulty processors receive a message one round after the first non-faulty processor receives a message. As processor failures are restricted to omission, all of the messages generated by such a protocol can only contain the transmitter's value. What we must demonstrate is an upper bound on the number of rounds such that all non-faulty processors can accept the default value δ if they have not received any messages by this time.

Consider a broadcast network where $f+1 \le b \le n-1$. In this case, the initial broadcast by the transmitter with local value v is received by at least one but not necessarily all

[1] The term *multicast* has also been used for such limited broadcasts.

Round 1: For p_1 (the transmitter with local value v)
broadcast(v,*);

Round j, $2 \leq j \leq r$: For all p_i
if p_i received a message in round j–1 **then**
let m *denote the message received in round* j–1
broadcast(m,*);
accept m;
fi;

At the conclusion of round r: For all p_i that have not yet accepted a value
if p_i received a message in round r **then**
let m *denote the message received in round* r
accept m;
else
accept δ;
fi;

Figure. 4.1. Protocol P3 with Parameter r:
Reliable Broadcast for Partial Broadcast Networks

non-faulty processors. By the above observations, all processors are guaranteed to receive the transmitter's message by the end of the second round. They accept v if a message is received in the first or second round. Otherwise they accept the default value. The resulting two-round protocol is identical to protocol P1 of the previous chapter. Recall that the RB protocol for omission processor failures presented in Chapter 3 requires $n \geq \lambda + f$. That is, it may be the case that no non-faulty processor receives the transmitter's initial message. In this sense, the requirement $b \geq f+1$ is more restrictive than $n \geq \lambda + f$. We can see this by noting that λ corresponds to an upper bound for the complement of the broadcast degree, i.e., n–b. Therefore, the requirement $n \geq \lambda + f$ can be rewritten as $n - \lambda \geq f$ or $b \geq f$. Compare this with the requirement $b \geq f+1$ for the protocol just proposed.

Protocol P3 displayed in Figure 4.1 implements RB for broadcast networks with broadcast degrees in the range $2 \leq b \leq f+1$. It is parametrized by the number of rounds to completion. Note that this protocol closely resembles the one presented in [Ha84a].

For simplicity, we let each processor *accept* at intermediate rounds of the protocol's execution. Since r is fixed and known by all processors, the intermediate accepts may be replaced by an assignment to a local variable and then at the r^{th} round the processors accept the value stored in this local variable. It should be clear that after this simple transformation, protocol P3 satisfies the simultaneity requirement. The following theorem establishes its correctness.

Theorem 4.1. *For a broadcast network with broadcast degree b such that* $2 \leq b \leq f+1$ *protocol P3 implements RB in r rounds provided* $f-b+3 \leq r$.

Proof: Let v denote the transmitter's local value. We proceed by case analysis. If the transmitter is non-faulty, all processors receive v by the end of the first round and accept it in the following round. Therefore the agreement and non-triviality requirements are satisfied.

In the remaining cases the transmitter is faulty. If it sent no messages at all, no processor ever receives a message and they all accept δ by the end of round r. If the transmitter broadcasts over some channel whose broadcast degree is n, then all processors receive the message and accept v in the second round.

Finally, if a faulty transmitter broadcasts some message and a non-faulty processor p_z receives it in round j, for $j<r$, then p_z will echo v in round j+1. By our assumption, all processors receive v in round j+1 and by round *min*(r, j+2) they all will have accepted v. Thus agreement can be violated only if the first non-faulty processor to receive v does so at round r while some other non-faulty processors do not receive v. For contradiction suppose this scenario occurs. According to the protocol, a processor can only broadcast at round j if it has received its first message at round $j-1$. Therefore, for the above scenario to occur there must be a chain of faulty processors $p_1 p_2 \ldots p_r$ with p_1 being the transmitter and each p_i, $1<i\leq r$, receiving its first message from p_{i-1} at round $i-1$ and echoing it in

round i. Some but not all non-faulty processors receive their first message from p_r at round r. Given that the network has broadcast degree b, we show that at least b+r–2 faulty processors are necessary for this scenario to occur. First, the broadcast of p_1 has to be received by b faulty processors including p_2 but none of $p_3\, p_4 \ldots p_r$. In the subsequent rounds (2 through r–1) the b faulty processors receiving the broadcast must contain at least one previously unaccounted faulty processor. Consequently, at least b+r–2 faulty processors are needed: b in the first round and r–2 in the remainder of the protocol. This contradicts the hypothesis of the theorem requiring $f-b+3 \leq r$. ∇

Clearly, if $f-b+3 \leq r$ then theorem 4.1 guarantees that all processors will have reached a decision by round f–b+3.

4.5. Coping with Networks of Arbitrary Diameter

Recall that within a round, each processor can communicate with all other processors. We define a *phase* to be the interval of time necessary for all processors to communicate only with their immediate neighbors in the current system graph. Note that during the execution of the protocol, the original system graph will dynamically evolve as processor and communication failures occur. Consequently, the set of immediate neighbors of a processor can change from phase to phase. The assumption we made in the previous section, whereby a broadcast by a non-faulty processor is received by all other processors in the same round, implies a system where phases and rounds are equivalent.

In this section we relax this assumption and consider systems where a single phase is no longer sufficient to implement a round. In such systems, let the *survivor diameter*, denoted as d, be the maximum number of phases required to achieve a round under all permissible failure scenarios.

The *diameter* of a graph, denoted as d_0, is defined to be the number of edges in the longest of the shortest paths between all pairs of vertices. Using our terminology, the diameter of a network is the maximum number of phases necessary to implement a round in the absence of failures. It is important to note that whereas d_0 has a purely static definition, the survivor diameter of a system is defined dynamically. Consider a network with survivor diameter d. Such a network can be actually partitioned during some phases but still be connected when viewed over the entire d phases. In fact, in systems where the communication components fail and recover frequently, it is conceivable that at any given time, the network may contain processor pairs that cannot communicate. For a network to have survivor diameter d, all that is required is that the failures in the system are such that no processor is prevented from receiving a message broadcast by any other non-faulty processor within d phases. Thus, we can relax the Connectivity Assumption by replacing it with the survivor diameter specification. We note that our definition of the survivor diameter is closely related to the notion of "eventually connected networks" [Aw84].

Protocol P3 of the previous section was presented in terms of rounds. Given that in general d phases are required to implement a round, protocol P3 could be used for networks with arbitrary d provided that the **broadcast** primitive is viewed as the invocation of a d-phase delivery protocol. In this approach, the total execution time of the protocol is clearly $d(f-b+3)$ phases. However, as the next result shows, we can achieve RB in a number of phases that is an additive (rather than a multiplicative) factor of d more than before by pipelining the phases through the network. Dolev and Strong present a similar result for arbitrary processor failures in perfectly reliable, point-to-point networks [Do83b].

In protocol P3, we simply replace rounds with phases such that r now is the phase counter and the **broadcast** primitive, when executed by a non-faulty processor, distributes the message to its current immediate neighbors, as before.

Theorem 4.2. *For a broadcast network with survivor diameter* d *and broadcast degree* b, *such that* $2 \leq b \leq f+1$, *protocol P3 implements RB in r phases provided* $f-b+d+2 \leq r$.

Proof: Let v denote the transmitter's local value. We proceed by case analysis. If the non-faulty processors receive no messages by the end of phase r they all accept δ. In the remaining cases some non-faulty processors receive v during the execution of the protocol.

If the first non-faulty processor receives v at phase j, for $j \leq f-b+2$, then all non-faulty processors will receive v by the end of the next d phases. Since $j+d \leq r$ for all allowed values of j, all non-faulty processors receive and accept v within r phases. Note that this case applies for non-faulty transmitters, and therefore the non-triviality condition is satisfied.

We claim that this represents the latest phase by which a non-faulty processor can receive the v message. For contradiction, suppose the first non-faulty processor to receive v does so at phase $j>f-b+2$. A chain of faulty processors $p_1,p_2,\ldots,p_j$ with the same properties as the one introduced in the proof of Theorem 4.1 is required. Also the $b-1$ processors (say $p_2,q_3,\ldots,q_b$) that receive the transmitter's initial broadcast must all be faulty and distinct from $p_3,\ldots,p_j$. Thus the total number of faulty processors necessary for this scenario is $j+b-2$. But this exceeds f for any value of j, a contradiction.

In summary, either a non-faulty processor receives v by phase $f-b+2$ and all non-faulty processors accept v, or no non-faulty processor receives a message and all accept δ at phase r. Therefore the agreement condition is satisfied, concluding the proof. ∇

4.6. Conclusions

We have studied the reliable broadcast problem for omission failures in distributed systems where the communication network is characterized by its broadcast degree and survivor

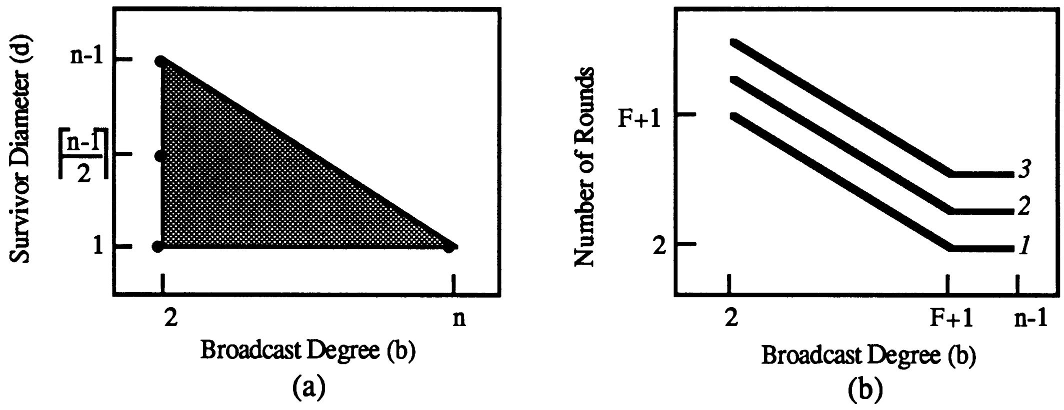


Figure 4.2. The (b,d) Networks

diameter. Conventional network topologies such as fully connected point-to-point graphs, simple linear chains and rings correspond to the special cases (b=2,d=1), (b=2,d=n−1) and (b=2,d=$\lceil (n-1)/2 \rceil$), respectively, of this characterization. Figure 4.2 (a) displays the spectrum of communication networks that are characterized by the parameters (b,d). The 1-connected broadcast network is marked in the figure along with the networks mentioned above. In general, the class of networks that are generated for arbitrary b and d correspond to common architectures consisting of a collection of (possibly non-homogeneous) broadcast networks that are interconnected through gateways where any of the network components may fail.

