ORDERED AND RELIABLE MULTICAST COMMUNICATION

Annemarie Spauster
(Thesis)

CS-TR-312-91

June 1991

Ordered and Reliable Multicast Communication

Annemarie Spauster

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JUNE 1991

To my mother, Anne Marie Spauster

Acknowledgments†

I am very grateful to my adviser, Hector Garcia-Molina, who not only taught me how to do research, but also taught me how to write about, speak about it, criticize it and publish it. I also thank him for always treating me with respect and consideration.

I would like to thank Rafael Alonso for reading my thesis and providing useful comments, especially regarding clarifications in Chapter 2. I also appreciate the time he has given me throughout my graduate career for discussing research and career options.

I also thank Kai Li for reading my thesis. He provided several references for related work and helped clarify the presentation of the material. I'd also like to thank Kai for giving me the opportunity to spend a summer at DEC SRC.

Daniel Barbara gave me guidance on my first attempt at research as a first year student. His kindness and spirit made the effort fun and productive. I'd also like to thank him for some ideas regarding the topology section of Chapter 4. Diane Souvaine, my office mate, was very supportive in my first few years at Princeton.

The support staff at Princeton has always been outstanding. Pat Parseghian, Steve Beck and Sharon Rodgers have always provided especially enthusiastic help.

I thank all the friends I've had at Princeton. I am especially glad to have had the opportunity to be a Cache Hitter.

I would like to thank a number of people at Smith College. My colleagues, Joe O'Rourke, Dominique Thiebaut, Bob Roos and Merrie Bergmann, have shown support and tact regarding my efforts to finish my thesis. A couple of students, Sihame Kairouani

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

and Jacqueline Lie, have made my first two years at Smith a pleasant and worthwhile time.

A few good friends have always been there for me: Jacky Bloom, Nancy Ketterer and Carol Kilpatrick.

My heart belongs to Alan B. Evans, who has stayed with me through thick and thin, good and bad and times of uncertainty. I thank him for his love, patience and constant encouragement. I anticipate with great joy the arrival of our baby.

I would be nothing without the help and love of my family. I think the world of my two brothers, Edward and Robert, and know I can always count on them. I appreciate the warmth and love I get from my sisters-in-law, Annie and Anne, my niece and nephew, Tina and Robert, Jr. and from Rachael Evans and Keelan Evans.

Most of all, I thank my mother, Anne Marie Spauster, to whom this work is dedicated. No one has taught me more than she has. Her example of strength, optimism and self-reliance has made me believe I can accomplish anything. Her constant support and encouragement has got me where I am today.

Abstract

Multicasting (sending a message to a subset of sites in a network) has become a popular mechanism for interprocess communication in distributed systems. Many applications (e.g., distributed databases) require that a message be transmitted to multiple processes. One indisputably desirable quality of any type of message transmission is reliability. A message that is sent from process A to process B should indeed arrive. Even better, it should arrive within a reasonable amount of time.

For distributed applications, it is also often desirable for multidestination messages to arrive at the destination processes in a consistent order. In a database application, if update requests are headed to two destinations with copies of the data, delivering the requests in the same order at both destinations helps maintain consistency. This is just one example of how consistent message delivery simplifies distributed applications.

In this thesis, we consider enhancing multicast communication by providing ordering and reliability properties. We present an algorithm, the *propagation graph algorithm*, that guarantees a strong ordering property: *multiple group ordering*. Experimental analysis of the propagation graph technique demonstrates that it is efficient compared to other solutions and exhibits a clear load/delay tradeoff. We also present several types of reliability that multicast ordering algorithms can provide. We address how to achieve these types of reliability with the propagation graph algorithm. Further, we clarify the issue of reliability by presenting a formal model of message ordering and by presenting formal definitions of reliability properties. These properties are then applied to the propagation graph solution.

Contents

Chapter One

Introduction

The basis for any distributed system is the ability of its member sites to communicate. Communication is accomplished by using established protocols on an underlying network. The network itself comes in many varieties, each providing its own special features; they are broadly classified into two categories: broadcast and point-to-point. Broadcast networks (e.g., Ethernet, token ring) allow each attached site to hear a message put on the channel. Point-to-point networks (e.g., Internet) generally allow only single-site to single-site communication. Messages from one site to another site to which it is not directly connected must be routed through intervening sites. Common now is the internetwork, which connects in a point-to-point fashion local-area (typically broadcast) networks.

Often, the physical form of communication provided by the network (i.e., broadcast or point-to-point) is not the one or only type of message transmission desired by the user. For this reason, protocols run in hardware and/or software are employed to effect various forms of logical communication. The simplest logical message type is a unicast: a message from one site to one other. Point-to-point networks provide this already; broadcast networks typically provide a hardware filter (channel interface) to weed out messages that sites are not interested in. The second type of logical communication is broadcast. A broadcast is a message that a sender wants all sites to receive. A broadcast network supplies this readily (no site filters this message out); point-to-point networks require special routing techniques.

Many protocols have been suggested for routing broadcast messages on a point-to-point network. One simple choice is just to have the source send a separate message to each site. Another possibility is to flood the network with the message. Any site that gets the message sends it to every site with which it communicates directly. Multidestination addressing sends information along with a message telling the recipient which sites to pass it on to. Destinations along the same route share message copies. Tree forwarding uses a spanning tree (e.g., a minimum spanning tree) of the network to forward the message. [FWB85] describes and compares these techniques using ratings of nil, low, medium, high and gross (best to worst) on the following criteria: bandwidth, delay, state information, computation, et. al. Separately-addressed messages require high bandwidth and incur high delay. The bandwidth associated with flooding is rated as gross, but delay is medium. Multidestination addressing consumes medium bandwidth and incurs medium delay, but variable-sized packets make this technique hard to implement. Tree forwarding consumes low bandwidth with medium delay.

A new local-area point-to-point network called Autonet has been implemented by a group at Digital's Systems Research Center [Schr90]. Forwarding tables at each switch route packets. Broadcast is supported by flooding packets on a spanning tree of links. The spanning tree is computed dynamically as the topology of the network changes.

The most general form of logical communication is a multicast - the transmission of a message from one site to a subset of sites in the system. (Unicasts and broadcasts both qualify as multicasts.) Both types of networks can employ protocols to multicast.

Multicasting on a point-to-point network can be handled using the broadcast methods. Separately-addressed messages may be a practical alternative for multicasts to a small subset of sites. Flooding and multidestination addressing also work, as do spanning trees. For efficiency, a different spanning tree can be used for each subset of sites, but this can be difficult to maintain. These techniques are addressed in [FWB85]. [DC90] addresses multicast routing in internetworks and presents several extensions to common routing algorithms (distance-vector routing and link-state routing) to provide multicast routing. [McKL90] uses multicast trees to route multicasts in bus-based networks.

Multicasting on a broadcast network can be handled in several ways. Often logical multicast group addresses can be hardwired into the network, so that the network interface can determine if it should pass the message to its corresponding site. If logical addresses are not implemented in hardware, multicasts can be accepted at all interfaces, leaving it up to software to weed out unwanted messages. (Logical group addresses checked in hardware are supported on the Ethernet, though applications often implement multicasting using the network's broadcast capabilities instead. Hosts must then rely on software to filter out unwanted messages, incurring higher overhead [DC90].) As an alternative to using a broadcast with software intervention, separately-addressed messages can be sent from the source to each group member to effect a multicast on a broadcast medium.

As networks have gotten larger and distributed services more various, multicasting has become increasingly regarded as the preferred paradigm of interprocess communication. Transmission of the same message to more than one site is undoubtedly useful for many distributed applications, for instance, load balancing, routing table updates, database updates. Replicated file updates, conferencing and dissemination of intermediate results of distributed computations are noted in [DC90] as motivation for multicasting. In addition, researchers have asserted that broadcasting is not a generally useful facility as there are few reasons to communicate with *all* sites [CD85]. Gray [Gray88] notes that the scalability of multicasting to very large networks appeals to practitioners, as opposed to broadcasting which does not scale well. Even the scalability of multicasting to large networks (in particular, internetworks) has been questioned, but research into efficient algorithms is providing support against that opinion [DC90]. The idea that broadcasting incurs more message transmission than is needed for many tasks is easily understood from the perspective of distributed database applications. Replication of data is expensive, so there are likely to be just a few copies of various fragments scattered in the network. Even when data is not replicated, a particular transaction will likely require coordination among a small subset of the sites. Motivated by the current research in this area, and given that multicasting is the most general form of logical communication, we concentrate in this thesis on multicast communication in distributed systems.

We have been discussing site-to-site communication, but typically messages are sent from process to process. More generally, we define a *multicast group* to be a subset of processes in a distributed system that are the destinations of the same sequence of messages. A *multicast* then is a message sent from a process to a multicast group. In [DC90] multicast groups (composed of hosts, not processes) are classified as being either

(a) pervasive	long-lived and containing many sites; or
(b) sparse	long-lived or transient but small; or
(c) local	transient and existing on subdomains of the internetwork.

Pervasive groups may best be handled by broadcast communication techniques. The authors of [DC90] contend that most groups are likely to be sparse. We will not distinguish here between networks and internetworks, and local groups are likely sparse compared to the size of the networks, so we, too, will be concerned with sparse groups. (Consider, for instance, the cost of replicated data to understand why groups are sparse.) Much of the research on multicasting focuses on a particular kind of group. [McKL90] assume sustained groups. We, too, expect the multicast groups to be long-lived. (Again, consider the distributed database application.)

Much of the research we have mentioned so far has been concerned with the routing of multicast messages. It is not our intention here to pursue this area any further, Instead, we focus on two important guarantees that are often required of process-to-process communication and consider providing these guarantees for multicasts. One of these properties is reliability - a guarantee that the destination processes do indeed receive the messages sent to them. The other is ordering - a guarantee that destination processes receive their messages in consistent orders.

Consider first the need for ordering guarantees with the following example of a banking system with two main computers. Each computer has a copy of the entire banking database and will process all transactions arriving from the branch offices. (The second machine is needed for disaster recovery.) Transactions should be executed in the same order at the main computers, else the database state will differ. For instance, consider a deposit and a withdrawal to the same account. If the withdrawal is done first, an overdraft occurs and a penalty is charged. With the deposit first, no penalty is incurred and the resulting account balance is different. See [GA87] for additional details on this type of application.

In the same banking example, consider now a second subset of sites to distribute new software releases or system tables (e.g., defining overdraft penalty charges). This second group includes the two main computers, but in addition other development machines. Even though two different subsets of sites are involved in transactions, it is still important to process all messages in the same order at the machines in the intersection of the groups.

This example is just one illustration of why ordering properties are important for multicasts. Birman and Joseph [BJ87] have studied multicasting and cite updating replicated data and deadlock avoidance in lock management as applications that can take advantage of ordered multicasts. The ISIS System that they have implemented at Cornell relies heavily on ordered communications. Their experience with a working system leads them to claim that "[our] communication primitives actually simplify the design of distributed software and reduce the probability that subtle synchronization or concurrency related bugs will arise." They further state, "We believe that this is a very promising and practical approach to building large fault-tolerant distributed systems, and the only one that leads to confidence in the correctness of the resulting software." Ordered multicasts are also useful in various concurrency control mechanisms (e.g., see [KG87]).

The need for reliability of message delivery for many applications is clear. Consider again the distributed database described above. If one site receives update messages that the other site does not, the data copies will diverge. Message loss may occur for many reasons, including buffer overflow.

In this thesis we consider the problems of guaranteeing consistent ordering and reliability of multicast message deliveries. We study several ordering properties in detail, concentrating on a property called multiple group ordering. We survey existing methods for providing the ordering properties (Chapter 2) and describe a new solution that provides all the ordering properties under consideration (Chapter 3). We compare the new solution to extant work through analysis and experimentation (Chapter 4). We consider the problem of sites entering and leaving multicast groups, in other words, how to handle dynamic multicast groups (Chapter 5). We then address the problem of providing reliable multicast delivery, in particular when ordering properties are also required. There are many different reliability alternatives, so we begin with a formal model of ordered and reliable multicasting (Chapter 6) to get a handle on the possibilities. We then discuss the reliability of our and other ordered multicast solutions both informally and using the model (Chapter 7). We then consider some future directions for our research (Chapter 8).

Chapter Two

Ordered Multicasts

Before we describe the ordering properties for multicast messages, we present a simple model for message delivery which we will use for the next four chapters. Our distributed system consists of a collection of processes running at various sites. Recall that a *multicast group* is a subset of processes in the system that are the destinations of the same sequence of messages. For simplicity, we will assume there is only one process at a site that is a possible destination of multicast messages. (Extending any of the solutions to accommodate multiple processes per site is straightforward.) Subsets of *destination processes*, then, form multicast groups, which may intersect. Multicast messages may originate at one or more *source* sites and may be destined to any of the multicast groups. To get to a destination process, messages must first arrive at the host site (the site where the process is running) and are then *delivered* to the destination process. (Delivery is not always immediate. Many protocols require some coordination among the sites before delivery. Or, more simply, a site may have to wait for an earlier message before delivering the one it has just received.) Before a message arrives at the host site it may be routed through various other sites.

Three useful ordering properties that multicast protocols can provide are the following ones, arranged from weakest to strongest.

(a) *Single source ordering.* If messages m_1 and m_2 originate at the same source site, and if they are addressed to the same multicast group, then they are delivered in the same order to all destination processes.

(b) *Multiple source ordering.* If messages m_1 and m_2 are addressed to the same multicast group, then they are delivered to the destination processes in the same order, even if m_1 and m_2 originate at different sources.

(c) *Multiple group ordering.* If messages m_1 and m_2 are delivered to two processes, p_1 and p_2, they are delivered in the same order, even if m_1 and m_2 originate at different sources and p_1 and p_2 are in different multicast groups.

Figure 2.1 illustrates these three properties.