We have exhibited a simple protocol to implement reliable broadcast in systems based on such communication networks. Figure 4.2 (b) shows the execution time of this protocol in networks with arbritrary broadcast degree and survivor diameters of 1, 2 and 3. The protocol degenerates into the familiar solutions to RB for the extreme points of the characterization—fully connected point-to-point networks and full broadcast channels. In both cases, our protocol achieves the established execution time lower bounds for RB ([Ha84a] and Theorem 3.2). The protocol is optimal in execution time for the entire design space as it achieves the lower bounds established by Babaoglu and Stephenson [Ba85b].

The functional dependency of the protocol execution time on the system parameters suggests various tradeoffs when constructing distributed systems suitable for fault-tolerant computing. For a given execution cost, additional resiliency to processor faults can be directly exchanged for a larger broadcast degree or smaller survivor diameter. Alternatively, for a desired level of resiliency to processor faults, additional performance can be achieved by investing in network components so as to guarantee larger broadcast degrees or smaller survivor diameters.

It should not be a surprise that partial broadcast is a powerful primitive for solving the reliable broadcast problem. However, given that distributed protocols often rely on broadcasting, we expect similar tradeoffs to exist in a larger class of problems.

CHAPTER 5

Complexity of Network Communication

5.1. Introduction

In this chapter, we address the following question: Is it meaningful to compare the time complexity of protocols as measured in rounds when they are executed in systems with different communication networks ?

While the concept of a round formally abstracts the information exchange and the logical ordering needed to reason about distributed protocols, it hides the communication effort required for its implementation. From a practical point of view what is most important is the length of the real time interval that elapses during protocol execution. Also of pratical importance is the cost of achieving a certain level of resiliency, and the speed with which it can be achieved. While we will not answer all these questions, we will provide some measures that can be used to answer them.

Note that a fully-connected point-to-point network with n processors has n(n−1) links[1] such that all of these links could be delivering messages simultaneously. In a broadcast network, however, only a single message can be in a channel at any given time. If the message delivery delay between two adjacent processors is taken as the unit of time, then a round in fully-connected point-to-point networks requires 2 such units— one unit in each direction. In broadcast networks, on the other hand, a round requires n time units.[2]

[1] We assume that the links are half-duplex. In other words, the messages can travel in both directions, however, not at the same time.

[2] We assume that broadcast channels are time multiplexed. There are broadcast networks where the channels are frequency multiplexed. In such networks multiple messages can be in the channel simultaneously. Logically, these networks could be modeled by dedicating one channel per processor.

5.2. Communication Primitives

In the fault-tolerant systems we consider, a processor wishing to disseminate any information broadcasts it to all processors. Thus, a round can be implemented by having each processor in the network simultaneously perform broadcasts. We call this implementation of a round *distributed message exchange* (DME).

For a broadcast to tolerate processor failures in a point-to-point network, multiple copies of the same messages must be sent through disjoint paths. A *k-disjoint broadcast* is one in which the broadcast message is sent through k-disjoint paths to all processors not directly connected to the broadcaster. A single copy is sent to the immediate neighbors of the broadcaster. In the absence of processor failures, each processor receives k copies of the same message, unless it is directly connected to the broadcaster. Similarly, a *k-disjoint DME*, denoted k-DME, is one in which all processors simultaneously execute a k-disjoint broadcast.

5.3. Lower Bounds

In this section we derive a lower bound for the time complexity of a k-DME operation in point-to-point networks. A point-to-point network can be represented as an undirected graph, where the vertices represent the processors and the edges represent the links. We will use the network and graph notation interchangebly.

Clearly, each vertex must have degree at least k in order for a k-DME to be possible. If not, there would be pairs of vertices that do not have k disjoint paths connecting them. Let each vertex in the graph have a common degree $d \geq k$.

In a k-DME, each processor receives one copy of each message sent by its d immediate neighbors, and up to k copies of each message sent by the other $n-d-1$ processors. A total of $nd+nk(n-d-1)$ messages are received during a k-DME. The total number of edges in the graph is nd/2. If we assume that each edge carries an equal number of messages, then $2+2k(n-d-1)/d$ messages are transmitted over each edge. If none of the edges or vertices are bottlenecks for communication, the minimum time necessary for performing a k-DME is $2+2k(n-d-1)/d$ time units.

The above analysis requires that the network graph be symmetric. A graph is *symmetric* if there are automorphisms between every pair of vertices and every pair of edges [Ha69]. In other words, each vertex (or edge) is in the same relative position in the graph as any other vertex (or edge). In general, symmetric graphs cannot be constructed for all values of n and d; moreover, few graphs with degree d have connectivity d. If the graph is not symmetric, certain edges or vertices may be required to carry more messages than others, and may become bottlenecks. In this case, a k-DME will require a longer time. Thus $2+2k(n-d-1)/d$ time units represents a lower bound for k-DME in point-to-point networks with n processors where the degree of each vertex is d.

5.4. Complexity of the Reliable Broadcast Problem

In the presence of malicious processor failures, we compare the time required to implement Reliable Broadcast in R-redundant broadcast networks to the minimum time possible in point-to-point networks with perfectly reliable links. Recall that F denotes the maximum number of faulty processors, and in the redundant broadcast networks, L and G denote the maximum numbers of faulty links and channels, respectively.

The first row of Table 5.1 indicates the minimum number of processors necessary for both systems. The next row is the minimum connectivity required for the point-to-point network and the minimum redundancy for broadcast networks. Given the first two rows, we compute the number of links required in row three. Based on our discussion in the previous section, the time required for a k-DME is given in row four. The lower bounds for RB in the two systems are F+1 and 2 rounds, respectively. Finally,

Table 5.1 Complexity of RB for F Malicious Processors

	R Redundant Broadcast Networks	Point-to-point Networks with degree d
n	2(F+L)+1	3F+1
connectivity (redundancy)	R=L+G+1	$d \geq k=2(F+1)$
number of links	2R(F+L)+R	$\frac{nd}{2}$
k-DME time	2(F+L)+1	$\frac{6F}{d}(2F+1)-4F$
rounds for RB	2	F+1
RB time	4(F+L)+2	$\frac{6F}{d}(2F+1)(F+1)-4F(F+1)$

the last row is the total time required to implement RB as obtained by multiplying the previous two rows together. Note that we have given the point-to-point network system every advantage possible. We have assumed that a fully-symmetric graph exists for every n and d combination. In practice, these graphs are extremely rare, and, most likely, the graph that is attainable will have communication bottlenecks such that the minimum k-DME time cannot be achieved. Also, we have assumed that a RB protocol realizing the lower bound for execution time (in rounds) exists. The best known RB protocol for malicious processors in a point-to-point network actually requires 2F+1 rounds [Sr85a].

5.5. Conclusions

The unit of time used for the comparisons in this chapter was defined as the message delivery delay between two adjacent processors. This measure is a better basis for comparing protocols in different network implementations than the round measure. However, the message delivery delay need not be identical in different network implementations. Most probably, the maximum delay for message delivery in a broadcast network is greater than the corresponding time in a point-to-point network. However, broadcast networks typically have extremely short minimum and average delay times.

In point-to-point networks that are less than fully connected, the total time required for RB is proportional to F^2. On the other hand, the total time required by R-redundant broadcast networks to implement RB is proportional to only F+L. Only when the point-to-point network becomes fully connected do the two execution times become comparable.

Note that up to now, we have not compared the two systems based on monetary criteria. Typically, the cost of a system is proportional to the sum of its hardware components—processors, links and channels. To make a meaningful comparison, one needs to assess the realtive value of each type of component. Since this assessment is very system dependent, we will refrain from comparing costs. However, the total component counts that are given in Table 5.1 could be easily turned into dollar values through appropriate scaling.

CHAPTER 6

Clock Synchronization

6.1. Introduction

In this chapter we demonstrate how to create the abstraction of a perfectly-synchronized distributed system when the underlying system can only be characterized as "being synchronous." Since our main interest in distributed systems is their use as a fault-tolerant computing environment, all our results are geared towards fault-tolerant distributed protocols. For example, we base our approach to synchronization on the Distributed Message Exchange (DME) operation since it is a basic communication primitive for designing fault-tolerant protocols.

We implement the perfectly-synchronized abstraction in layers. In Section 6.2, we show how any protocol that requires a perfectly synchronized system can be modified to work in synchronous systems that only guarantee a bound on the maximum clock differences. In Sections 6.3 through 6.8 we describe how this intermediate layer can be implemented on top of a typical hardware layer. Finally, Section 6.9, concludes the chapter.

The main contribution of this chapter is the demonstration that the perfectly-synchronized abstraction can be achieved at no extra cost in message complexity if DME operations occur with sufficient frequency.

6.2. Coping with Clocks that are not Perfectly Synchronized

Let us first consider a synchronous system where the clocks of non-faulty processors differ by at most Δ units (known *a priori* by all processors). We call such systems Δ-synchronized.

Let L be a constant known to all processors that denotes the duration of each phase.[1] Since all processors know when the protocol is to be initiated, they progress through phases in lockstep. Let T be the starting time of an arbitrary phase, known to all processors. If virtual clocks are allowed to be at most Δ units apart, a (slow) processor could receive messages from (fast) processors as early as time $T-\Delta$, as measured on its virtual clock.[2] Conversely, a (fast) processor could receive messages from (slow) processors as late as virtual time $T+L+\Delta$. Therefore, the duration of a phase could be redefined to be $L+2\Delta$ with messages received within the interval $[T-\Delta, T+L+\Delta]$ belonging to the current phase. This interval consists of the two *passive* segments $[T-\Delta, T]$ and $[T+L, T+L+\Delta]$ where processors simply wait for arriving messages, and the *active* segment $[T, T+L]$ during which they execute the protocol.

Using this simple construction, a protocol requiring r phases would take $r(L+2\Delta)$ time units. By overlapping the passive segments of two adjacent phases, we can reduce this time to $r(L+\Delta)+\Delta$. All that is required is that messages be identified with the appropriate phase number, so that late messages from the previous phase and early messages from the current phase can be distinguished. We have just demonstrated that Δ-synchronized and perfectly-synchornized systems are equivalent in the sense that any protocol that executes in one system can execute in the other.

6.3. Overview of the Clock Synchronization Problem

Next we will demonstrate how to achieve Δ-synchronized systems given only processors with hardware clocks. In the remaining sections of this chapter, a processor is defined to be non-faulty only if it correctly executes its protocol and possesses a non-faulty clock

[1] Since the message delivery time is bounded, the duration of a phase can also be bounded by assuming that each message takes the maximum delivery time.

[2] Actually, no message could be received before $T-\Delta$ plus the minimum message delivery time that belong to the phase starting at time T. For simplicity we will ignore this term for the time being.

(formally defined below). Any other situation describes a faulty processor. We shall use lower case letters (including Greek letters) to denote real time values and upper case letters to denote clock time values. Clocks are modeled as functions from real time to clock time and are given upper case names. The inverse function of a given clock is identified by the same clock name but in lower case. Thus for any clock C, real time t and clock time T

$$C(c(T))=T \text{ and } c(C(t))=t.$$

Let $H_i(t)$ denote the value of processor p_i's hardware clock at real time t. For a given constant ρ, a non-faulty clock H_i satisfies:

Hardware Clock condition[3] (HC): $\left|(dH_i(t)/dt) - 1\right| \leq \rho$

Thus, any two non-faulty clocks that were initially X units apart could be at most $X+2\rho\tau$ apart after a real time interval τ. Clearly, the clocks must be resynchronized periodically if their differences are to remain bounded. To simplify the presentation, we model resynchronization by starting a new *virtual clock* at each synchronization period. After the j^{th} synchronization period, the virtual clock of processor p_i at real time t is denoted as $C_{i,j}(t)$. We shall drop the synchronization period subscript when it is clear from context. The virtual clock of a processor advances at the same rate as its hardware clock except when it is being resynchronized. Even then, it is never set backwards.

For a given network, let d_M and d_m denote the maximum and minimum message delivery times, where $0 \leq d_m \leq d_M$. The *network uncertainty* for message delivery times is defined to be $\mu = d_M - d_m$.

Clock synchronization is achieved when processors possess virtual clocks that are monotone functions of real time and the clocks of any two non-faulty processors never differ by more than a constant Δ [La82a]. Since we start a new clock at each period, we

---

[3] We do not require the clock functions to be differentiable. We make use of derivatives only as a notational shorthand.

only require clocks belonging to *the same period* to be within Δ of each other. Note that processors may not be able to agree precisely on when a period begins or ends. Let $e_{*,j}$ denote the real time corresponding to the end of the j^{th} period for the slowest non-faulty processor. Thus for $t \in [e_{*,j-1}, e_{*,j}]$ we require

Clock Synchronization condition (CS): $\left| C_{i,j}(t) - C_{k,j}(t) \right| \leq \Delta$

In some of its formulations, the Clock Synchronization Problem also requires a bound on the drift rates of virtual clocks with respect to real time. That is, for some constant ρ' and $t \in [e_{*,j-1}, e_{*,j}]$, the virtual clock of each non-faulty processors p_i should also satisfy:

Bounded Rate condition[4] (BR): $(1-\rho')t \leq C_{i,j}(t) \leq (1+\rho')t$

Recently, Schneider has presented a common framework for studying clock synchronization [Sc86]. In his terminology, the Clock Synchronization Condition specifies the *precision* (Δ) and the Bounded Rate Condition specifies the *accuracy* (ρ') for synchronization protocols.

6.4. A Clock Synchronization Protocol

Imagine the occurrence of an event that is observed by all processors at slightly different times. If these (real) time differences are smaller than the processors' current clock differences, then the processors could resynchronize their clocks based on the time of that event.

The interval of time within a synchronization period during which a processor executes the clock synchronization protocol is called the *synchronization phase*. Suppose the length of each synchronization phase is fixed and known to all processors and the end of a synchronization phase coincides with the end of a synchronization period. Let B and E

[4] Typically BR has been stated as a bound on the adjustment made to a virtual clock during resynchronization. [Sr85c] and [Ma85] use a definition similar definition to ours.

denote the virtual times of the beginning and end, respectively, of a given synchronization phase. At time E, a new virtual clock is started. Let I denote the value to which this clock is to be initialized. Clearly, clock synchronization involves:

(1) Determining B (or equivalently E) and

(2) Determining I

The synchronization problem can be simplified if one of B or I is agreed upon *a priori.* The other variable has to be determined with the cooperation of other processors in the system. For example, suppose B (and thus E) are known to all processors. The value of I can be determined in the following way: at virtual time B each processor broadcasts a message signaling "it is time to synchronize." Based on the times at which a processor receives the broadcast messages from the other processors, it computes I. The messages of a synchronization phase represent the common event, while the "average" of the arrival times of these messages represents the event time. Our protocol is based on this simple approach.

Note that all communication taking place in a synchronization phase is just a DME operation. The messages are used only to signal the processors that they should synchronize, and need not carry an explicit value. Therefore, any DME operation can be used for synchronization. All that is required is that the processors agree in advance which DME operations will be used for synchronization. For example, they could, by convention, use every DME operation to synchronize.

Clocks have to be resynchronized periodically because they may drift with respect to each other with the passage of time. The maximum allowable time between resynchronizations depends on the parameters of the network (μ, d_M, ρ), the number of faulty processors and the maximum clock difference (Δ) that is to be allowed. A DME operation will have to be executed exclusively for synchronization only when this maximum time has elapsed since the last DME. Since DME operations are one of the most basic operations in fault-tolerant distributed systems, the need for an explicit DME operation on behalf of synchronization should arise only when the system is nearly idle.

Let L denote the length of a DME operation for a given network. Since the clocks can be up to Δ units apart, we use the approach described in Section 6.2 and adjust the length of a synchronization phase to be $L+2\Delta$. Thus for any given synchronization phase we have the following relationship

$$E = B+L+2\Delta$$

Each processor p_i broadcasts the message belonging to the j^{th} synchronization phase at virtual time $B_{i,j}+\Delta$.

For processor p_i and j^{th} synchronization phase:

```
at  virtual time B_{i,j}  do
        start collecting arrival times of messages of the j^{th} phase ;
od ;

at  virtual time E_{i,j}  do
        stop collecting arrival times of messages of the j^{th} phase ;
        compute the average A_{i,j} of the message arrival times during the j^{th} phase ;
        C_{i,j+1} := 2E_{i,j}-A_{i,j} ;
        set  B_{i,j+1} and E_{i,j+1} to the next synchronization phase ;
od ;
```

Figure 6.1. Clock Synchronization Procedure

At each synchronization phase, our clock synchronization protocol requires the execution of the procedure shown in Figure 6.1. Since the averaging function ($A_{i,j}$) depends on the communication network used, we postpone its discussion. In the next section we define the conditions for clock synchronization based on this approach. Then, in the following two sections, we describe the averaging function and analyze the performance of our clock synchronization protocol for point-to-point and broadcast networks.

We do not analyze the accuracy of our clock synchronization protocol. Although a bound on its accuracy can be derived, there is a loss of accuracy at each synchronization, and since our protocol is intended to be executed very frequently, it is expected that accuracy will be quite poor. The protocols in [Ha84b, La84b, Lu84] present the same behavior. That is, an increase in precision results in a loss of accuracy. In Section 6.8, however, we show how optimal accuracy can achieved whenever it is needed.

6.5. Conditions for Clock Synchronization

Note that processors may not be able to agree precisely on when a phase begins or ends. Let b_* and e_* denote the real times corresponding to the beginning of a synchronization phase for the fastest non-faulty processor and the end of the same phase for the slowest non-faulty processor, respectively.

Our protocol will guarantee that virtual clocks will be at most Δ_{min} units apart immediately after a synchronization phase (at real time e_* for that phase). The frequency with which resynchronization has to be performed will depend on the maximum clock difference Δ that is to be allowed. Given that the relative drift rate between clocks is at most 2ρ, the minimum real time necessary for any two clocks to increase their differences from Δ_{min} to Δ is

$$\tau = (\Delta-\Delta_{min})\,(2\rho)^{-1}$$

Thus the synchronization protocol has to be executed at least once in each time period PER, where

$$PER \leq (\Delta-\Delta_{min})\,(2\rho)^{-1}(1-\rho) = \tau(1-\rho) \tag{6.1}$$

The term $(1-\rho)$ is necessary to ensure that even the slowest processor will start the next synchronization phase before the real time interval τ has elapsed.