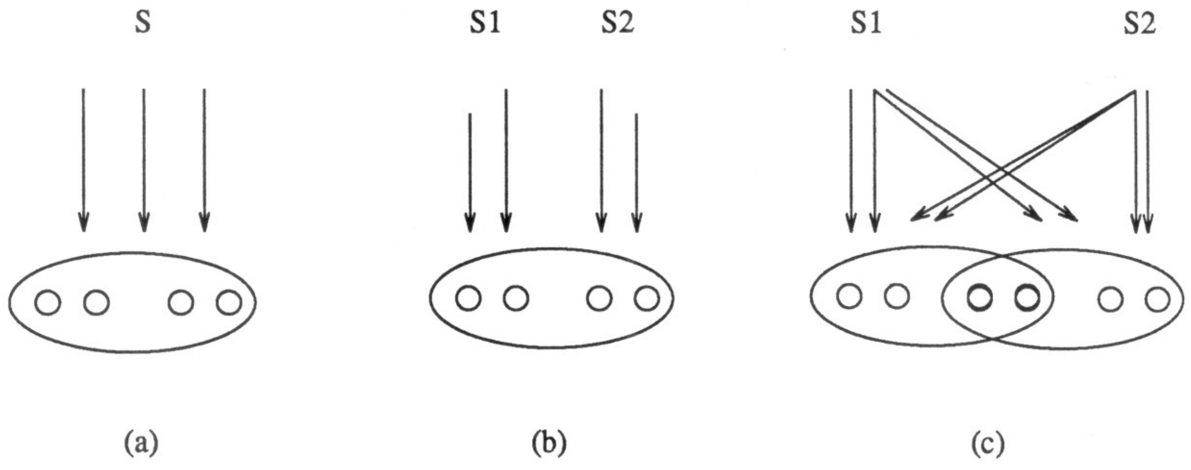


Figure 2.1

2.1 Sequence Numbers

Guaranteeing the single source ordering property (a) is relatively simple and is sometimes done by the underlying communication network. The basic idea is to number the messages at the source and to have destination sites hand the messages to the destination processes in that order. Note this also allows the destination to determine if it is missing any messages. We refer to this simple method by the name of *sequence numbering*.

2.2 Timestamps

Enforcing the multiple source and group properties (b,c) is harder than guaranteeing single source ordering. One solution is to use *timestamps* [Lamp78, Schn82]. Each source assigns a unique timestamp to its messages. A site can deliver a message when it can ensure that there are no outstanding messages with smaller timestamps. For a site a in multicast groups α and β, if multiple source ordering is required, site a must check with all potential sources of an α message before it can deliver any α message. For multiple group ordering, site a must check with potential sources of both α and β messages before it can deliver either an α or β message. If the potential sources are unknown, a must check with all sites in the system.

2.3 Token Passing

A centralized solution presented by Chang and Maxemchuk [CM84] is designed for broadcast networks and solves the multiple source ordering problem. To guarantee consistent ordering, all messages are funneled (logically) through a primary receiver called the token site which determines a total order for all messages. When a source transmits a message, m, on the broadcast channel, sites that receive it (do not miss it due to buffer overflow, for instance) queue the message for later delivery to the destination process.

The token site, assuming it received m, acknowledges m to the source and includes a sequence number with the acknowledgment. It is the sequence number which uniquely identifies the message and gives it a position in the total order.

The token site responsibility is rotated among the sites. A site only accepts the responsibility of being token site if it has received all messages so far. Once L sites have accepted the token since m was sent, m can be delivered at the destinations. This implies that the system is "L-resilient," in other words, as long as there are L or fewer site failures no message will be lost. Once the token has gone all the way around to the original acknowledger the message has been seen everywhere and can be discarded.

[CM84] does not discuss multiple multicast groups, but the same approach could be used to guarantee the multiple group ordering property, as long as all overlapping groups use the same central controller.

2.4 Central Sequencer

A completely centralized version of the token-passing solution is presented in [KTHB89] in the form of a broadcast algorithm. The algorithm is designed for broadcast networks and is not designed for fault tolerance. That is, the algorithm handles message loss, but is not concerned with site failures. Whenever a site wishes to broadcast a message, it sends it first to the sequencer - the site whose job it is to totally order broadcasts. The sequencer assigns the next sequence number to the message and broadcasts it. Sites can determine they are missing messages using the sequence numbering. As with the token-passing solution, the focus of the work is on broadcasting and multicasting is not mentioned. The algorithm could also provide multiple group ordering for multicasts as long as the same central sequencer is used for all groups, and multicast is effected with just one network message. The authors briefly address the issue of reliability, first by stating, "If the sequencer fails, the whole system will come to a grinding halt." Recovery from sequencer failures is then sketchily considered: the history of message delivery could be replicated and a new node could be elected as the new sequencer. Since this work does not consider fault-tolerance to be a pressing issue, we will focus on the token-passing solution here, even though the spirit of both algorithms is similar.

A similar idea is presented in [CZ85] where messages are filtered through a *publisher*, which numbers them and issues them to the *subscribers*. The subscribers, then, are the multicast group; the publisher is the central sequencer. Missed messages, again, are detected by a gap in the numbers. The issue of fault tolerance is not addressed in this work either.

2.5 Two-phase Solution

Birman and Joseph present an interesting solution (attributed to Dale Skeen) to the multiple group ordering problem as part of their work on the ISIS distributed system at Cornell [BJ87]. Their algorithm resembles two-phase commit [Gray78]. Each site in the multicast group maintains an ordered queue for the destination process. The source sends the messages to the multicast destinations. Each destination gives the message its own priority number, a systemwide unique number higher than any given so far by that site. The message is marked "undeliverable" and put on the end of the queue at each site. Each receiver returns the priority number to the source. The source picks the highest priority number it sees and sends it back to the receivers, who replace their original number with the new one and tag the message as "deliverable." Each receiver reorders its queue in sequence number order. Whenever a message at the front of the queue is "deliverable," it is delivered.

This algorithm is targeted to sites connected on a local area network where communication is fast and partitioning is rare. Contrary to [KTHB89], the two-phase solution is concerned with fault-tolerance. The algorithm can be used on any type of network; however, poor performance prohibits this.

2.6 The Total Protocol

A recent solution to the problem of multiple group ordering is presented in [MMA90] and is called the Total protocol. It is designed for broadcast networks and the authors envisage their algorithms implemented at the interface level. The Total protocol makes use of the Trans protocol which uses a series of positive and negative acknowledgments † to guarantee that all sites receive messages. The technique is best illustrated by an example taken from [MMA90]. In Figure 2.2, each group of letters designates a message, labeled by the upper-case letter. The messages are ordered left to right in the order they are broadcast on the network. (Sources are not shown.) Upper case letters denote the message, lower case letters indicate piggy-backed acknowledgments and overhead bars indicate piggy-backed negative acknowledgments. In the example, the first message that is broadcast is message A. There are no piggy-backed positive or negative acknowledgments in this message. The second message is message B. The lower-case "a" juxtaposed with "B" indicates that message A is acknowledged by message B. Recall that the underlying network is a broadcast network, so all messages and any acknowledgments are heard by all sites (barring failures). Once a site knows a message has been acknowledged it does not acknowledge it, though it is possible that multiple sites will acknowledge the same message. In the example, C acknowledges message B and D acknowledges C. In the example, the acknowledgment of C alerts a processor that it missed C. This processor sends a new message, E, and includes a positive acknowledgment for D and a negative acknowledgment for C. C and its accompanying

† A positive acknowledgment for a message m is an acknowledgment that indicates the destination received m. A negative acknowledgment indicates that message m was *not* received and should be retransmitted.

acknowledgments are retransmitted. The processor that acknowledges C with message D implicitly acknowledges B and A.

$$A \quad Ba \quad Cb \quad Dc \quad E\bar{c}d \quad Cb \quad Fce$$

Figure 2.2

The Total protocol uses the partial ordering of messages provided by the acknowledgments combined with an implicit voting scheme to define a total order for message delivery. See [MMA90] for the details.

2.7 Other Solutions

Table 2.1 summarizes the solutions in terms of the ordering properties they provide and the networks for which they are intended. There are also other solutions that we do not review here. In particular, [Wuu85] uses logs of message receipts at each site. In [NCN88] a solution to the multiple source ordering problem is presented. Each multicast group has a group manager (and backup managers) responsible for delivering the messages to the group members. [LG88,LG90] presents a decentralized solution that relies on majority consensus among designated processes at each site to commit on the ordering of broadcasts. The paper [GKL88] focuses on the single source problem and how to make failure recovery particularly efficient. A system described in [Ahuj89] provides *Flush* primitives for non-FIFO channels. These primitives guarantee single-source ordering of messages. They are not designed for a continuous stream of ordered multicasts (they are too inefficient), but rather for intermittent ordered messages, for instance to generate global snapshots.

Algorithm	single source ordering	multiple source ordering	multiple group ordering	broadcast	point to point
Sequence numbers	*			*	*
Timestamps	*	*	*	*	*
Token-passing	*	*	†	*	
Sequencer	*	*	†	*	‡
Publishing	*	*	†	*	‡
Two-phase	*	*	*	*	‡
Total	*	*	*	*	

*	provided as is
†	provided if modified
‡	provided, but not intended for point-to-point network

Table 2.1

2.8 Summary

Though the solutions presented in this chapter are valid, each has its share of performance problems. When applied to multiple group ordering, the timestamping solution is prohibitively expensive in most cases. (The only time this solution makes sense is if sites broadcast regularly.) The token-passing solution places too much of a burden on the central site and incurs long delay if multiple failures are to be tolerated. Also, this centralized solution is limited to broadcast networks and severely limits which sites can serve as the central controller when applied to multiple group ordering. The central sequencer solution is not designed for fault tolerance. The Total protocol is also limited to broadcast networks. The two-phase solution incurs substantial message overhead. To address these concerns, we have developed a new solution - the propagation graph algorithm - which we present in the next chapter.

Chapter Three

The Propagation Graph Algorithm

3.1 Motivation and goals

To alleviate some of the problems associated with the solutions of Chapter 2, we have developed a solution to the multiple group ordering problem (and the multiple source ordering problem) that compromises between a highly centralized approach and a solution which requires many messages. In addition, we have attempted to make our solution more flexible than others by allowing simple variations on the technique to accommodate the needs of the system. In particular, the method displays a clear load/delay tradeoff (where load is processing time at the sites and delay is how long it takes for a multicast to be delivered to all destination processes). The method is easily modified to favor one over the other or balance the two. Finally, we attempt to gain efficiency by not forcing sites to process messages for other sites when these messages are not of interest locally. Our solution is geared toward sparse, long-lived multicast groups (see Chapter 1). For the purposes of this chapter we assume that the system is nearly perfect, that is, sites do not fail, messages are not lost and the network does not partition. We will approach the issue of fault tolerance when we discuss reliability in Chapters 6 and 7.

3.2 Our solution

Our solution, the *propagation graph algorithm*, attempts to achieve the goals mentioned above. It is inspired by the token-passing solution in [CM84], but contrary to their method, messages are not ordered at a single central site. Rather, they are ordered by a collection of nodes structured into a *message propagation graph*. In particular, the graph is a forest. Each node in the forest represents a computer site and the graph indicates the paths messages should follow to get to all intended destinations. Instead of sending the messages to the destinations and then ordering them (which happens with timestamping, the two-phase solution and the token-passing solution) the messages get propagated via a series of sites that order them along the way by merging messages destined for different groups. When a site receives a message it knows immediately where in the order the message belongs and need only wait for earlier messages before delivering it to the destination process. The key idea is to use sites that are in the intersections of multicast groups as the intermediary nodes.

Figure 3.1(a) illustrates a scenario where sites x and y are sending messages to multicast group $\alpha = \{a,b,c,d\}$, while site z is sending messages to $\beta = \{c,d,e,f\}$. A simple propagation graph is indicated in Figure 3.1(b). In order to guarantee that all messages are delivered in the same relative order, sources x, y, and z send their messages only to site c, who merges them. Site c forwards the group α messages from x and y to a and b, the β messages from z to e and f, and the merged α, β messages to d. Thus, all sites deliver their messages in the order defined by site c.

Our propagation graph algorithm has two components: the propagation graph (PG) generator and the message passing (MP) protocol. The PG generator builds the

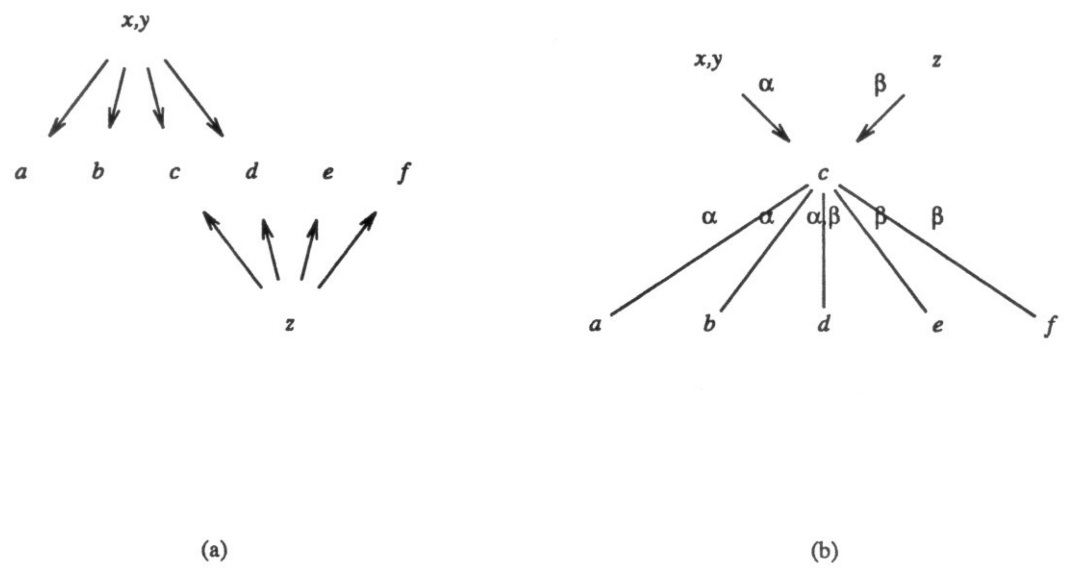


Figure 3.1

propagation graph for a given set of multicast groups. For simplicity, we assume one site (designated the manager) runs the PG generator and transmits the resulting graph to the other sites. (A two-phase commit can be used to ensure all sites receive the graph.) We discuss dynamic changes to the propagation graph in Chapter 5. Once a site knows what the graph looks like, it uses the MP protocol to send, receive, propagate and forward messages.

We establish some terminology for message passing. The group that is to receive a message from a *source* is called the *destination group*. The source sends the multicast message to one designated site in the multicast group, called the *primary destination*. (The primary destination could be the source.) Any time a site sends a message directly to another site, we refer to these sites as the sender and receiver, respectively.

One important requirement for the algorithm is that single source ordering be satisfied; if the underlying network does not provide this, we use sequence numbering. Note the implied use of single source ordering in the example of Figure 3.1. Site *c* delivers its α and β messages locally in the same order in which it sends them to site *d*. Site *d* is able to determine the order in which they were merged by the sequence numbers. In fact, every edge in the graph relies on the messages being ordered at the receiver the same way in which they were sent by the sender.

In Figure 3.2 we show a propagation graph for a more complicated example. Here we have nine sites: *a, b, c, d, e, f, g, h* and *j* and eight destination groups:

$\alpha_1 = \{c,d\}$, $\alpha_2 = \{a,b,c\}$, $\alpha_3 = \{b,c,d,e\}$, $\alpha_4 = \{d,e,f\}$, $\alpha_5 = \{e,f\}$,

$\alpha_6 = \{b,g\}$, $\alpha_7 = \{c,h\}$ and $\alpha_8 = \{d,j\}$.

Site *d* is the primary destination for α_1, α_3, α_4 and α_8, *c* is the primary destination for α_2 and α_7, *e* is the primary destination for α_5 and *b* is the primary destination for α_6. Note that messages do not necessarily flow down to the bottom of the tree. For instance, *g* only receives α_6 messages.

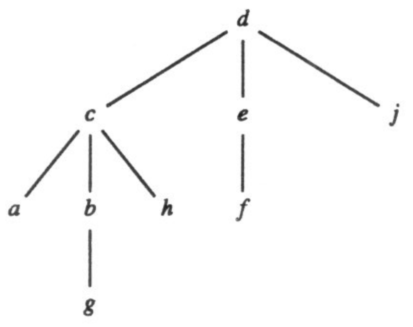


Figure 3.2

Detailed pseudo-code for the PG generator is given in Appendix I and a proof of correctness is provided in Section 3.6. In the rest of this chapter we present our approach in a less formal fashion.

3.3 The PG Generator

To ensure correctness our technique must guarantee the following two properties:

(1) Multiple group ordering, i.e., all messages are delivered in the same relative order, and

(2) If x is in group α, then x gets all messages destined to group α.

To satisfy these requirements, it is sufficient for the propagation graph to have the following two properties:

(PG1) For every group α there is a unique primary destination p; and

(PG2) For every site $x \in \alpha$, there is a unique path from p to x.

There are also two optional properties that the graph can exhibit and which our PG generator attempts to provide:

(PG3) The primary destination of group α is a member of α; and

(PG4) Let p be the primary destination of α and x be another site in α. Then, the nodes in the path from p to x are all members of α.

When there exists a node a on the path from p to x where a is not a member of α, we call a an *extra node*. A propagation graph that has no extra nodes possesses the desirable *independence property*.

Both PG3 and PG4 are desirable because they yield more efficient graphs: there is no need for nodes that are not involved in a multicast group to be handling messages for that group. Our PG generator does guarantee property (PG3), but unfortunately generates extra nodes sometimes. For example, if we add group $\alpha_9 = \{d, a\}$ to the example of Figure 3, we obtain the same tree. However, node c is an extra node for α_9. We discuss the impact of extra nodes in Section 3.5 and in Chapter 4.

To start building the forest, the PG generator selects the site in the largest number of groups (d in our example) and makes it a root. This greedy heuristic helps keep the trees in the forest short. (We therefore do not consider the cost of processing the messages at a primary destination to be substantial, but rather attempt to minimize the length of the path down the tree to cut communication cost.) For purposes of explanation, we call the groups to which the root belongs *root groups* and the other sites in the root groups *intersecters*. The root, then, is the primary destination for all root groups.

To determine the children of the root, procedure *new_subtree* is called, with the root d as parameter. This procedure works as follows. It partitions the non-root groups so that no group in a partition intersects a group in another partition. In our example there are two partitions, $P_1 = \{a,b,c\},\{b,g\}, \{c,h\}$ and $P_2 = \{e,f\}$. This step also has the effect of partitioning the sites (a,b,c,g and e,f). In an attempt to achieve property (PG4), among the partitions, the generator only considers those that contain an intersecter. From each of these, one of the intersecters is chosen to be a child of the root using the same heuristic used for picking the root: choose the site that is in the most groups in the partition. In our example, for P_1, b and c occur in the most groups, so we arbitrarily pick c over b. In P_2, we arbitrarily pick e over f. (In practice, there may be a good reason not to choose arbitrarily, e.g., one site tends to be less loaded than the others.) Finally, there may be sites that are intersecters but do not occur in any partition. In our example, this is true of j. These sites become children of the root. At this point the tree looks as shown in Figure 3.3.

To generate the next level of the tree, a recursive call is made to *new_subtree* for each child, with the child as parameter. Since by determining the root and adding the children, we have found primary destinations for the groups $\{c,d\}$, $\{a,b,c\}$, $\{b,c,d,e\}$,

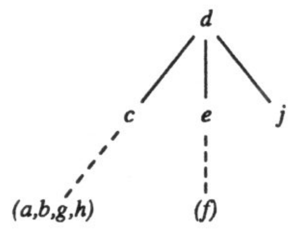


Figure 3.3

$\{d,e,f\}$, $\{e,f\}$, and $\{d,j\}$, we no longer consider these groups for partitioning in these recursions. Also, we have placed *d*, *c*, *e* and *j* in the graph, so these are no longer candidates as children.

In our example, the recursion on *c* leads to a single partition consisting of $\{b,g\}$. Sites *a*, *b* and *h* are attached as children of *c*. The recursion on *e* leads to no partitions and to attaching *f* as a child of *e*. The recursion on *j* leads to no new nodes in the tree. At the next level, *new_subtree* is called four times with *a*, *b*, *h* and *f* as parameters. The call with *b* as parameter leads to *g* being added. The others result in no new nodes. Finally, the recursive call to *new_subtree* for *g* leads to no new nodes and the process terminates having determined the propagation graph. In this case, just one tree is obtained. If, however, there were still sites that had not been placed (hence, groups whose primary destinations had not been determined) after the original call to *new_subtree* returned, another root would be picked from the groups left and *new_subtree* called again to determine the next tree. The loop continues until no more sites are left.

3.4 The MP Protocol

The propagation graph specifies the flow of the messages in the network. The primary destination for each multicast group is the member closest to a root. A site that receives a message propagates it down any subtree that contains members of the destination group for the message. In our example, *d* is the primary destination for $\{c,d\}$, $\{b,c,d,e\}$, $\{d,e,f\}$ and $\{d,j\}$, *c* is the primary destination for $\{a,b,c\}$ and $\{c,h\}$, and so on. When, for instance, *d* receives a message for $\{b,c,d,e\}$, it sends copies to *c* and *e*. Site *c*, in turn, sends a copy to site *b*.

To be more precise about processing messages, we describe the message passing protocol for the case of point-to-point networks. The MP protocol requires every site to maintain sequence numbers for each site to which it sends messages, as determined by the propagation graph. This guarantees that a receiver can order the messages from a sender correctly in case they arrive out of order. (It also allows for the detection of lost messages. Acknowledgments are not required in the MP protocol. Null messages and timeouts are used for failure detection, a topic we return to in Chapter 7.)

At each site, two queues are maintained, a queue for messages destined to a local process and a wait queue for messages that arrive out of sequence. When a site receives a message, it checks the sequence number against the sequence number it expects from that sender. If they do not match, the message is queued on the appropriate wait queue until the earlier one is received. If they do match, the receiver determines if any of its descendants in the tree are destinations for this message. If so, it sends it to the children that are the subroots of those subtrees, using the appropriate sequence number for each child it sends it to. If the receiver is a member of the destination group, the message is queued for local delivery. In addition, the receiver checks if there are any messages in the wait queue from that sender that were waiting on this message. If so, it processes these message(s) in the same manner.

3.5. Optimizations

In Section 3.3, we mentioned the *independence* property and claimed that the PG algorithm attempts to preserve independence. Independence, stated again, is achieved when no site is required to process messages for a group to which it does not belong. In other words, no site is an extra node for some group. Unfortunately, the PG algorithm does not always build a forest that achieves independence. Consider the following example. There are eight sites: *a,b,c,d,e,f* and *g* and destination groups $\alpha_1 = \{a,b,d,e\}$, $\alpha_2 = \{b,c,f\}$, $\alpha_3 = \{d,e,f,g\}$. One possible tree generated by the PG algorithm is indicated in Figure 3.4(a). In this case, *d* must propagate α_2 messages to *f*, even though *d* is not in α_2.

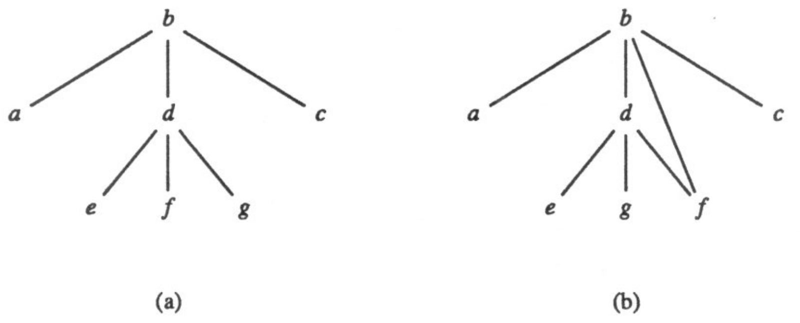


Figure 3.4

In systems where site autonomy is important, independence is an appealing property. For this reason, we consider extending the PG algorithm to provide more independence to the sites. We will see, however, that it is not always possible to achieve complete independence for every site with our methods.

In the example of Figure 3.4(a), an edge can be added from *b* to *f* and the α_2 messages can bypass *d*. (See Figure 3.4(b).) This, of course, destroys the tree structure, so care must be taken to do this correctly. The important question is: What allows the extra edge? The answer is that there is no other site establishing a relative order among α_2 and α_3 messages. In other words, *f* is now the only node in the tree merging α_2 and α_3 messages. This implies the general rule that all merges that are now required at the new destination site must be uniquely performed at that site.

There are other scenarios that require a more complicated algorithm for achieving independence. For instance, if we have sites *a,b,c,d,e,f,g* and *h* and groups $\alpha_1 = \{a,b,d,e\}$, $\alpha_2 = \{b,c,f,g\}$ and $\alpha_3 = \{d,e,f,g,h\}$, then the PG algorithm produces the tree in Figure 3.5(a). In this case, *d* is not independent since it is forwarding α_2 messages. Figure 3.5(b) indicates how the graph can be changed to avoid sending α_2 messages through *d*. This change requires more than just adding an edge; the subtree rooted at *g* must become a child of *f*. This is necessary because both *f* and *g* need the merged α_2 and α_3 messages.

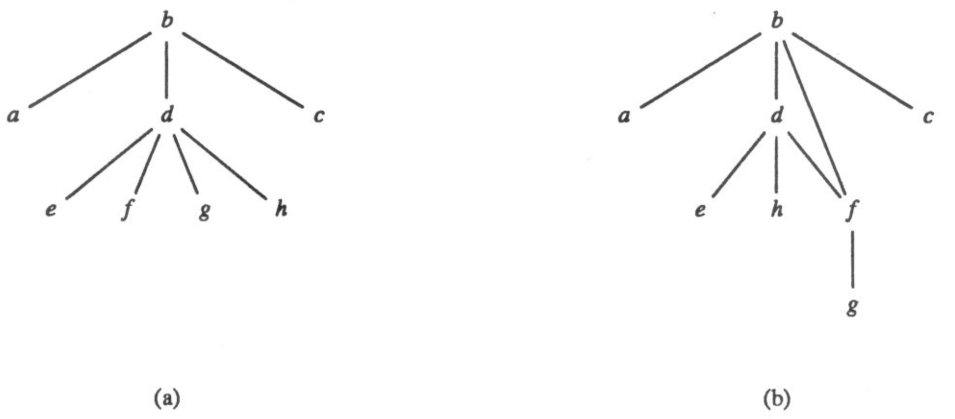


Figure 3.5

The following example illustrates a situation where independence cannot be preserved. Here we have four sites: *a,b,c* and *d* and four destination groups: $\alpha_1 = \{a,b,c\}$, $\alpha_2 = \{a,b,d\}$, $\alpha_3 = \{a,c,d\}$ and $\alpha_4 = \{b,c,d\}$. The propagation graph produced for this example is indicated in Figure 3.6(a). In this case, *b* is not independent because it is forwarding α_3 messages. It is tempting to use the same method as in the previous example, producing the graph of Figure 3.6(b). Now *c* is receiving α_3 messages from *a* that it merges with α_1 messages from *b*, but there is a problem since these messages are already being merged by *a*. (Note, in addition, that now *c* is no longer independent; it is forwarding α_2 messages.) It is possible to send both α_1 and α_3 messages from *a* to *c* already merged, but then *c* must merge α_1 with α_4 messages, which is done by *b*. So, another idea is to send α_1 messages from both *a* and *b* to *c* and let *c* resolve the α_1, α_3 ordering with the α_1, α_4 ordering (Figure 3.6(c)). But this technique gets complicated when we try to make *c* independent. We can add an edge from *b* to *d* to send the merged α_2, α_4 messages, *c* can send *d* the merged α_3, α_4 messages, but now *d* must merge α_2 messages with α_3 messages, which is done by *a*. So, we can add the edge from *a* to *d* and send the α_2, α_3 ordering to *d*. (See Figure 3.6(d).) Now *d* receives α_2, α_3 messages already merged, α_2, α_4 messages already merged and α_3, α_4 messages already merged. It must resolve these to achieve a total ordering on α_2, α_3 and α_4 messages and deliver them locally. Note, however, that this may be impossible. Consider message m_{α_2} that *a* orders before m_{α_3}. Site *b* orders m_{α_4} before m_{α_2}. Site *c* orders m_{α_3} before m_{α_4}. Site *d* cannot resolve m_{α_2}, m_{α_3} and m_{α_4} to a total order. In fact, we cannot achieve independence in this case with our methods.