Since ρ is of the order 10^{-6} for typical hardware clocks, quadratic and higher order terms in ρ are negligible and will be ignored for simplicity. For sufficiently small α, we also consider $(1-\alpha\rho) \approx (1+\alpha\rho)^{-1}$ and $(1+\alpha\rho) \approx (1-\alpha\rho)^{-1}$ since they differ by ρ^2 and higher order terms, which are negligible. These simplifications render approximate values for the equations we are to derive and let us concentrate on the dominant values. Using these approximations, equation 6.1 can be expressed as:

$$PER \leq (\Delta - \Delta_{min})\,(2\rho)^{-1} \tag{6.2}$$

If the clocks are at most Δ_* apart at time b_*, the actual duration of a phase $(e_* - b_*)$ may be as long as $L+2\Delta+\Delta_*$. Clearly, the period PER has to be at least as long as $L+2\Delta+\Delta_*$.

Since Δ is to be the maximum clock difference at all times and since the clocks may drift apart at rate 2ρ, the maximum Δ_* should accommodate the possible drift among clocks within the synchronization phase. Next we calculate the maximum allowable Δ_* in terms of ρ, Δ and the phase length. The value of Δ_* plus the drift allowed within the synchronization phase should not exceed Δ. In other words

$$\Delta_* + 2\rho(L+2\Delta+\Delta_*) \leq \Delta$$

After eliminating the ρ^2 terms, the upper bound for Δ_* becomes

$$\Delta_* \leq \Delta - 2\rho(L+3\Delta) \tag{6.3}$$

Recall that $PER \leq L+2\Delta+\Delta_*$. Substituting the maximum value for Δ_* from equation 6.3 into this expression, the minimum period length, PER_{min}, can be stated as

$$PER_{min} = (1-2\rho)(L+3\Delta)$$

We will consider time intervals between resynchronizing clocks as multiples of this minimum period length PER_{min}. In other words, $PER = \eta PER_{min}$ for some positive integer η. Substituting ηPER_{min} for PER in equation 6.2, we derive the feasible values of Δ in terms of L, ρ, η and Δ_{min}:

$$\Delta \geq \frac{2\eta\rho L+\Delta_{min}}{1-6\eta\rho} \tag{6.4}$$

6.6. Clock Synchronization in Point-to-Point Networks

We will now consider a fully-connected point-to-point network with F faulty processors during the synchronization phase. Let k=F/n denote the fraction of processors that are faulty.

Recall that d_m and d_M are the minimum and maximum message delivery times, respectively. In fully-connected point-to-point networks, these times include the message delivery time and the time taken by a processor to prepare messages for each of its n-1 links. As n may be large message preparation time cannot be ignored. For these networks the length (L) of a DME operation is d_M and its adjusted length (as described in Section 6.2) is $d_M+2\Delta$.

We assume that messages are stamped with virtual times as soon as they are received by a processor. This can be easily accomplished either at the network transport layer or by a high-priority user process.

Messages (belonging to the current DME) that arrive within the interval $[B+d_m, B+d_M+2\Delta]$ are said to be *valid messages*. If a processor receives two or more messages from the same processor q within the same DME, then q is clearly faulty and its messages are simply ignored (i.e., treated as invalid messages). The value A is the arithmetic average of the arrival times of all valid messages for the current DME.

We next derive an upper bound for Δ_{min} for a given synchronization phase j. For clarity we shall drop the subscript j from all variables referring to this phase.

Let p_1 and p_2 be two non-faulty processors whose clock differences are such that processor p_1 starts the j^{th} synchronization phase Δ_0 time units ahead of processor p_2. In other words,

$$b_2=b_1+\Delta_0$$

Recall our convention that lower case letters refer to real time whereas the same letters in upper case denote the corresponding clock times. Thus

$$B_1=C_1(b_1) \quad \text{and} \quad b_1=c_1(B_1).$$

As B_1 and B_2 are simply constants known by p_1 and p_2 that have the same value, we shall use B to denote the common value. Similarly we use E instead of E_1 and E_2. Note that the real time corresponding to E (denoted e_i for processor p_i) may be different for distinct processors. The value of e_* that we defined earlier is the maximum of e_i over all i.

Recall that Δ_{min} denotes the maximum clock difference at real time e_* among all processors. Let p_1 and p_2 be the processors whose clocks are the farthest apart at e_*. At this point, however, we do not know which of p_1 or p_2 will be ahead e_* and Δ_{min} can only be expressed as:

$$\Delta_{min}=|C_1(e_*)-C_2(e_*)|$$

Recall from the synchronization procedure description that at virtual time E clocks are initialized to 2E–A. We can express the new clock value of p_1 at real time e_* as:

$$C_1(e_*)=2E-A_1+S_1(e_1,e_*) \tag{6.5}$$

where

$$S_1(x,y)=C_1(y)-C_1(x) \tag{6.6}$$

Next we evaluate p_1's average A_1. Note that A_1 can be expressed as the weighted sum of two terms: one corresponding to the arrival times of the messages broadcast by faulty processors and the other corresponding to the arrival times of the messages broadcast by non-faulty processors. Thus,

$$A_1 = kF_1 + (1-k)G_1$$

where k is the fraction of processors that are faulty, and F_1 and G_1 are the averages of the arrival times of the messages broadcast by the faulty and non-faulty processors, respectively.

The minimum and maximum values for F_1 correspond to the earliest and latest times at which p_1 would accept messages as belonging to the j^{th} synchronization phase. These times are $B+d_m$ and E, respectively , thus

$$B+d_m \leq F_1 \leq E.$$

Note that a faulty processor can send the synchronization messages destined for p_1 and p_2 at different times in an attempt to increase Δ_{min}. Next we analyze G_1.

Without loss of generality we can pretend that all messages from non-faulty processors were received by p_1 at time G_1. Let g_1 be the real time corresponding to G_1. Let G_2 and g_2 be defined analogously for processor p_2. Since the network uncertainty is μ and the processors considered in the G_1 average are non-faulty (and therefore send the synchronization messages to all processors at the same virtual time), then processor p_2 will receive these messages within the real time interval $[g_1-\mu, g_1+\mu]$. We can relate g_1 and g_2 through

$$|g_1-g_2| \leq \mu$$

The value of G_1 can be expressed in terms of g_1 as

$$G_1 = B + S_1(b_1, g_1) \tag{6.7}$$

The average A_1 can be written as

$$A_1=kF_1+(1-k)(B+S_1(b_1,g_1))$$

and therefore:

$$C_1(e_*)=2E-kF_1-(1-k)(B+S_1(b_1,g_1))+S_1(e_1,e_*) \tag{6.8}$$

We now turn our attention to the term $S_1(e_1,e_*)$. The value of e_1 depends on the rate of C_1 in the interval [B,E]. Lets us define a function $s_1(U,V)$ that gives the elapsed real time between the two virtual clock readings U and V. In other words, for processor p_1 $s_1(U,V)=c_1(V)-c_1(U)=v-u$.

Note that

$$s_1(U,U+S_1(u,v)))=s_1(U,V)=v-u\ ,$$

$$S_1(u,u+s_1(U,V))=V-U \quad \text{and}$$

$$\text{if } S_1(u,v)=(v-u)z \text{ then } s_1(U,V)=(V-U)/z \tag{6.9}$$

The term e_1 can be expressed as

$$e_1=b_1+s_1(B,E)$$

Note that the rate of C_1 in the real time interval $[b_1,g_1]$ appears in equation 6.8 as $S_1(b_1,g_1)$. Note also that the interval $[C_1(b_1),C_1(g_1)]$ is contained in [B,E]. If we substitute e_1 as defined above in equation 6.8, it will contain two dependant subexpressions which makes the maximization and minimization of $C_1(e_*)$ difficult. . We solve this difficulty by splitting the [B,E] interval into appropriate subintervals. Thus,

$$e_1=b_1+s_1(B,G_1)+s_1(G_1,E)$$

$$=b_1+s_1(B,B+S_1(b_1,g_1))+s_1(B+S_1(b_1,g_1),E) \qquad \text{(from equation 6.7)}$$

$$=b_1+(g_1-b_1)+s_1(B+S_1(b_1,g_1),\ E)$$

since $s_1(B,B+S_1(b_1,g_1))=g_1-b_1$

$=g_1+s_1(B+S_1(b_1,g_1),E)$

The term $S_1(e_1,e*)$ can be expressed as

$$S_1(e_1,e*)=S_1(g_1+s_1(B+S_1(b_1,g_1),E),e*)$$

and finally $C_1(e*)$ becomes

$$C_1(e*)=2E-kF_1-(1-k)(B+S_1(b_1,g_1))+S_1(g_1+s_1(B+S_1(b_1,g_1),E),e*) \qquad (6.10)$$

The expression for $C_2(e*)$ can be developed in a similar manner as for $C_1(e*)$, giving

$$C_2(e*)=2E-kF_2-(1-k)(B+S_2(b_2,g_2))+S_2(g_2+s_2(B+S_2(b_2,g_2),E),e*)$$

Next we derive the minimum and maximum values for $C_1(e*)$ (and $C_2(e*)$). There are three time intervals in which C_1 can change its rate in an attempt to increase Δ_{min}. These are $[B,G_1]$, $[G_1,E]$ and $[E,C_1(e*)]$. The first interval is referred to twice in equation 6.10 (the first and last (left-to-right) applications of S_1). The last two intervals are referred to only once (the first application of s_1 and the second application of S_1, respectively). The applications of the s_1 and S_1 functions can be substituted for by the corresponding defining expressions in equation 6.9. Recall that the clock rates of non-faulty processors can vary from $(1-\rho)$ to $(1+\rho)$. A simple, but tedious analysis shows that $C_1(e*)$ is maximized when C_1 runs at rate $(1+\rho)$ throughout the synchronization phase and is minimized when it runs at rate $(1-\rho)$. Expressions for the maximum and minimum values of $C_1(e*)$ and $C_2(e*)$ are given below. We have already substituted the s and S functions with the proper clock rates and made the proper choice of the terms F_1 and F_2.