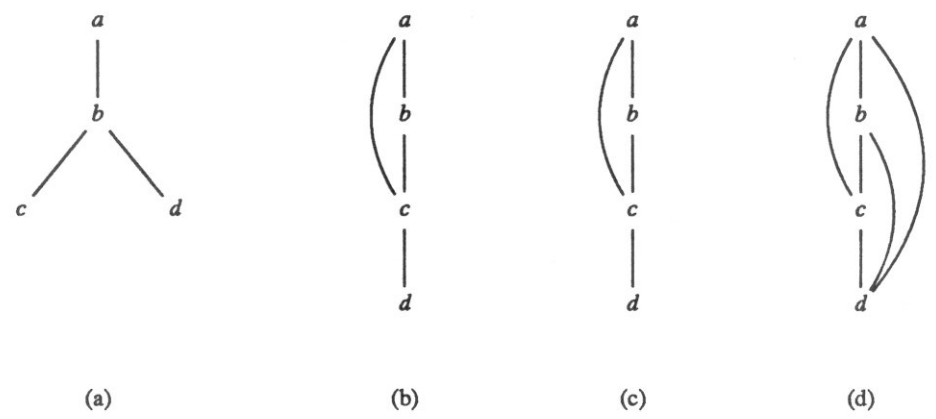


Figure 3.6

Because of the potential difficulties with resolving the ordering of messages, we do not resort to sending the same messages via several paths, even though sometimes this technique achieves independence. The prudence of such a solution is questionable anyway, since extra communication may be required for the redundant messages. It is possible that some of this can be reduced by sending a message along only one path and sending only a message identifier on the other paths, but we do not investigate these possibilities further.

3.6 Correctness of the PG Generator

Finally, we show that the forest we build using the methods of Section 3.3 is correct. It is not difficult to see that the PG generator indeed builds a forest that includes every site. To show that the forest guarantees the multiple group ordering property we must prove two things: (1) all sites receiving the same messages deliver them in the same order; (2) all sites receive the messages destined to them.

To see that property 1 is satisfied, say we have two sites, a and b, that receive messages m_1 and m_2. Say that a delivers these in the order m_1m_2 and b delivers them in the order m_2m_1. If m_1 and m_2 are messages for the same multicast group α, then initially they are ordered by the primary destination for α. It is easy to see that the sequence numbering scheme used in the MP protocol guarantees that this order is maintained as m_1 and m_2 are propagated.

Suppose, then, that m_1 is destined for group α_1 and m_2 is destined for α_2. Call the primary destinations for these types $pd(\alpha_1)$ and $pd(\alpha_2)$, respectively. If $pd(\alpha_1)$ and $pd(\alpha_2)$ are the same site, then the situation is the same as when m_1 and m_2 both are destined for α_1. Say then that $pd(\alpha_1)$ and $pd(\alpha_2)$ are two different sites. Certainly a, b, $pd(\alpha_1)$ and $pd(\alpha_2)$ are all in one tree of the forest, and $pd(\alpha_1)$ and $pd(\alpha_2)$ are both ancestors of a, b. By the properties of trees, we know that there is only one path from the root of the tree to any node. Thus, there is only one path from the root to a, only one path from α_1 to a and only one path from α_2 to a. This implies that either $pd(\alpha_1)$ is ancestor to $pd(\alpha_2)$ or viceversa. Say $pd(\alpha_1)$ is ancestor to $pd(\alpha_2)$. Then, at $pd(\alpha_2)$ m_1 and m_2 are merged and propagated to a. By the same reasoning, m_1 and m_2 are merged at $pd(\alpha_2)$ and propagated to b. Certainly $pd(\alpha_2)$ determines the order just once by the MP protocol and the messages are propagated to both a and b. Since this ordering is easily seen to be preserved by the MP protocol, a and b cannot deliver these messages in inconsistent orders.

It is also not difficult to see that the algorithm guarantees that all sites receive their message types. Say some site does not get some message type that it should. The situation should look as in Figure 3.7, where a and b are supposed to receive type α messages, but b is not on a path for α messages. Figure 3.7 indicates that the PG generator places nodes a and b in different subtrees of x, even though they are in the same group (α). This is an impossibility since the partitioning step of new_subtree puts all sites that share groups in one subtree of the current subroot.

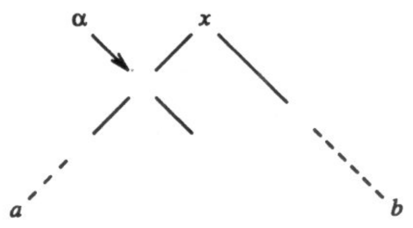


Figure 3.7

Chapter Four

Performance

Performance is the crucial measure of the practicality of the propagation algorithm, so here we compare the propagation method to the two-phase algorithm of [BJ87] and to a strictly centralized version of [CM84] †. Two models are considered, a point-to-point network and a broadcast network. We look specifically at three performance measures: N, the number of messages required to send a multicast under the multiple group ordering property; D, the time elapsed between the beginning of the ordered multicast and the time when all the members of the multicast destination group can mark the message ready for local delivery; and the load incurred at the sites to process the multicast messages.

† Since [CM84] relies on a broadcast network we do not consider changing the central site. Instead, we look at a centralized solution which is essentially a propagation graph that consists of one tree of depth 1 and is similar to the central sequencer algorithm of [KTHB89].

4.1. Point-to-point model

The point-to-point network model we consider here is typical of networks like the ARPANET. For a single source to send the same message to n sites it must send n messages, one to each receiver. For site a sending a message to site b, we say it takes a processing time P to put the message on the network and it takes latency time L for the message to get to site b (network delivery time). Thus, a simple multicast with no ordering requirements from source s to n sites requires n messages and the time elapsed before the last site receives the message is $nP+L$.

4.1.1 *N* and *D*

Table 4.1 indicates the performance of the three multicast methods for the first two measures, N and D. Consider first N for an ordered multicast from source s to n sites. The two-phase algorithm requires n messages to initially get the message from the source to the destinations. Another n messages are required to return the locally-assigned priority number from the destinations to the source. Finally, the source sends out the final priority number of the message, for a total of $3n$ messages. The centralized solution requires one message from the source to the central site and $n-1$ messages from the central site to the remaining nodes, for a total of n messages. The propagation algorithm requires 1 message from the source to the primary destination. In the best case, only group members form the path down the tree, so $n-1$ more messages are required to get the message to every destination. If there are extra nodes on the path, then the number of messages totals $n+\varepsilon$, where ε is the expected number of extra nodes.

To compute the delay for the two-phase method we consider the three rounds of messages. The time it takes between when the source sends its first message and the last site receives the message is $L+nP$. Then, the delay for that last site to send the local ordering information back to the source is $L+P$. The source then sends the final priority order to the sites, again $L+nP$. The total is $3L+(2n+1)P$. The centralized algorithm has delay $L+P$ from the source to the central site plus $L+(n-1)P$ delay from the central site to the last recipient, for a total of $2L+nP$.

The delay in the propagation graph case depends on the length of the longest path from the primary destination to a member of the multicast destination group. We introduce the variable d to represent the expected depth of this recipient from the primary destination.

The total delay in this case, then, is the sum of the delays from the source to the primary destination and the delay from the primary destination to the group member that is furthest away (at depth d). The delay from the source to the primary destination is simply $L+P$. It is simple to show that the delay from the primary destination to the group member at depth d is maximized at $dL+(n-1+\varepsilon)P$ (and in general is much less). Total delay then for the propagation algorithm is at most $(d+1)L+(n+\varepsilon)P$.

	two-phase	*centralized*	*propagation*
N	$3n$	n	$n+\varepsilon$
D	$3L+(2n+1)P$	$2L+nP$	$(d+1)L+(n+\varepsilon)P$

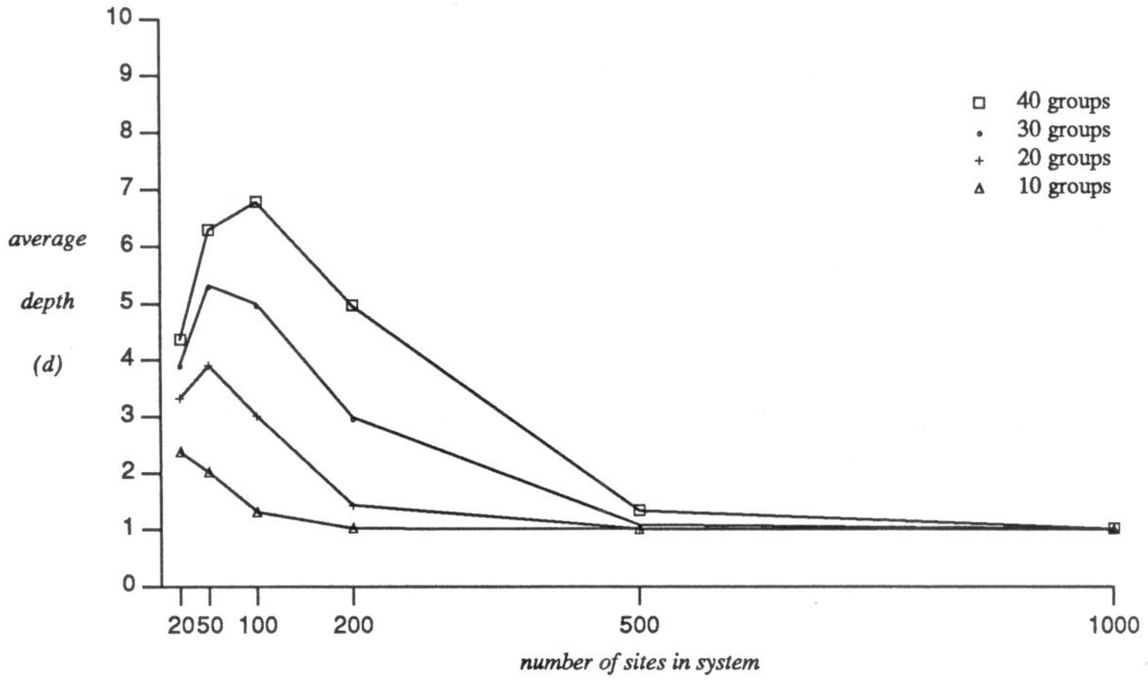
Table 4.1

Clearly, the performance of the propagation algorithm depends on the values of ε and d. It turns out that in most cases of interest, ε and d are relatively small. We established this via experiments on randomly generated multicast groups. For a fixed number of sites, number of groups, and group size we chose sites from a uniform distribution to generate a random set of multicast groups and then computed their propagation graph. We have looked at a broad range of network sizes (from 20 to 1000 sites), group sizes (from 5 to 40) and number of groups (from 10 to 40). A representative sample of those results is given here.

Graphs 4.1 and 4.2 indicate results for the average depth, with Graph 4.1 considering a static group size for a varying number of sites participating in the system and Graph 4.2 considering varying group sizes for a static number of system sites. These curves represent an average over all the runs of the average value of *d* for the groups in a run. For these and all graphs, each data point is asserted with 95% confidence. For Graphs 4.1 and 4.2, the confidence interval is within ±10% of the mean. The behavior of the curves is explained by the ratio of group size to the number of sites. When this ratio is large (left end of the horizontal axis), there are many intersections among the groups; the abundance of common nodes leads to short bushy trees. As the ratio decreases, there are fewer intersections, the trees get longer, so *d* increases. However, when the ratio is very small (right end of the horizontal axis), there are so few intersections that the algorithm produces a forest of many sparse, short trees.

Another factor in the measurement of D for the propagation graph algorithm is ε, the expected number of extra nodes (see Table 4.1), for which we show results in Graph 4.3. (The confidence interval is ±0.5.) For each propagation graph generated in the experiment, the number of extra nodes required to deliver a message to each site in a group was averaged over all the groups. The graph indicates the average over all these runs. The number of extra nodes is low in general and the curves display the same behavior as those for the depth.