$$\mathit{max}\ C_1(e*)=E-kd_m+k(g_1-b_1)(1+\rho)+(e*-g_1)(1+\rho)$$

$$\mathit{min}\ C_1(e*)=E-k(L+2\Delta)+k(g_1-b_1)(1-\rho)+(e*-g_1)(1-\rho)$$

$$\mathit{max}\ C_2(e*)=E-kd_m+k(g_2-b_2)(1+\rho)+(e*-g_2)(1+\rho)$$

$$min\ C_2(e_*)=E-k(L+2\Delta)+k(g_2-b_2)(1-\rho)+(e_*-g_2)(1-\rho)$$

There are two possible expressions for Δ_{min} from which we have to choose the one that yields the maximum:

$$\Delta_{min1} = max\ C_1(e_*)-min\ C_2(e_*) \quad \text{or}$$

$$\Delta_{min2} = max\ C_2(e_*)-min\ C_1(e_*)$$

We analyze Δ_{min1} first. Substituting the expressions for *max* $C_1(e_*)$ and *max* $C_2(e_*)$ into the above we get

$$\Delta_{min1}=k(L+2\Delta-d_m)+k(b_2-b_1)+(1-k)(g_2-g_1)+[2e_*-(1-k)(g_1+g_2)-k(b_1+b_2)]\rho$$

Recall that $b_2-b_1=\Delta_0$. Hence $b_1+b_2=\Delta_0+2b_1$ and Δ_{min1} can be rewritten as:

$$\Delta_{min1}=$$

$$k(L+2\Delta-d_m)+k\Delta_0+(1-k)(g_2-g_1)+[2e_*-(1-k)(g_1+g_2)-k(\Delta_0+2b_1)]\rho \qquad (6.11)$$

Recall that $|g_1-g_2|\leq\mu$. Clearly, from equation 6.11, Δ_{min1} is maximized when $g_2-g_1=\mu$ and g_1+g_2 is minimum. If $\Delta_0\leq\mu$ then g_1 can be set to its minimum, $g_1=b_1$ and g_2 becomes $\mu+b_1$. The $\Delta_0\leq\mu$ condition guarantees that g_2 is kept within its proper bounds, $b_2\leq g_2\leq e_2$. We replace g_1 and g_2 by these values in equation 6.11 and after algebraic simplifications, the expression for Δ_{min1} when $\Delta_0\leq\mu$ can be written as:

$$\Delta_{min1}(\Delta_0\leq\mu)=k(L+2\Delta+\Delta_0-\mu-d_m)+\mu+[2(e_*-b_1)-(1-k)\mu-k\Delta_0]\rho$$

We turn our attention to the e_*-b_1 term. Clearly $\Delta_{min1}(\Delta_0\leq\mu)$ increases with Δ_0. In other words, $\Delta_{min1}(\Delta_0\leq\mu)$ is maximized when C_2 is the farthest behind and C_1 is the farthest ahead among all processors at the beginning of the synchronization phase. Since C_2 runs at the slowest possible rate throughout the synchronization phase, it is the last clock to reach the end of the phase and therefore $e_*=e_2$. Thus

$$e_*-b_1=e_2-b_1 \qquad \text{(from } e_*=e_2\text{)}$$

$$e_*-b_1=e_2-b_2+\Delta_0 \qquad \text{(from } b_2-b_1=\Delta_0)$$

$$e_*-b_1=(L+2\Delta)(1-\rho)+\Delta_0 \qquad \text{(from } e_2-b_2=(L+2\Delta)(1-\rho))$$

The expression for $\Delta_{min1}(\Delta_0 \leq \mu)$ after eliminating the ρ^2 terms becomes

$$\Delta_{min1}(\Delta_0 \leq \mu)=k(L+2\Delta+\Delta_0-\mu-d_m)+\mu+[2(L+2\Delta)-(1-k)\mu+(2-k)\Delta_0]\rho.$$

If $\Delta_0 \geq \mu$ then g_2 can be set to its minimum, $g_2=b_2$ and g_1 becomes $b_2-\mu$. As $b_2=\Delta_0+b_1$ (from $b_2-b_1=\Delta_0$), g_1 and g_2 can be expressed in terms of b_1 instead of b_2. The $\Delta_0 \geq \mu$ condition guarantees that g_1 is kept within its proper bounds, $b_1 \leq g_1 \leq e_1$. We replace g_1 and g_2 by their values in equation 6.11 and after algebraic simplifications, the expression for Δ_{min1} when $\Delta_0 \geq \mu$ becomes

$$\Delta_{min1}(\Delta_0 \geq \mu)=k(L+2\Delta+\Delta_0-\mu-d_m)+\mu+[2(e_*-b_2)+(1-k)\mu+k\Delta_0]\rho.$$

As before, e_* has to be equal to e_2 and $\Delta_{min1}(\Delta_0 \geq \mu)$, after the ρ^2 terms are eliminated can be expressed as:

$$\Delta_{min1}(\Delta_0 \geq \mu)=k(L+2\Delta+\Delta_0-\mu-d_m)+\mu+[2(L+2\Delta)-(1-k)\mu+k\Delta_0]\rho \qquad (6.12)$$

A similar development yields the following expression for Δ_{min2}. In this case the expression is independent of the relation between Δ_0 and μ.

$$\Delta_{min2}=k(L+2\Delta+\Delta_0-\mu-d_m)+\mu+[2(L+2\Delta)-(1-k)\mu+k\Delta_0]\rho \qquad (6.13)$$

Note that Δ_{min} is always greater than or equal to μ. Thus we will only consider the case where $\Delta_0 \geq \mu$. Clearly, from equations 6.12 and 6.13, $\Delta_{min1}(\Delta_0 \geq \mu) \geq \Delta_{min2}$ for all possible values of k, ρ and μ. Thus, in our subsequent analyses we shall use $\Delta_{min}=\Delta_{min1}(\Delta_0 \geq \mu)$.

Note that $\Delta_0=b_2-b_1$ denotes the real time difference between when p_1 and p_2 begin the synchronization phase. We are interested in expressing Δ_{min} as a function of the clock difference at time b_*. Recall that $\Delta_* = |C_1(b_*)-C_2(b_*)|$ and that the maximum Δ_{min} is achieved when $b_*=b_1$ and Δ_* can be expressed as:

$$\Delta_* = |C_1(b_1) - C_2(b_1)|$$

$$= C_1(b_1) - C_2(b_1) \qquad \text{(since } b_2 > b_1\text{)}$$

$$= B - C_2(b_1)$$

$$= B - (B - S_2(b_1, b_2))$$

$$= S_2(b_1, b_2)$$

$$= (b_2 - b_1)z \qquad \text{where } z \text{ is the rate of } C_2 \text{ within } [b_1, b_2]$$

$$= \Delta_0 z$$

Since we are interested in the smallest Δ_* that implies the largest Δ_{min}, we choose $z=(1-\rho)$ and then

$$\Delta_* = \Delta_0(1-\rho)$$

$$\Delta_0 = \Delta_*/(1-\rho)$$

$$= \Delta_*(1+\rho)$$

The expression for Δ_{min} can be written in terms of Δ_* as:

$$\Delta_{min} = k(L+2\Delta+\Delta_*-\mu-d_m)+\mu+[2(L+2\Delta)-(1-k)\mu+2k\Delta_*]\rho \tag{6.14}$$

Substituting Δ_* by its maximum value as given in equation 6.3, equation 6.14 becomes

$$\Delta_{min} = k(L+3\Delta-\mu-d_m)+\mu+[2(L+2\Delta)-\mu](1-k)\rho. \tag{6.15}$$

A lower bound for Δ can be found by substituting the expression for Δ_{min} above into equation 6.4.

$$\Delta \geq \frac{\mu+k(L-\mu-d_m)+[(2L-\mu)(1-k)+2\eta L]\rho}{1-3k-[6\eta-4(1-k)]\rho} \tag{6.16}$$

Since $L=d_M$ (equation 6.16) can be rewritten as,

$$\Delta \geq \frac{\mu+[2\eta(d_M+3\mu)+(2d_M-\mu)(1-k)]\rho}{1-3k-[6\eta-4(1-k)]\rho} \tag{6.17}$$

clearly, Δ tends to infinity as k tends to 1/3. Thus, our protocol results in diverging clocks as the number of faulty processors approaches 1/3 of the total number of processors. Dolev, *et. al.* [Do84] have shown that clock synchronization has a solution only if less than 1/3 of the total number of processors are faulty when messages are not authenticated and processors may fail maliciously. Equation 6.17 shows that the precision of our protocol is heavily dependent on k and μ since the other terms which are multiplied by ρ, should be several orders of magnitude smaller than k and μ. For comparison, we show below the minimum Δ achieved when the protocol is executed as frequently as possible, that is when $\eta=1$.

$$\Delta \geq \frac{\mu+[d_M(4-2k)-\mu(1-k)]\rho}{1-3k-(4+4k)\rho}$$

The network uncertainty can be decreased if the system is such that non-faulty processors always send the messages of DME operations to the other processors in the same order and the ordering is known by all processors. If the i^{th} processor in this ordering receives a message at time T it can correct the receiving time to T–iY where Y is the time to prepare a message and present it to the link. In this way we can compensate for the time differences resulting from the sending order.