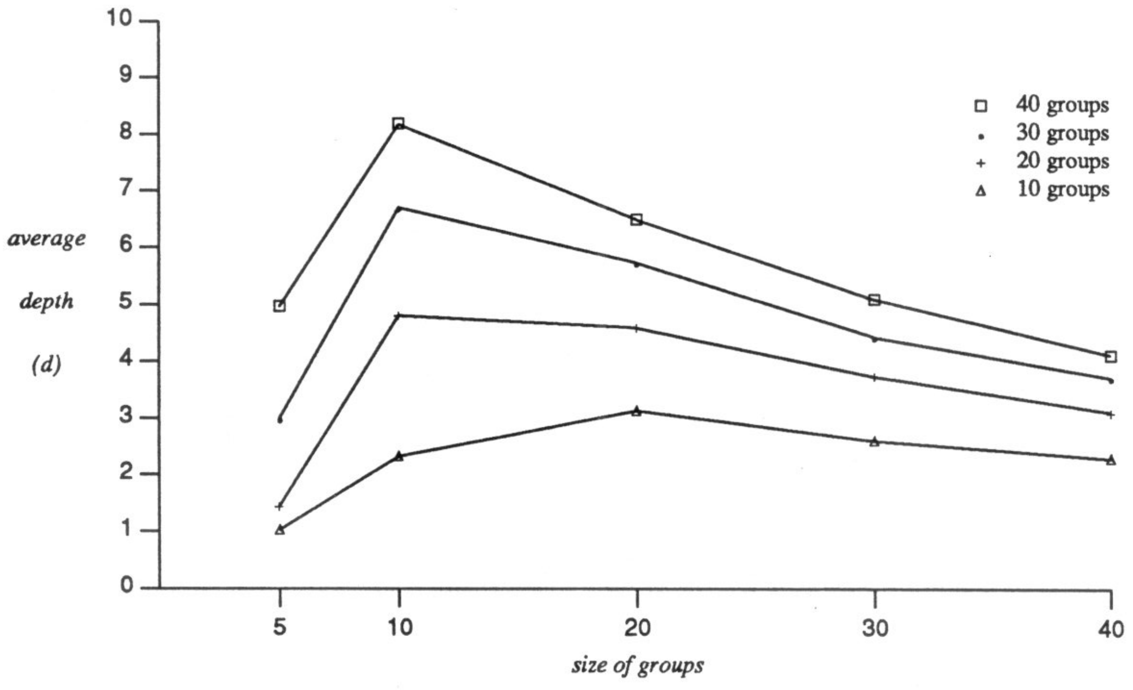
With a range of values available for ε and *d*, we can return to Table 4.1 and compare the algorithms in detail. Clearly, for *N*, the number of messages, the propagation



Group size 5

Graph 4.1

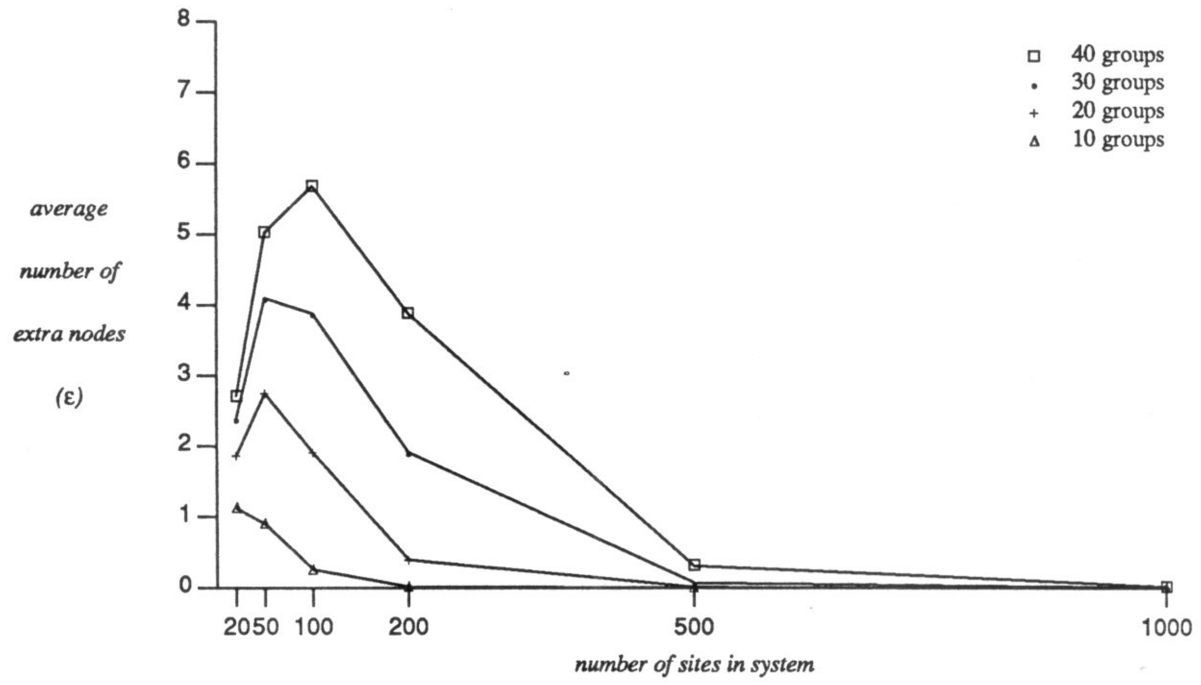
method is significantly more efficient than the two-phase approach, since ε is shown to be small in our experiments. Also, the propagation technique is only slightly worse than the centralized solution for this measure. For *D*, the delay, if the depth of the tree, *d*, is less than or equal to 2, then the propagation method is better than the two-phase solution. If the depth is greater than 2, the propagation method may incur longer delay for message delivery than the two-phase approach. Note, though, that this is only the case if the latency time of a message dominates. If the processing costs are significant, then the propagation algorithm may still achieve better performance. Further, we expect *d* to be less than or close to 2 when the group size is small. Small group sizes occur in many interesting applications; for instance, since maintaining copies of replicated data is expensive, copies are kept at only a few sites.



200 sites in system

Graph 4.2

Also, it is important to remember that the depth computed in the experiments is an average over the worst case delay for a group. Thus, for any one group, the delay may be long for one or just a few nodes, whereas the other nodes may get the message with small delay. In the two-phase algorithm, it is only when the commit messages are sent out that there is any variance in the delay among the nodes; each site must wait at least two full rounds. Finally, note that the centralized solution is better in terms of delay and number of messages. However, for large group sizes, he central site of the centralized solution is a bottleneck, so D and N are not sufficient measures of performance. For this reason we next consider the load at the sites.



Group size 5

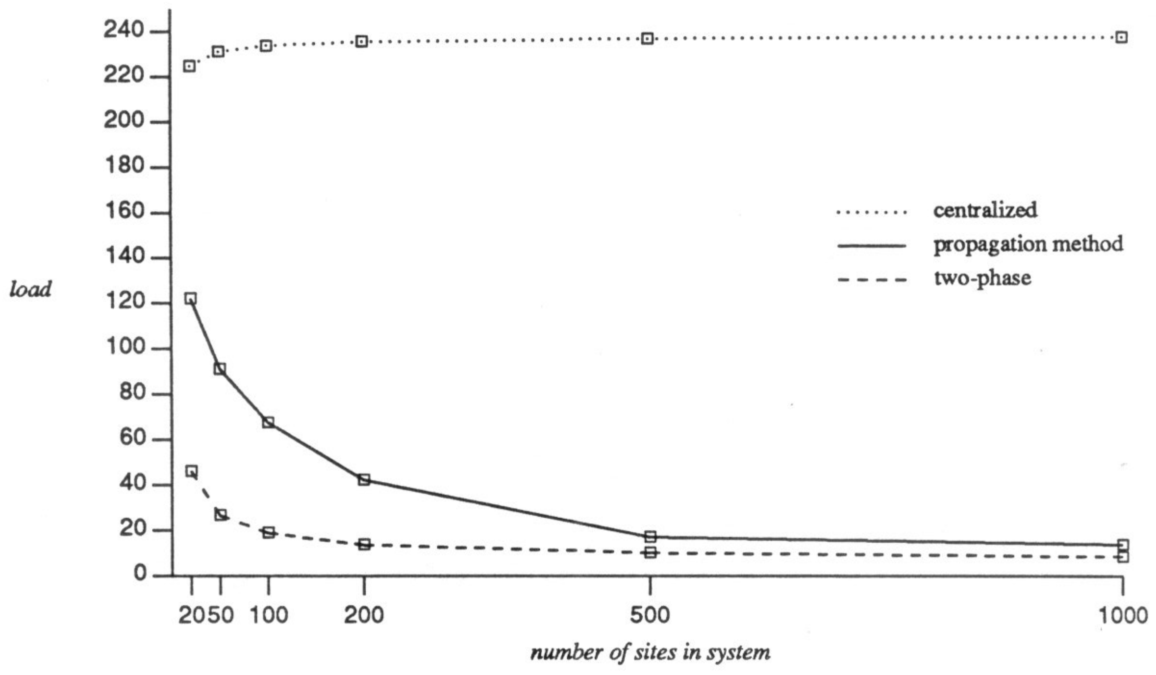
Graph 4.3

4.1.2 Load

To measure the load of the three methods we generated multicast groups as in the experiments for ε and *d*. Then we considered how many messages a site must process if a message were sent to each group. Both send and receive messages are included in the load. In the experiments presented here, for each set of groups we determine the maximum over the loads at the sites and average this over all th[illegible]ns.

In Graphs 4.4 and 4.5 we show the load results, using a confidence interval of ±10% of the mean. For the propagation graph method, load is high when just a few sites are primary destinations and we have short, bushy trees. As trees get longer, load decreases until trees are sparse and load is very low. The load for the two-phase method is very well distributed, yielding low average numbers. This is the load at receivers, though.

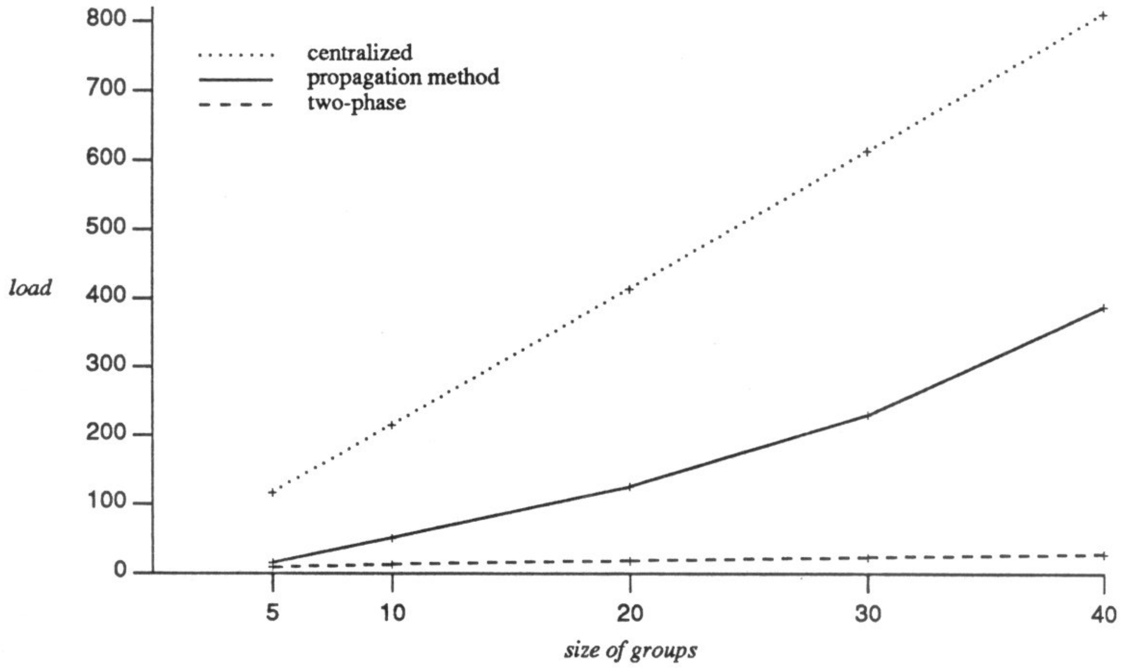
Load at the source will be higher (possibly much higher) if the groups are large since the source must communicate directly with every site in the group. As expected, load in the centralized case is high, reflecting the central site bottleneck.



Group size 5

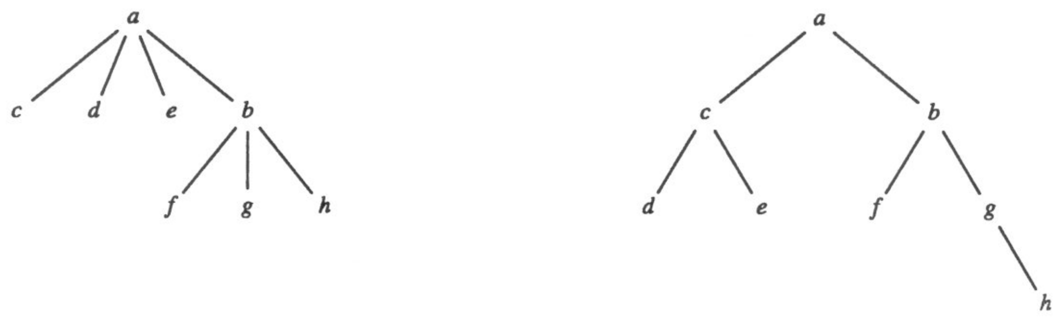
Graph 4.4

The experiments show that the propagation graph method strikes a good compromise between minimizing delay and distributing load. In addition, it provides a flexible solution that can be tailored to a particular network or application. For instance, the greedy heuristic for picking primary destinations is just one option for generating propagation trees; other techniques with different goals can be used. Also, the graph can often be modified to find the correct balance between delay and load. For example, groups $\alpha = \{a,b,c,d,e\}$, $\beta = \{b,f,g,h\}$, $\gamma = \{a,b\}$ yield the tree in Figure 4.1(a). To reduce the load at a and b, we can transform our graph into the tree in Figure 4.1(b), which is correct and does not use extra nodes. The depth, however, has increased. Further, the propagation



200 sites in system

Graph 4.5



(a)

(b)

Figure 4.1

graph requires fewer messages than the two-phase algorithm, making it a desirable alternative in cases where network traffic is already high.

Finally, we comment that we have performed many other experiments using the exponential distribution for determining the size of the groups, instead of the fixed value for which we presented results here. The graphs produced by these runs have the same shape but generally give lower numbers for the average depth. Thus, the propagation strategy yields good results when a propagation graph is formed for groups of various sizes.

4.2. Broadcast Model

The second model we consider is a broadcast network of the Ethernet variety. With this model, any message sent by a source is transmitted to all sites on the network at once. Each site must check if it is a destination of the message. In the case of a unicast or broadcast, this is done in hardware at each site's network interface. Multicast addressing is available, but requires the source to broadcast and each site on the network to check in software if it is a member of the destination group. † Note that no message transmission is reliable (for example, buffers may overflow); thus, the network does not provide any ordering properties. Note, also, that we do not achieve any concurrency in sending messages; there is just one message on the network at a time.