Our protocol also tolerates communication failures. Consider links that may corrupt the contents of messages, generate spurious messages, lose, delay or even (unrealistically) anticipate messages. Although the protocol that executes the DME operation may not be able to tolerate link failures of this type, our clock synchronization procedure reamins valid. While a faulty link can corrupt messages between the two processors it connects, a malicious processor can send incorrect messages to all n–1 processors to which it is connected. Clearly a processor failure is more severe than a link failure. Next we show that 2m−1 faulty links are equivalent to m faulty processors with respect to the worst case analysis of our clock synchronization protocol.

A faulty link can transform a message so that it appears as if the sender were faulty. Therefore, the two processors connected by a faulty link receive "incorrect messages" from each other. Recall that p_1 and p_2 denote the two processors whose clocks are farthest apart at real time $e*$. Therefore, the only link failures that can result in a larger Δ_{min} value are those that are incident to p_1 or p_2.

Let f and λ denote the actual number of faulty processors and links, respectively. Let p_1 and p_2 be connected to λ_1 and λ_2 faulty links, respectively. Now, from the point of view of p_1, this system is equivalent to one containing $f+\lambda_1$ faulty processors. Similarly, for p_2 the system has $f+\lambda_2$ faulty processors. Thus the contribution of messages from faulty processors to the average computed by p_1 is $(f+\lambda_1)/n$ while it is $(f+\lambda_2)/n$ to that computed by p_2. In the worst case, these contributions will be in opposite directions. Thus, the differece in the two averages will be equivalent to that of a system having a total of $f+\lfloor(\lambda_1+\lambda_2)/2\rfloor$ faulty processors. Since $\lambda_1+\lambda_2$ can be as large as $\lambda+1$ (the faulty link connecting p_1 and p_2 counts twice), the fraction of faulty processors becomes

$$k= \frac{f+\lfloor(\lambda+1)/2\rfloor}{n}$$

6.7. Clock Synchronization in R-redundant Broadcast Networks

The systems considered in this section consist of any n-processor R-redundant broadcast network with at most F malicious processors. The failures of the communication media are restricted to omission as described in Section 2.2. The network has broadcast degree b and unit survivor diameter. We require that the maximum number of faulty processors satisfy

3F<b. As before, let d_m and d_M denote the minimum and maximum message delivery times, respectively, and μ the network uncertainty defined as $d_M - d_m$. The length of a DME operation is $L = nd_M$ time units.[5] Given that the protocol is to ensure a precision of Δ, the adjusted length of a DME operation, as given in Section 6.2, is $L+2\Delta$.

We assume that all of the processors that receive a message over the same channel receive it at exactly the same real time. This assumption can be relaxed if each processor knows its relative position in the channel with respect to the other processors. If processor p_i receives a message at time T from processor p_j and knows that their distance is $d_{i,j}$, then p_i can correct the receiving time to $T - \tau d_{i,j}$ where τ is the signal propagation speed of the channel. Rom [Ro84] gives an algorithm through which processors can determine their relative positions on a broadcast network.

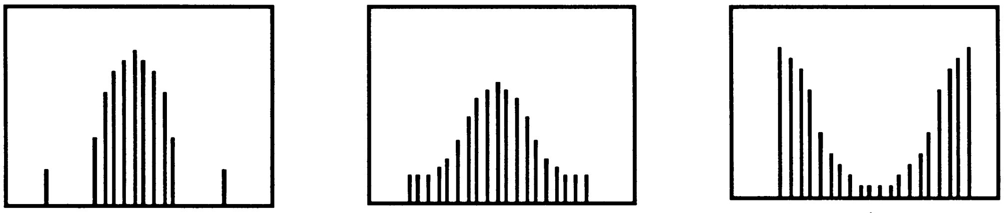
Next we describe the averaging function $A_{i,j}$ of processor p_i for the j^{th} synchronization phase. Note that if the network redundancy is R, processor p_i may receive up to R messages from each p_w. For each processor p_w from which p_i receives a valid message, it chooses as the arrival time the time of the first message to be received from p_w. If p_w is faulty, p_i may receive two or more messages from p_w over the same channel in the same phase. In this case, the messages from p_w can be ignored since p_w is clearly faulty. Also, if p_i receives two messages from p_w belonging to the same synchronization phase such that they arrive more than $\mu(1+\rho)$ time units apart as measured by p_i's virtual clock, it must be the case that p_w is faulty. Because, otherwise, p_w would have broadcast through all R channels at the same time and its messages would arrive at any processor within $\mu(1+\rho)$ time units as measured by the receiving processor. All messages from such processors are simply ignored. Finally, $A_{i,j}$ is the simple average of the arrival time as defined above.

5 There are channel access protocols in which the length of a DME is very close to d_M.

Consider a R-redundant broadcast network with broadcast degree b. For any value $b<n$ we will argue that such a network is equivalent to a point-to-point network as far as clock synchronization is concerned. The equivalence is in the sense that the expression for the best precision attainable in both systems, Δ (equation 6.16), will be the same, albeit with different values for the system parameters such as d_m, d_M, ρ and L.

Our argument is informal. Let $b=n-1$. In such a system, at most one processor will receive a broadcast by a non-faulty processor over a channel different from the other processors. However, it is possible for one processor to not receive a broadcast by a faulty processor while the remaining $n-1$ processors do receive it. Let the f faulty processors fail by performing two broadcasts during the same synchronization phase — one as early as possible and the other as late as possible. Since the broadcast degree is $n-1$, a processor, say p_1, could fail to receive all of the late messages and another processor, say p_2, could fail to receive all of the early ones. The remaining $n-2$ processors receive both broadcasts during the same synchronization phase and (correctly) conclude that each of the broadcasters is faulty and ignore its messages. Consequently, the faulty processors have succeeded in causing p_1 and p_2 to compute average arrival times that are as far apart as they could have been in a point-to-point network. For broadcast degrees smaller than $n-1$, the situation is similar; only now the faulty processors have more flexibility in their behavior.

While the precision achievable by our synchronization scheme is independent of the broadcast degree of the network, the *variance* of the virtual clock values right after a resynchronization is not. Suppose we divide the time axis into non-overlapping intervals of some short, constant length (say μ). Right after a resynchronization, we count the number of virtual clocks that have values within each of the intervals. The *spread* of such a histogram (the difference between the largest and smallest clock values) is proportional to the precision of the synchronization scheme. Informally, we define the *variance* of a clock synchronization scheme as a measure of the distribution of clock values within the bounds that establish the spread. The more clustered together the clock values are, the smaller the variance. The more spread out the clock values are, the greater the variance. In Figure 6.2 we illustrate the variance of our clock synchronization scheme when operated in three networks of decreasing broadcast degree.



(a) (b) (c)

Figure 6.2. Clock Distributions for Three Different Broadcast Degrees

Figure 6.2 (a) depicts a system with a large broadcast degree (such as n−1). As we discussed earlier, such a system may have a large spread but it is due to a small number of processors at both extremes. A majority of the processors have clock values that are close together. Figure 6.2 (b) represents a system with some intermediate broadcast degree (2<b<n−1) and consequently an intermediate variance. Finally, Figure 6.2 (c) depicts a system with a small broadcast degree (such as b=2 as it is for a point-to-point network). Here, the variance could be large since it is possible for processors to be divided into two groups — those with clock values close to the slowest clock, and those with clock values close to the fastest clock.

For some applications it may suffice for a clock synchronization protocol to guarantee only that at least some of the processors have clocks within Δ units of each other. For such applications, broadcast networks with large broadcast degrees offer efficient solutions with high precision.

6.8. Achieving High Accuracy

Srikanth and Toueg [Sr85a] have shown ρ to be a lower bound for ρ', the accuracy of virtual clocks. That is, no synchronization algorithm can guarantee that virtual clocks will have a smaller drift rate with respect to real time than the underlying hardware clocks. Srikanth and Toueg [Sr85a] also present authenticated and non-authenticated synchronization protocols that achieve this optimal accuracy.

In this section we show how virtual clocks that satisfy only the Clock Synchronization condition (CS) in conjunction with correct hardware clocks, can substitute for virtual clocks that satisfy both the CS and BR conditions. We will show that this substitution is valid for the following domain of timing needs:

> *Interval Timing* (IT): Each processor must be able to measure the passage of time during an interval with some accuracy ρ'.
>
> *Real Time Scheduling* (RTS): Each processor must be able to act at a specific real time, with some accuracy ρ'.
>
> *Approximate Simultaneity* (AS): A collection of processors must be able to act at the same real time, with some precision Δ.
>
> *Approximate Timed Simultaneity* (ATS): A collection of processors must be able to act at the same specific real time with some accuracy ρ' and with some precision Δ.

Typical applications of timing systems that require each of these properties are given below. IT allows processors to measure time-out intervals in response to a local or external event. Given RTS, a processor can be scheduled to perform operations ahead of time, such as "backup disk unit A at 00:00 on 2/21/86." If AS is satisfied, an update to a replicated database can be performed by the participating sites such that the system is in an inconsistent state for only a known short time interval. If the distributed update is to be performed at a given real time, such as "credit Sam's account by $1000.00 at 00:00 on 2/29/86," then ATS must be satisfied.

We conjecture that a system exhibiting these four properties is equivalent to a system possessing virtual clocks that achieve both CS and BR.

Virtual clocks satisfying both CS and BR trivially achieve all of the above properties: The BR condition guarantees IT and RTS. For AS and ATS, either all processors know in advance the time at which they must act, or this time is disseminated (or agreed upon, in case fault-tolerance is sought) earlier. Then CS ensures AS and the combination of BR and CS ensures ATS.