The performance results for this case are shown in Table 4.2 for the three algorithms we are considering here. N for each case is determined as follows. The two-phase algorithm requires one message from the source to the n destinations, n messages from the destinations to the source and one message back from the source to the destination group, for a total of $n+2$. The centralized solution requires simply two messages: one

† It is true that the Ethernet does provide hardware check for multicast addresses, but we choose to use a more general model.

from the source to the central site and one from the central site to the rest of the members of the destination group. The propagation graph algorithm requires a message from the source to the primary destination and a multicast at each level of the subtree. If the expected depth of the subtree is *d*, the resulting number of messages is $d+1$.

	two-phase	*centralized*	*propagation*
N	$n+2$	2	$d+1$
D	$(n+2)T$	$2T$	$(d+1)T$

Table 4.2

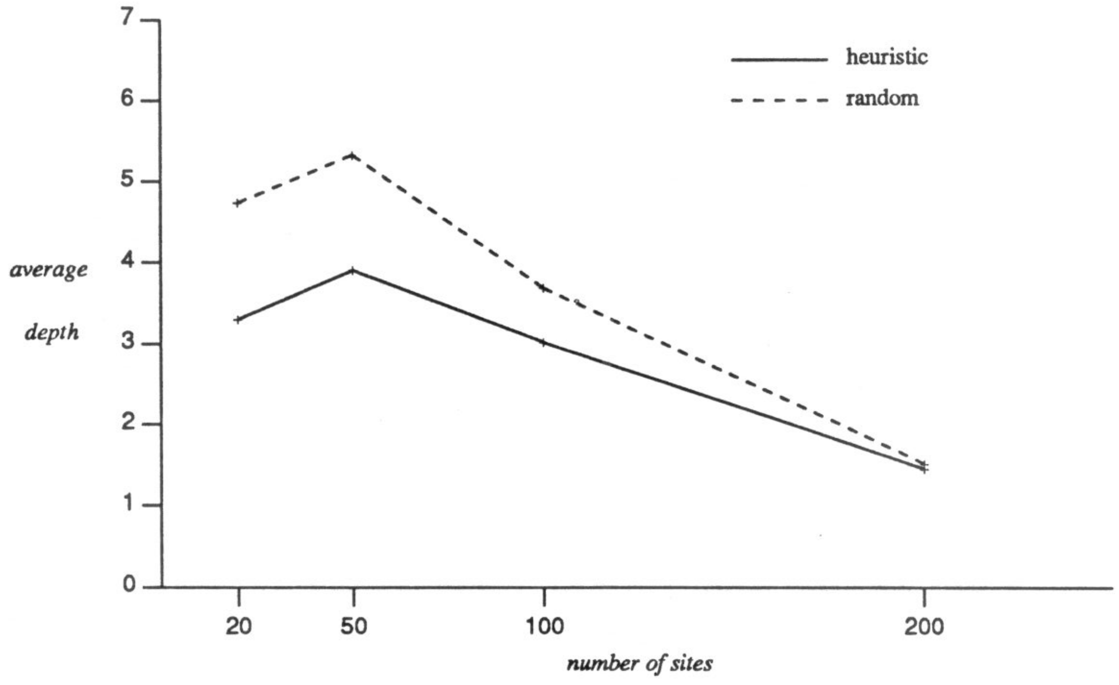
Due to the nature of the broadcast network, each message has the same delay. This delay consists of the time for the sender to put the message on the network, plus the time for the message to get to the other sites plus the time for each site to determine if it is a member of the destination group. Let T be this total delay. The delay in each of the three algorithms is simply the number of messages multiplied by T, as indicated in Table 4.2.

The table seems to indicate that the centralized solution is superior in the broadcast case. Also relevant, however, is the bottleneck at the central site, which will increase the value of T and the delay. Given the results for values of d as presented in the last section, the propagation graph method has small delay and requires few messages. The two-phase algorithm is only comparable for small values of n.

With the broadcast model, it is important to consider the fact that many of the messages are multicasts, requiring a software check (for membership) at the receiving computers. Any such check will affect the load at a site. These checks occur not just at destinations but at *all* sites on the network, whether or not they are involved with multicasts. Indeed, it may be preferable to ignore the network-provided multicast facility and revert to point-to-point mechanisms (i.e., a series of unicasts) to multicast a message. Such an option might be followed if the number of sites receiving the multicasts is small compared to the number of sites on the network and/or load at the sites is already high.

4.3 Random Graphs

The performance results presented in the previous sections consider only one heuristic for generating a propagation graph (choosing the site in the most groups as the next subroot as long as it is an intersecter). Note, however, that the sufficient conditions for correctness of the graph (Section 3.3) imply that any forest is a correct propagation graph. Sources need only send group messages to a common ancestor of all group members. (This may violate optional property PG3.) The obvious question, then, is: If any tree works, can we be sure that the heuristic described in Section 3.4 is yielding "good" trees, trees that meet our goals? To show that the heuristic we have developed is indeed yielding desirable trees, we compare it, in Graphs 4.6 and 4.7, to trees that have been generated randomly. Graphs 4.6 and 4.7 indicate results for average depth for group size 5 (20 and 40 groups). By a random graph, we mean a graph for which the root is chosen randomly, the remaining sites are partitioned as before and the children are chosen randomly from each partition. (Then the algorithm recurses on the children.) As shown in the graphs, the heuristic we have studied yields better trees than just picking sites randomly. Of course, there are other possible heuristics; we analyze some of these in the next section where we consider the topology of the network when building the propagation graph.



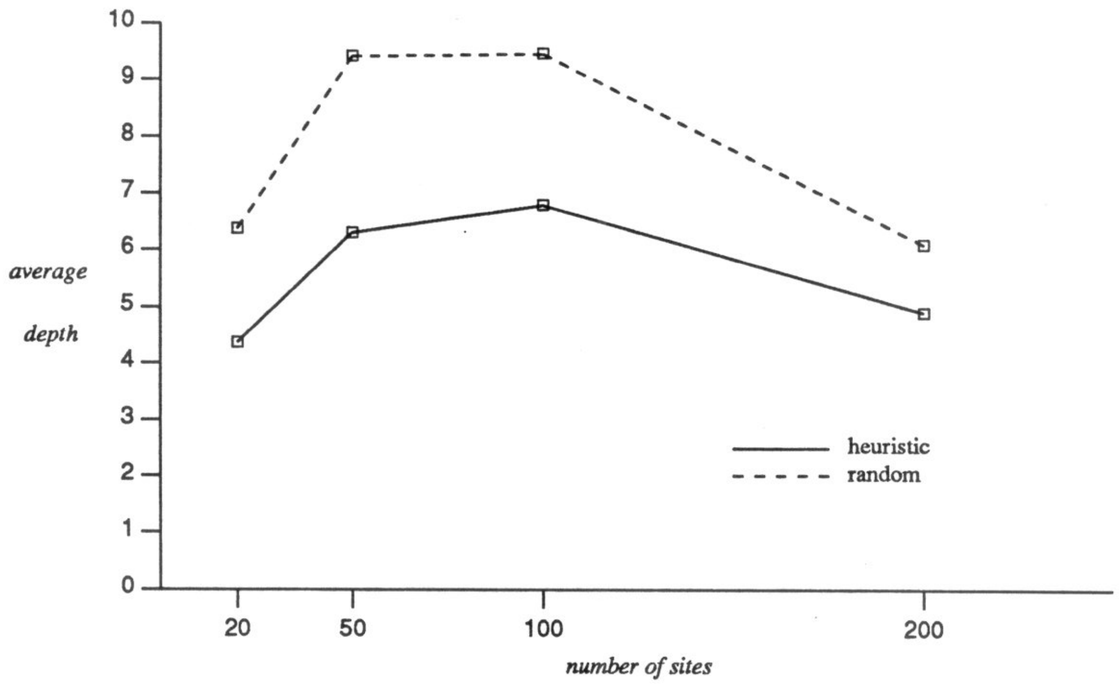
Group size 5

20 groups

Graph 4.6

4.4 Topology of Point-to-Point Network

An interesting question regarding the propagation graph algorithm (or any algorithm for message ordering) is how the topology of the network affects performance. In our previous analyses, we assume that the cost of sending a message from one site to another is the same for all pairs of sites. On a point-to-point network, however, there is likely to be a higher delay for messages sent between sites that are far apart than between sites that are neighbors. It is sensible, then, to consider designing propagation graphs to take advantage of the topology of the network. Here, we examine the cost of the original



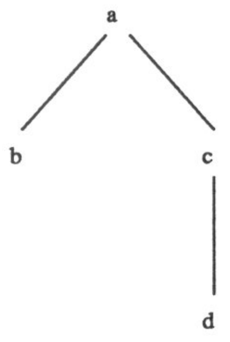
Group size 5

40 groups

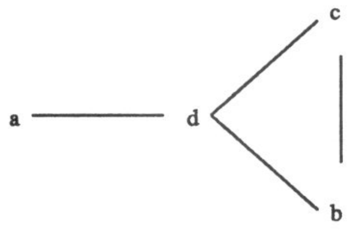
Graph 4.7

propagation graph algorithm when the topology is taken into consideration. We compare it to two other heuristics for generating propagation graphs. We also consider two tree structures that are commonly in use on networks and could serve as propagation graphs - minimum spanning trees and shortest path trees.

For these experiments we use a square grid for the topology, as this is representative of a network like the ARPANET. All links on the network incur a cost of 1. The simulations were done similarly to those in the previous section, except now the delay measure includes the cost of the physical path between two sites. For instance for the propagation graph of Figure 4.2(a) and the network of Figure 4.2(b), the cost of sending a message from *a* to *b* is 2. Since message delay is no longer the same for all messages sent, the



(a)

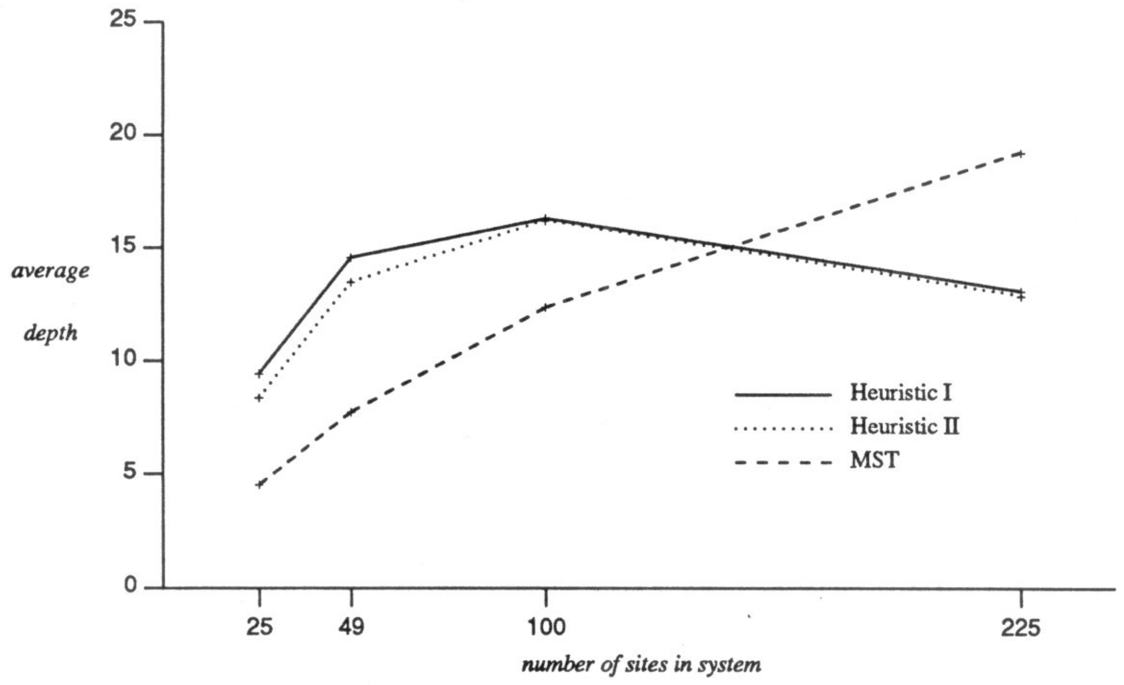


(b)

Figure 4.2

network delivery time described in Section 4.1.1 and used in Table 4.1 is no longer accurate. Instead, we compute a new average for network delivery time based on the cost of the physical paths between logical tree nodes. Thus, the portion of the expression for D in Table 4.1 that accounts for latency ($(d+1)$L) is replaced by the average cost of the physical path that messages must follow to get to the furthest destination. This average cost is determined via the experiments. Extra nodes are still counted only as they occur in the logical propagation graph. We do not consider the load here.