We will now prove our claim that hardware clocks satisfying HC and virtual clocks satisfying only CS are sufficient to realize all four timing properties. Clearly, properties IT and RTS are satisfied through the hardware clocks with optimal accuracy ($\rho'=\rho$). If all processors know in advance the time t_X at which some action X is to be performed, then AS is satisfied simply by having the processors schedule X at time t_X on their virtual clocks. If t_X is known only to a single processor, it can broadcast this value to all other processors. If fault-tolerance is sought, then t_X should be disseminated through a Reliable Broadcast. These procedures are exactly the same as those for systems with virtual clocks satisfying both CS and BR. In both systems, the precision achieved is simply the precision of the virtual clocks. In our system, ATS can be satisfied by protocol S1 of Figure 6.3 when t_X is known *a priori* to all processors

at hardware clock time $t_X - (1-\rho)\, d_m$ **do**
 if not yet broadcast "Let's do X" **then**
  broadcast(Let's do X);
 fi;

when F+1 distinct "Let's do X" messages are received **do**
 if not yet broadcast "Let's do X" **then**
  broadcast(Let's do X);
 fi;

when 2F+1 distinct "Let's do X" messages are received **do**
 perform action X;
od;

Figure 6.3. Protocol S1: Approximate Timed Simultaneity

The next 3 lemmata prove various properties of protocol S1 in systems where there are at most $F<n/3$ malicious processors, and hardware clocks have accuracy ρ. We assume that the broadcaster of a message always receives its own broadcast d_m units after the broadcast, as measured on its own clock.

Lemma 6.1. *For protocol S1, the real times at which non-faulty processors perform X differ by at most* $\mu+d_M$ *time units.*

Proof: Let p be the first non-faulty processor to receive 2F+1 "Let's do X" messages. At least F+1 of those messages must be from non-faulty processors. Thus, all processors receive at least F+1 messages within μ time units of the time that p receives the $2F+1^{st}$ message. The non-faulty processors that have not yet broadcast "Let's do X" will do so when they receive the $F+1^{st}$ message. These last messages may take up to d_M time units to be received by all other processors. Thus, all processors receive their $2F+1^{st}$ messages and perform X within $\mu+d_M$ real time units *after* p performs X. ∇

Lemma 6.2. *For protocol S1, there exists a non-faulty processor* p_i *such that no non-faulty processor performs X before real time* $h_i(t_X)$.

Proof: Since F+1 messages must be received by some processor p_j for it to broadcast "Let's do X" before $h_j(t_X-(1-\rho)d_m)$, it must be the case that for at least one non-faulty processor p_i its hardware clock reads no earlier than $t_X-(1-\rho)d_m$. This processor p_i must have broadcast "Let's do X" such that its clock reads t_X or later when the message is received by any other processor. As 2F+1 messages must be received before a processor performs X, no non-faulty processor performs X before $h_i(t_X)$. ∇

Lemma 6.3. *For protocol S1, there exists a non-faulty processor* p_i *such that no non-faulty processor performs X after real time* $h_i(t_X+(1+\rho)\mu+2\rho d_m)$.

Proof: Let p_L be the last processor to perform X. Let t_L denote the time when p_L performs x. We proceed by case analysis.

If $h_L(t_X)\geq t_L$, that is, p_L performs X before its (hardware) clock reads t_X, then the Lemma is trivially satisfied by choosing p_L as p_i. In the remaining case $h_L(t_X)<t_L$.

For contradiction, suppose that the Lemma is false. That is, there exists a non-faulty processor p_L that performs X at real time t_L such that $t_L>h_i(t_X+(1+\rho)\mu+2\rho d_m)$ for all non-faulty p_i. Since by the description of the protocol, no non-faulty processor p_i broadcasts "Let's do X" later than real time $h_i(t_X+(1-\rho)d_m)$,it follows that processor p_L receives at least 2F messages (from non-faulty processors) no later than $h_i(t_X+(1-\rho)d_m)+d_M$. Next we show that $t_L>h_i(t_X+(1-\rho)d_m)+d_M$ for all non-faulty p_i. Given our supposition, it is enough to show that for all non-faulty processors p_i:

$$h_i(t_X+(1+\rho)\mu+2\rho d_m)\geq h_i(t_X-(1-\rho)d_m)+d_M$$

$$h_i(t_X+(1+\rho)\mu+2\rho d_m)-h_i(t_X-(1-\rho)d_m)-d_M\geq 0$$

$$(t_X+(1+\rho)\mu+2\rho d_m-(t_X-(1-\rho)d_m))z-d_M\geq 0$$

$$((1+\rho)\mu+2\rho d_m+(1-\rho)d_m)z-d_M\geq 0$$

where z is the rate of h_i in the time interval $[t_X-(1-\rho)d_m, t_X+(1+\rho)\mu+2\rho d_m]$. Thus

$$(1+\rho)(d_m+\mu)z-d_M\geq 0$$

Note that for any value z within $[(1-\rho), (1+\rho)]$ this inequality is satisfied. Therefore processor p_L receives at least 2F+1 "Let's do X" messages before real time t_L. According to the protocol description p_L performs X before time t_L, a contradiction. ∇

Theorem 6.1. *In a system with at most* $F<n/3$ *malicious processors and hardware clocks that have accuracy* ρ, *protocol S1 implements the approximate timed simultaneity property at time* t_X *with precision* $(d_M+\mu)$ *and accuracy* $\rho'=\rho+\varepsilon$ *where* $\varepsilon\to 0$ *as* $t_X\to\infty$.

Proof: By Lemma 6.1 protocol S1 achieves precision $d_M+\mu$. By Lemma 6.2, protocol S1 presents no lack of accuracy (beyond ρ) by processors acting earlier (since all non-faulty processors act no earlier than $h_i(t_X)$ for some non-faulty p_i). By Lemma 6.3 no non-faulty processor acts later than $h_i(t_X+(1+\rho)\mu+2\rho d_m)$ for some non-faulty processor p_i. If p_i is the slowest non-faulty processor and was running at the slowest possible rate of $(1-\rho)$, then $h_i(t)=(1+\rho)t$[6]. Consequently, no processor acts later than real time $(1+\rho)(t_X+(1+\rho)\mu+2\rho d_m)$. Thus the resulting accuracy can be calculated as follows.

$$(1+\rho')t_X=(1+\rho)(t_X+(1+\rho)\mu+2\rho d_m)$$

$$\rho'=\rho+\frac{(1+2\rho)\mu+2\rho d_m}{t_X}$$

Since $(1+2\rho)\mu+2\rho d_m$ is a constant, $((1+2\rho)\mu+2\rho d_m)/t_X\to 0$ as $t_X\to\infty$. ∇

Note that in Protocol S1 if the processors were scheduled to broadcast "Let's do X" at $t_x-(1-\rho)d_m-((1+2\rho)\mu+2\rho d_m)/2$, then the processors could perform X within the real

---

[6] Note that if $H(t)=(1-\rho)t$ then $h(T)=(1+\rho)T$ since we assume $(1-\rho)^{-1}=(1+\rho)$.

time interval $[h_i(t_x-((1+2\rho)\mu+2\rho d_m)/2), h_i(t_x+((1+2\rho)\mu+2\rho d_m)/2)]$ for some non-faulty processor p_i. Thus the inaccuracy introduced by Protocol S1 could be halved.

In [Sr85a] the condition for optimal accuracy of virtual clocks is stated as follows

Optimal Accuracy condition: $(1-\rho)t+a \leq C_{i,j}(t) \leq (1+\rho)t+b$

where a and b are constants. With respect to this definition, Protocol S1 is optimal.

Note that in Protocol S1 there is no need for virtual clocks and the precision achieved $(\mu+d_M)$ is quite good since $\mu/2$ is known to be a lower bound for the precision achieved by any clock synchronization protocol [Do84].

Protocol S1 can also be used to implement clock synchronization. It suffices for the processors to know *a priori* when the next synchronization begins and the time to which they initialize their new virtual clocks. At the j^{th} synchronization period the action X to be performed becomes "start the j^{th} clock with the predetermined initial value of the j^{th} period." This protocol achieves precision $(\mu+d_M)+2\rho L$ where L is the length the synchronization periods. This precision is independent of the number of faulty processors provided that the total number of such processors is less than a third of the total number of processors. With respect to accuracy it presents the same behavior we mentioned in Section 6.4 where there is a cumulative loss of accuracy at each synchronization phase.

6.9. Conclusion

We have shown that for many distributed systems engaged in fault-tolerant computing, clock synchronization need not be implemented as an explicit protocol. The message exchange that occurs as part of the distributed computation can be exploited to synchronize clocks as a byproduct. Consequently, if the computation generates DME's with sufficient frequency, clocks can be kept synchronized at no additional message cost. DME's explicitly for clock synchronization have to be generated only if the computation itself is silent. In this case the communication network is already idle, and the additional message traffic does not steal bandwidth away from another need.

The precision of our synchronization scheme is inferior to most other proposals. We cannot be too demanding given that our scheme is cost free. If the network is such that the uncertainty in message delivery times is small, the precision attained is probably good enough for most applications.

In formulating our synchronization scheme for broadcast networks, we were surprised to find that precision is independent of the network broadcast degree (except for the special case when b=n). This is a consequence of the definition for precision. All that is required for a synchronization protocol to have poor precision is for two non-faulty processors to have widely differing clocks. The possibility that the remaining processors have clocks that are close together is immaterial. In fact, we have argued that increasing broadcast degrees for the network results in a distribution of clock values with a smaller variance.

While we have not explored them, there is no reason why our scheme could not be used in conjunction with other synchronization protocols to obtain better precision. For example [Do83a] [Ma85] select only some restricted sets of values to be included in the average computation. This technique, if applied to our selection of message arrival time, will diminish the influence of faulty processors and consequently enhance precision.