One simple possibility for a propagation graph that accounts for the topology of the network is a minimum spanning tree. For these experiments, a minimum spanning tree was generated using the most frequent group member as the root and giving that root as many children as possible while still yielding a minimum cost tree. Other choices of equal cost were determined randomly. The minimum spanning tree was compared to the original propagation graph method described in Section 3.4, which we now call Heuristic I. A new heuristic (Heuristic II) was analyzed as well. This heuristic is similar to Heuristic I except that when a choice is available among sites in the most groups, (i.e., there are several sites in the same number of groups), the site that is the shortest distance away is chosen. The chosen site is always an intersecter. Results for minimum spanning trees, Heuristic I and Heuristic II are indicated in Graphs 4.8 and 4.9 and show that a

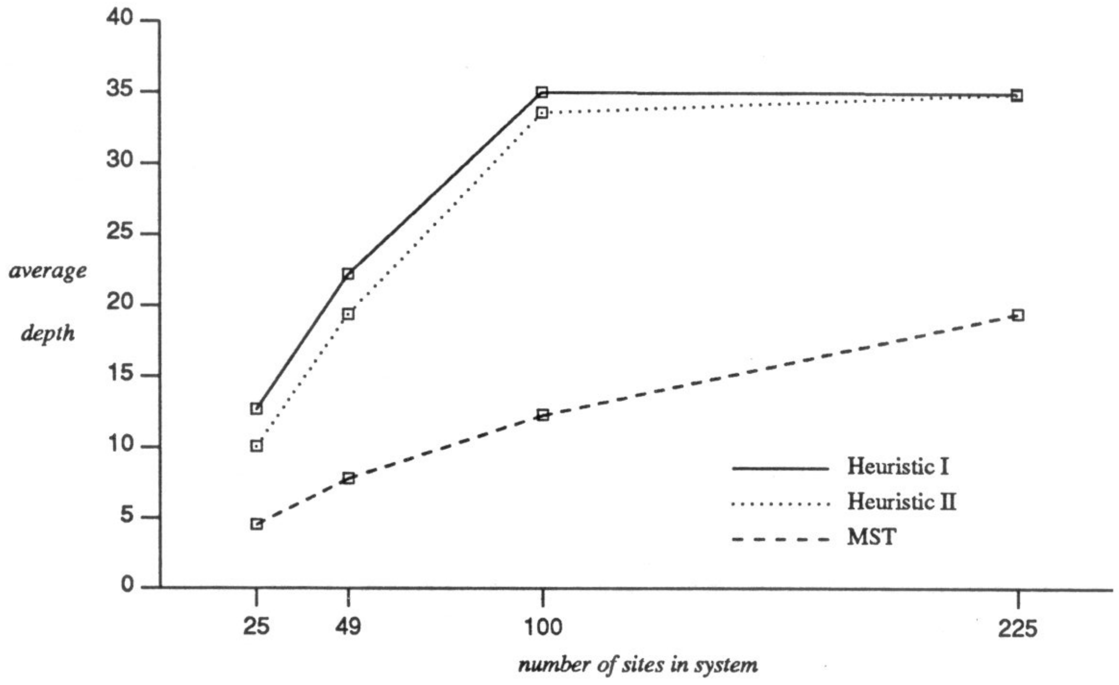


Group size 5

20 groups

Graph 4.8

smarter heuristic yields slightly better results than the original, but for smaller numbers of sites, the minimum spanning tree does very well. The depth always increases, though, for the minimum spanning tree algorithm as opposed to both the heuristics, which decrease in Graph 4.8. The heuristics are exhibiting the same behavior as Heuristic I showed in Section 4.'.1. As the number of sites grows, the large trees are breaking up into sets of smaller trees. A group that does not intersect any other yields one short tree that uses optimal physical paths between logical tree nodes. The minimum spanning tree technique, on the other hand, always yields a graph that consists of just one tree. Thus, group members may be spread around the tree making the logical path from primary destination to furthest member longer. This path likely passes through extra nodes. The



Group size 5

40 groups

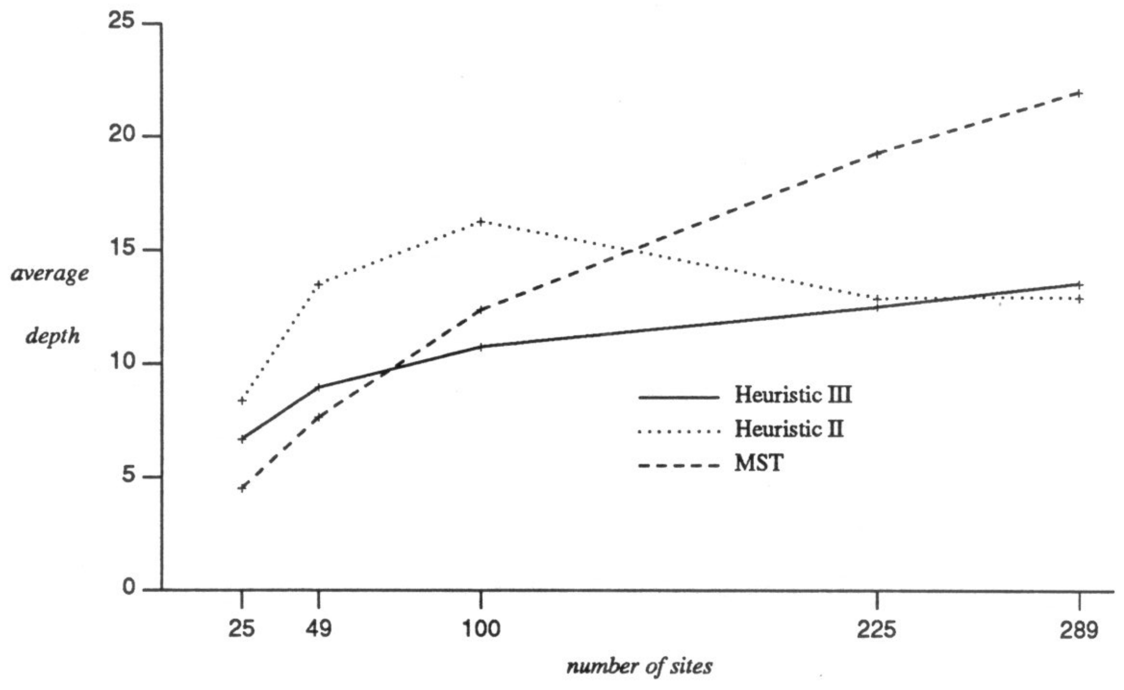
Graph 4.9

longer logical path is apt to make the physical path longer. This behavior is not as marked in Graph 4.9 as the larger number of groups makes group intersection likelier.

The good performance of minimum spanning tree in many cases encourages us to find a better heuristic than Heuristic II. Heuristic III makes choices for children that give priority first to neighbors. The closest site is always chosen as a child, but if there is a selection among sites that are tied for closest site then an intersecter in the most groups is picked. Graphs 4.10 and 4.11 indicate results for Heuristic III compared with the results from minimum spanning tree and Heuristic II. This third heuristic indeed outperforms Heuristic II, though cannot beat minimum spanning tree in some cases.

The poorer performance of minimum spanning tree for higher numbers of sites lead us to consider another common network structure, the shortest path tree. † This technique

† A shortest path tree of a graph, G, is a spanning tree rooted at a particular vertex, v, all of whose paths from v to another site are shortest in G [Tarj83].



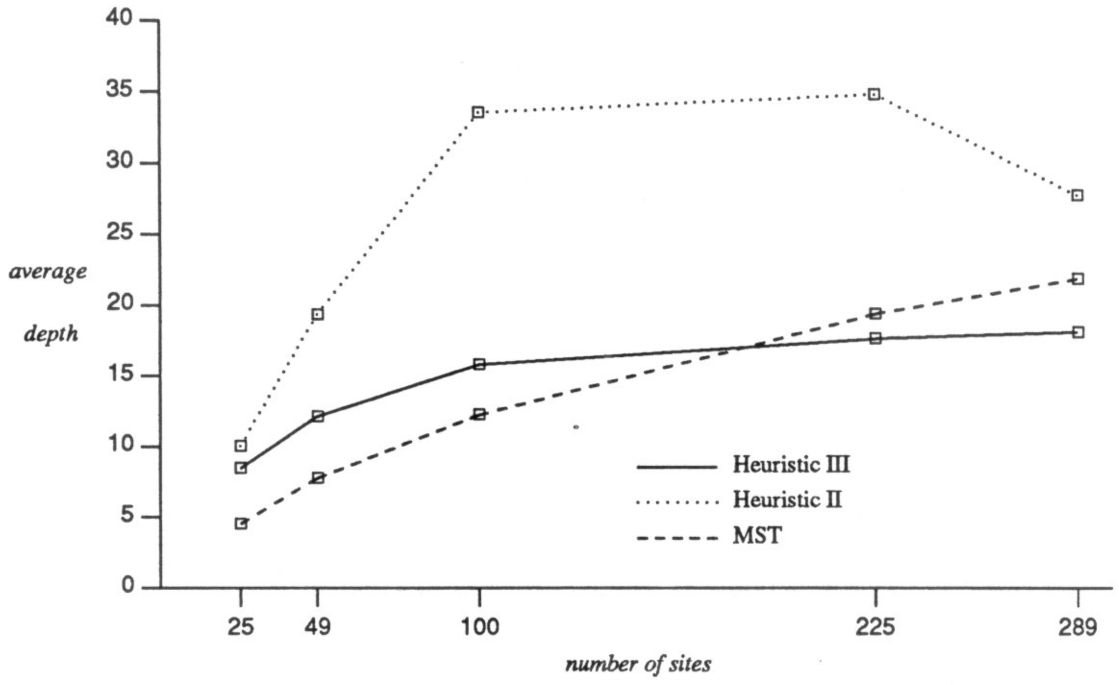
Group size 5

20 groups

Graph 4.10

makes sense because we can pick the source to be the site that occurs in the most groups, and end up with a short path to other group members from that site. Graphs 4.12 and 4.13 show results for the average depth of shortest path trees compared to minimum spanning trees and Heuristic III.

The results in Graphs 4.12 and 4.13 might lead one to conclude that it usually does not make sense to use any of the heuristics described here; instead, simply build a shortest path tree. A look at the average number of extra nodes for these techniques shows otherwise. Graph 4.14 demonstrates that the logical tree structures generated by minimum spanning trees and shortest path trees yield many more extra nodes than the heuristics. (The results for 40 groups are similar and are not included.) This can have more of an impact on delay than does the length of the physical path from primary destination to furthest group member. It is reasonable to expect that the physical routing of



Group size 5

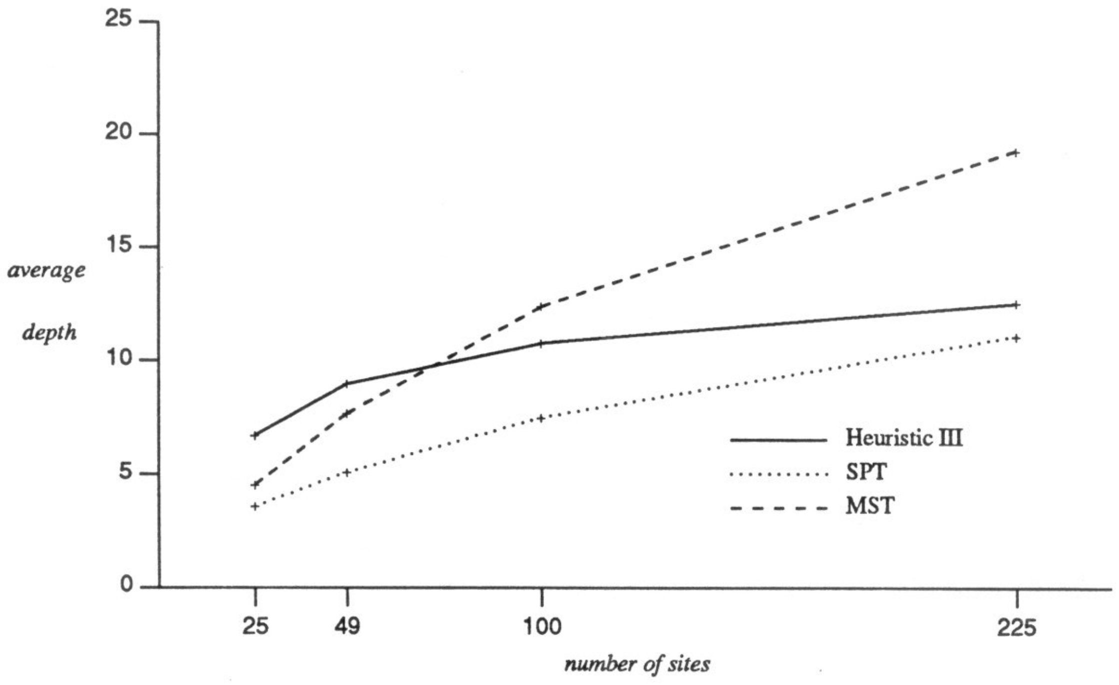
40 groups

Graph 4.11

messages will be handled in hardware at the interface level. Propagation of the message on the logical tree, however, will likely be done in software. The more sites in the logical tree that a message passes through, the longer it takes for the message to get to its destination *and* the cost at each such site is much higher than the cost of simply routing a message through a site. With this taken into consideration, the heuristics (especially Heuristic III) yield good performance.

4.5 Performance Conclusions

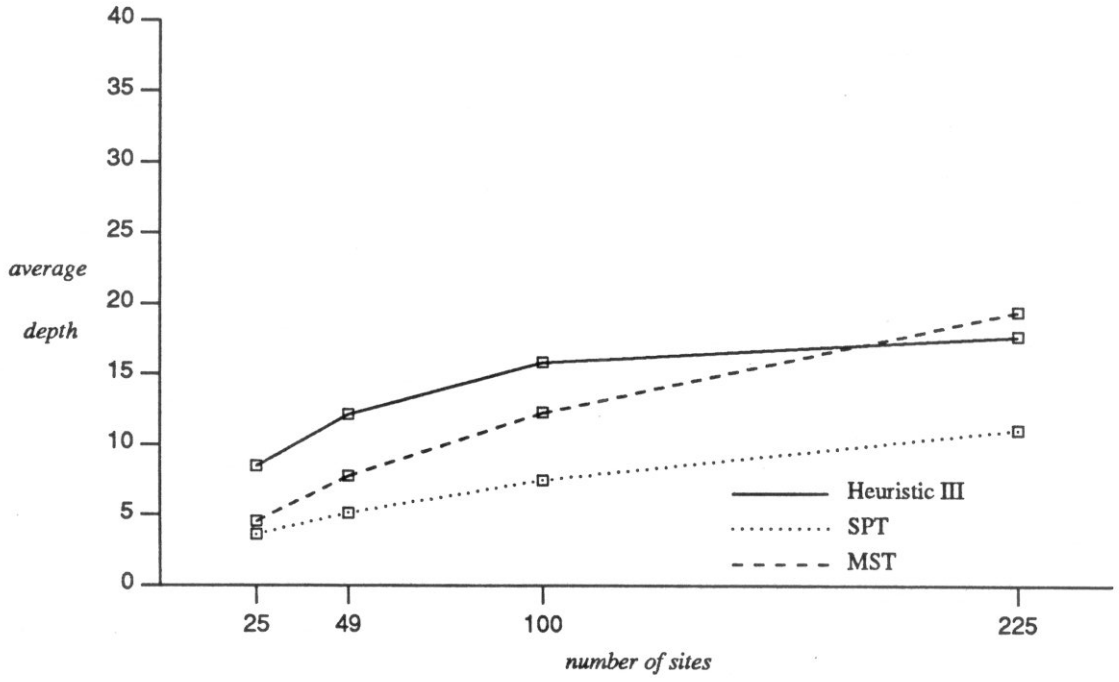
The experimental results presented in this chapter make it evident that the propagation graph algorithm provides a viable alternative for ordered multicasts in both point-to-point and broadcast networks. It generally requires fewer messages than the two-phase solution. It provides a compromise between minimizing delay and distributing load, avoiding a central site bottleneck. Further, it is a flexible solution in that there are many ways in which to build the propagation graph in order to meet the needs of a particular system. The heuristic we concentrated on here (Heuristic I) was certainly found to improve over a random graph. Two other heuristics that take into account the topology of the network also provide good performance. All heuristics obtained worse measurements for d, the expected depth, than did propagation graphs obtained via minimum spanning trees and shortest path trees. But the measures for ε, the number of extra nodes, were much better and show the heuristics to be better choices. Finally, it is important to note that there is a substantial cost associated with setting up the propagation graph that is not reflected in our results. Hence, this approach is of most interest when multicast groups are long-lived and receive many messages over the lifetime of the group. We return to the issue of setting up the propagation graph in the next few chapters, where we consider dynamic groups and reliability.