Finally, we have shown that a clock synchronization protocol need not be obliged to achieve the Bounded Rate Condition. Given a protocol that ensures only the Synchronization Condition, we demonstrated how the properties of the Bounded Rate Condition could be simulated on demand. As most applications typically rely on the Bounded Rate Condition much less frequently than the Synchronization Condition, requiring a clock synchronization protocol to achieve both is inefficient.

CHAPTER 7

Conclusions

Previous research on the Reliable Broadcast (RB) problem has assumed models of communication based on point-to-point networks. In this thesis, we characterized the RB problem in distributed systems with respect to the underlying communication model. By investigating network models based on broadcast channels, we obtained efficient solutions to the RB problem that are not possible with a point-to-point model.

Communication models that lead to efficient solutions to the RB problem are readily implemented in practice by common networks such as Ethernets. Furthermore, typical uses of point-to-point networks have been for long-haul data communication (e.g. Arpanet), which are unlikely to be used as the basis for fault-tolerant distributed computing.

	Omission Processor Failures			**Malicious Processor Failures**		
	n	time lower bound	known time upper bound	n	time lower bound	known time upper bound
Point-to-Point Networks	F+1	F+1	F+1	3F+1	F+1	2F+1
R-RBN	F+L	2	2	2(F+L)+1	2	2

We developed the concept of an R-redundant broadcast network that enables a 2-round solution to the RB problem. In contrast, any solution that tolerates F processor failures in a point-to-point network requires F+1 rounds. Table 7.1 summarizes the relevant properties of our solution to the RB problem in R-redundant broadcast networks and compares them to their counterparts for solutions in point-to-point networks.

Two parameters, the broadcast degree and the survivor diameter, were used to characterize all common network architectures, including point-to-point and broadcast networks. We developed a protocol that optimally solves the RB problem for all networks characterized by these two parameters. The time complexity of this protocol varies with the values of these parameters. This behavior is at the root of the trade-offs between resiliency, performance and network cost that are possible. Thus, by characterizing the complexity of RB with respect to the communication model, we have added a new dimension to the design space of fault-tolerant distributed systems.

Finally, we showed that it is possible to synchronize clocks as a byproduct of communication that is generated for other purposes. Therefore, clock synchronization need not be a separate activity requiring additional messages.

REFERENCES

[Ab73] Abramson, N. The ALOHA system. In *Computer-Communication Networks*, N. Abramson and F. Kuo (eds.). Prentice Hall, Englewood Cliffs, N.J., 1973.

[Aw84] Awerbuch, B. and Even, S. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada (August 1984), 278-281.

[Ba85a] Babaoglu, Ö. and Drummond, R. Streets of Byzantium: Network architectures for fast reliable broadcast. *IEEE Transactions on Software Engineering SE-11*, 6 (June 1985), 546-554.

[Ba85b] Babaoglu, Ö. and Stephenson, P. Reliable broadcasts through partial broadcasts. Technical Report TR 85-720, Cornell University, Ithaca, New York (November 1985).

[Ba86] Babaoglu, Ö. On the Reliability of Fault-Tolerant Distributed Computing Systems. Technical Report TR 86-738, Cornell University, Ithaca, New York (February 1986).

[Br84] Broder, A., Dolev, D., Fischer, M., Simons, B. Efficient fault tolerant routings in networks. In *Proceedings of the 16th ACM Symposium on Theory of Computing* (1984), 536-541.

[Br85] Bracha, G. An $O(\lg n)$ expected rounds randomized Byzantine generals protocol. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, Providence, Rhode Island (May 1985), 316-326.

[Ch84] Chang, J.M. and Maxemchuk, N.F. Reliable broadcast protocols. *ACM Transactions on Computer Systems 2*, 3 (August 1984), 251-273.

[Co85] Coan, B. Private Communication (1985).

[Cr84] Cristian, F., Aghili, H., Strong, R. and Dolev, D. Atomic Broadcasts: From simple message diffusion to Byzantine Agreement. Research Report RJ 4540 (48668), IBM Research Lab., San Jose, California (December 1984).

[Di76] Diffie, W. and Hellman, M. New directions in cryptography, *IEEE Transactions on Information Theory IT-22* (1976), 644-654.

[Do82a] Dolev, D., Reischuck, R. and Strong, H.R. Bounds on Information Exchange for Byzantine Agreement, *In Proceedings of the 2nd Symposium on Principles of Distributed Computing* (1982).

[Do82b] Dolev, D., Reischuck, R. and Strong, H.R. "Eventual" is earlier than "immediate." *In Proceedings of the 23rd Symposium on Foundations of Computer Science*, Chicago, Illinois (November 1982), 196-203

[Do83a] Dolev, D., Lynch, N., Pinter, S., Stark, E. and Weihl, W. Reaching approximate agreement in the presence of faults. In *Proceedings of the 3rd IEEE Symposium on Distributed Software and Database Systems* (1983).

[Do83b] Dolev, D. and Strong, H.R. Authenticated algorithms for Byzantine Agreement, *SIAM Journal on Computing 12*, 4 (November 1983), 656-666.

[Do84] Dolev, D., Halpern, J., and Strong, R. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of the 16th ACM Symposium on Theory of Computing* (April 1984), 504-511.

[Fa73] Farber, D., Feldman, J., Heinrich, F., Hopwood, M., Larson, K., Loomis, D. and Rowe, L. The distributed computing system. In *Proceedings of the CompCon 73*, IEEE Computer Society (February 1973), 31-34.

[Fi82] Fischer, M. and Lynch, N. A. A lower bound for the time to assure interactive consistency. *Information Processing Letters 14*, 4 (April 1982) 183-186.

[Fi83a] Fisher, M.J. The consensus problem in unreliable distributed systems (a brief survey). Technical Report YALEU/DCS/RR-273, Computer Science Department, Yale University (June 1983).

[Fi83b] Fisher, M.J., Lynch, N.A. and Paterson, M.S. Impossibility of distributed consensus with one faulty process. In *Proceedings of the 2nd ACM Symposium on Principles of Database Systems* (March 1983).

[Ga84] Garcia-Molina, H., Pitelli, F., and Davidson, S. Applications of Byzantine agreement in database systems. Technical Report TR-316, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey (June 1984).

[Ha69] Harary, F. *Graph Theory*, Addison Wesley, 1969.

[Ha84a] Hadzilacos, V. Issues of fault tolerance in concurrent computations. Ph.D Thesis, Technical Report TR-11-84, Aiken Computation Laboratory, Harvard University, Cambridge, Mass. (June, 1984).

[Ha84b] Halpern, J.Y., Simons, B., Strong, H.R. and Dolev, D. Fault-tolerant clock synchronization. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada (August 1984), 89-102.

[Ki84] Kim, W. Highly available systems for database applications, *ACM Computing Surveys 16*, 1 (March 1984), 71-98.

[La82a] Lamport, L. and Melliar-Smith, P.M. Synchronizing clocks in the presence of faults. Technical Report, Computer Science Laboratory, SRI International (March 1982). Also in *Journal of the ACM 32*,1 (January 1985), 52-78.

[La82b] Lamport, L., Shostak, R. and Pease, M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 382-401.

[La84a] Lamport, L. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems 6*, 2 (April 1984), 254-280.

[La84b] Lamport, L. and Melliar-Smith, P.M. Byzantine clock synchronization. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada (August 1984), 68-74.

[Le81] Lelann, G. A distributed system for transaction processing. *IEEE Computer 14*, 2 (February 1981), 43-48.

[Lu84] Lundelius, J. and Lynch, N. A new fault-tolerant algorithm for clock synchronization. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada (August 1984), 75-88.

[Ma83] Marzullo, K. Loosely-coupled distributed services: a distributed time service. Ph.D. thesis, Stanford University (1983).

[Ma85] Mahaney, S.R. and Schneider, F.B. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings of the 4th Symposium on the Principles of Distributed Computing*, Minaki, Canada (August 1985).

[Me76] Metcalfe, R. and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM 19*, 7 (July 1976), 396-403.

[Mo83] Mohan, C., Strong, H.R. and Finkelstein, S. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada, August 1983), 89-103.

[Pe80] Pease, M., Shostak, R. and Lamport, L. Reaching agreement in the presence of faults. *Journal of the ACM 27*, 2 (April 1980), 228-234.

[Ri78] Rivest, R.L., Shamir, A. and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (February 1978), 120-126.

[Ro84] Rom, R. Ordering subscribers on cable networks, *ACM Transactions on Computer Systems 2*,4 (November 1984), 322-334.

[Sc82] Schneider, F.B. Synchronization in distributed programs. *Transactions on Programming Languages and Systems 4*, 2 (April 1982), 125-148.

[Sc84] Schneider, F.B. Byzantine generals in action: Implementing fail-stop processors, *ACM Transactions on Computer Systems 2* , 2 (May 1984), 145-154.

[Sk82] Skeen, M.D. Crash recovery in a distributed database system, Ph.D. dissertation, Computer Science Division, University of California, Berkeley, CA (1982).

[Sp84] Spector, A.Z. Computer software for process control, *Scientific American, 251*,3 (September 1984), 174-187.

[Sr85a] Srikanth, T.K. and Toueg, S. Byzantine agreement made simple: Simulating authentication without signatures. In *Proceedings of the 4th Symposium on the Principles of Distributed Computing* , Minaki, Canada (August 1985), 71-86.

[Sr85b] Srikanth, T.K. Design of fault-tolerant algorithms for distributed systems using communication primitives, Ph.D. dissertation. Cornell University, Ithaca, New York (1985).

[St84] Stallings, W. Local networks, *ACM Computing Surveys 16*, 1 (March 1984), 3-41.

[Ta81] Tanenbaum, A. *Computer Networks*. Prentice Hall, Englewood Cliffs, N.J., 1981.