Group size 5

20 groups

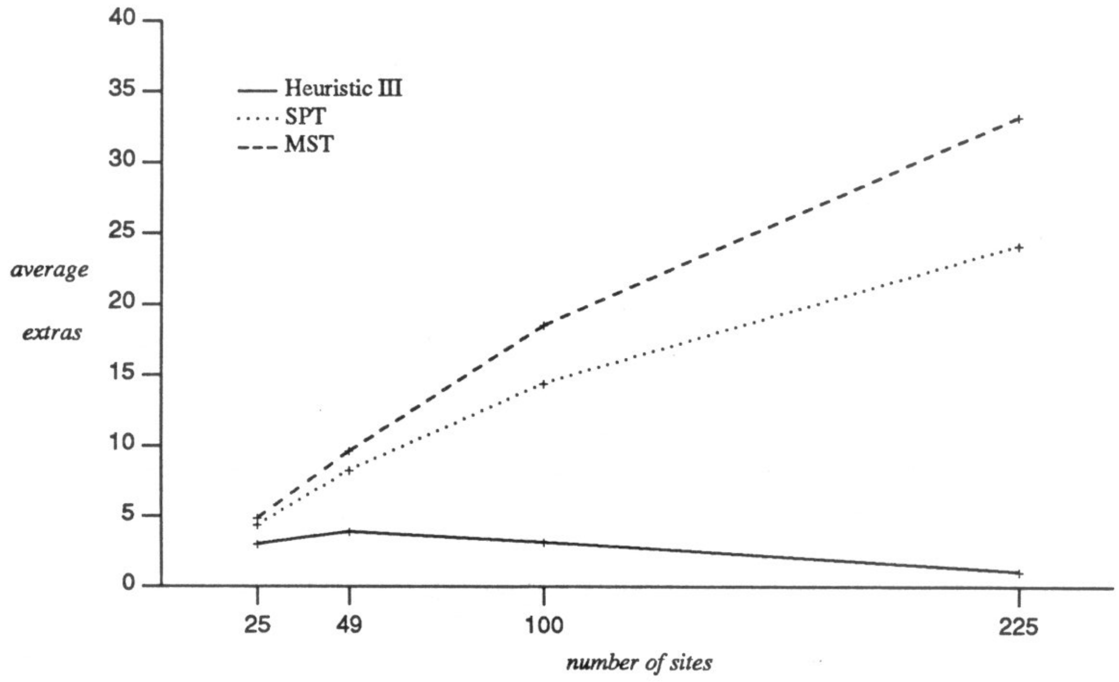
Graph 4.12



Group size 5

40 groups

Graph 4.13



Group size 5

20 groups

Graph 4.14

Chapter Five

Dynamic Groups

So far we have only considered how to provide the ordering properties when the multicast groups are static. It is likely, however, that the multicast groups will change over time. For instance, in the banking example of the Introduction, a new site with a new copy of the data may be added to the network. Or, a site may be permanently removed from the network. An entire group may be disbanded or a new group may be added. Site failures may also necessitate temporary changes to the multicast groups. For instance, in the example of Figure 3.2, if c fails, it may be desirable to temporarily delete c from α_1, α_2, α_3 and α_7 and rebuild the graph, then return to the original graph when c comes back up.

For these reasons, we briefly consider how the algorithms described in Chapter 2 accommodate dynamic multicast groups. We then describe in detail extending the propagation graph algorithm to allow for dynamic multicast groups. This initial description applies when the changes to the groups are not due to failures. We consider failures in Chapter 6.

5.1 Dynamic Groups in Other Solutions

One very nice feature of the two-phase algorithm is that it easily accommodates adding sites to groups and deleting sites from groups. Once a source knows what the amended group looks like, it just sends its multicasts to the new group and expects to hear from each member. No special protocol is needed. The token-passing solution, however, requires a complex "reformation phase" where a new list is committed (with three phases) and a new token site is elected. The propagation graph algorithm does require some special work, but is much easier than a three-phase algorithm.

5.2 Dynamic Multicast Groups with the Propagation Graph Algorithm

The main objective of our technique for providing dynamic multicast groups is to institute changes to the propagation graph efficiently, i.e., without requiring tight coordination among the sites and without preventing the delivery of messages for long periods of time. One solution is to have one site compute the new graph and use a commit protocol involving all the sites to terminate the old graph and install the new graph. This is a clean solution that prevents discrepancies over the state of the system; however, it involves high communication overhead and long delays. Instead, it is possible to take advantage of the ordering properties guaranteed by the propagation graph to make the changes to the graph consistently and rapidly.

The new algorithm is essentially the same as for the static case, except now we designate one site as the manager. When the multicast groups change, the manager is responsible for computing a new propagation graph and initiating the change system-wide. The manager is analogous to the coordinator of a commit protocol solution, but instead of committing the change, the propagation method will guarantee that it happens safely. Two operations are required: *Close* and *Open*. First the manager *Close*s the old tree by broadcasting a *Close* message. (Broadcast simply requires sending the message to each root and having it propagated down to every node in the trees.) Upon receiving the *Close*, a site stops processing later messages (i.e., does not deliver them locally or propagate them). Any new messages from sources are queued. Note that messages from sources are the only messages a site will receive on its tree after a *Close* since each parent *Close*s before its child and then does not propagate any more messages. Since the *Close* message is ordered along with all other messages, for a message *m*, either all destinations will order *m* before the *Close* or all destinations will order *m* after the *Close*. Thus, a message *m* is either delivered at all destinations before the *Close* or is not delivered anywhere until the next graph is *Open*ed.

The manager opens a new graph by broadcasting an *Open* message to each new root, along with the new graph information. This *Open* message is also ordered among the other messages. After receiving the *Open*, each site incorporates the new graph information locally and propagates the *Open* to its new children. Then it checks each message it queued after the *Close*. If the site is still the primary destination for the message, it processes it in the usual way (delivers it locally if appropriate and propagates it down the new tree to its new children). If the site is not the primary destination for the message, it throws it away and informs the source of the new primary destination. The source resends the message to the new primary destination. A source does not begin sending messages to the new primary destination until the old primary destination informs it of the change, even if the source is already aware of the change. This prevents messages from a single source from getting delivered contrary to the order in which they were first sent.

Since sources must sometimes resend messages, we assume that each site maintains a message history - a log of messages received and sent - along with message contents. Such an assumption is not unreasonable for many applications, including a database, where recovery information is generally maintained anyway. The message history will be useful for reliability purposes, as well.

The solution presented here allows for concurrent incomplete *Opens*; for instance, a site may be added to a group, thereby initiating *Open* messages from the manager, before the new graph for a different site addition is opened everywhere. Due to communication delays, it is possible for sites to receive these *Opens* in different orders, since they are traveling via different graphs. To prevent confusion, the manager must number the graphs and include this with the *Open* and *Close* messages. When a site receives an *Open*, it must be sure that this is the *Open* for the next graph following the last graph closed. If not, the *Open* must be queued until all earlier graphs are opened and closed. In addition, all messages must include a graph number field to ensure that they are delivered following the *Open* of the correct graph. Any arriving messages that do not belong to the current graph are queued until the appropriate graph is opened.

It is not difficult to see that the multiple group ordering property is preserved using this algorithm. A two-level message ordering is established: the top level orders messages by the graph number and within the graph numbering messages are ordered by the propagation path of the tree. It is simple to verify that this hierarchical ordering is correct; we do not present the proof here.

Chapter Six

A Formal Model for Reliability

As mentioned in the Introduction, reliability of message delivery is important to many applications. A variety of causes, such as buffer overflow, can lead to message loss. In addition, many algorithms require that sites retrieve missed messages upon recovery from a failure, hence reliability includes some sort of recovery protocol for providing message delivery at all sites. In the course of analyzing the reliability of the propagation graph algorithm, we determined that there are many different degrees of reliability which a protocol might provide. Further, the issue of reliability is roughly orthogonal to that of ordering in that one can "mix and match" ordering and reliability properties. Also, in the course of comparing the reliability of the PG algorithm with other ordered multicast protocols, we have noted a wide disparity regarding the types of reliability the algorithms provide, even though many purport to solve the same problem. Also, we have noticed (even in some of our own work) a tendency to be vague and imprecise about what reliability is. To address these issues, we have worked out a formal model for ordering and reliability properties of multicasts in order to be more precise about the correctness of multicast protocols. The model provides a means for proving properties of multicasts and also lends insight into the difficulty of providing reliability. In addition, the model brings out some of the differences between protocols in terms of the assumptions made about the system, the properties they guarantee and their performance. We begin with a discussion of the conflicting (and imprecise) types of reliability several multicast algorithms have claimed to provide. We then compare our formal approach to other work that has been done on proving properties of distributed protocols. In the rest of the chapter we present the model and apply it to some simple multicast protocols. In the next chapter, we return to the propagation graph algorithm already studied.

6.1 Motivation and Previous Work

Reliable message delivery (with or without message ordering) has been defined in various ways for different algorithms. In [CM84], reliable broadcast (or multicast) is achieved with a protocol that "guarantees that all of the broadcast messages are received at all of the *operational* receivers in the broadcast group." When a site recovers it again becomes part of the broadcast group. [BJ87] considers processes instead of sites; a broadcast is "made to a set of processes" and is eventually "received by all operational destinations." The authors describe the failure detection mechanisms of the underlying ISIS system that determine whether a site (and hence its processes) is operational. But, contrary to token-passing, failed processes do not again become destinations upon recovery: "Once a process is observed to fail, it will never be heard from again." [SGS84] present a fault-tolerant broadcast protocol that ensures message delivery to the "functioning processors" in a system. But in their case a processor does more than just rejoin the broadcast group when it recovers, it also insists on getting all the messages it missed while down. It is not clear, however, what constitutes an "operational" or "functioning" site. If a site has been operational until the time all other sites receive a message, should it also receive the message? If not, how much longer must it stay up to be considered "operational?" Many protocols besides the two-phase solution of [BJ87] use the word "eventually" to describe reliability of message delivery. This led us to wonder how soon "eventually" is. If a protocol manages to deliver a message by next week, is it reliable? Certainly not, nor certainly do the authors intend this to be reliability. Our model attempts to specify more precisely when a message can be guaranteed to end up at its destinations.

Despite the imprecision noted above, the reliability alternatives are useful since different applications may require different degrees of reliability. Consider, for instance, one site that wishes to be elected leader using majority voting, so sends messages to all the sites. The fact that some sites never receive the message may be irrelevant if the originator manages to collect a majority. On the other hand, if an update message is being sent to all sites with a copy of the data, the update should be applied at all sites. If a site is down when the message is sent, the message should be obtained upon recovery. Often, for performance reasons, we would like to temporarily deliver messages in an inconsistent order and then redeliver them later. This may be the case, for instance, at a site that has just delivered a series of messages that no other site has and then fails. If these are updates, we can let the operational sites go on with processing and then roll back the messages (hence the updates) at the recovered site.

At first glance, the problems of ordering messages and guaranteeing reliability seem similar to the problem of reaching agreement in distributed systems, as treated by work on commit protocols [Gray78] and Byzantine agreement [LSP82]. However, there are several important differences. For one, whereas in a commit protocol a site votes "yes" or "no", with message delivery there is no reason to refuse a message. Sites agree on *when* to deliver, not *if*. Commitment also implies that the operation is an atomic one. Message ordering is relevant for applications with less stringent requirements; it may be practical and meaningful to allow a site, because of a failure, to miss messages or deliver them incorrectly (temporarily), as mentioned above. This allows us to explore various reliability alternatives. Further, in the sense that sites are achieving agreement, each decision does not stand alone. There is a sequence of agreements among which certain ordering properties must hold. This is not a concern of commit protocols and Byzantine agreement solutions. It is true, though, that delivering a message in a reliable, orderly way requires agreement among the deliverers [MMA90]. We propose, however, that if we are given reasonable assumptions about the network, then there are guarantees that can be made about sites receiving the messages sent to them. Contrary to work in [FLP85,MMA90b], instead of proving what can never be done (e.g., agreement cannot be reached) we provide a means to prove what can be done under favorable, but reasonable, conditions.

Our work on performance evaluation is unique and valuable for several reasons. First, since message ordering involves a continuum of decisions, rather than distinct agreements, we do not measure the delay of a protocol merely in terms of how long it takes to decide where in the message ordering a message belongs. More important, we think, is when a message can actually be delivered to its destination process. To address this, we consider how earlier messages can interfere with the delivery of a new message. Second, our treatment does not measure performance simply on the basis of rounds of message exchange. It separates the measure of processing time from the measure of network time and considers both. This is relevant since, as mentioned in [Gray88], for small high-speed networks, the cost of communication does not overwhelm the cost of processing to the degree it does in larger networks. Hence, processing time is not insignificant. Also, in networks where a physical multicast is accomplished by a series of unicasts, a site responsible for multicasting to a large number of destinations may require substantial processing time. Further, our model is flexible enough to allow us to analyze the performance of algorithms under a variety of conditions. For example, we can consider how well an algorithm performs when no failures occur during its operation and also when failures do occur.

Overall, with this work we attempt to model the message delivery process and formally define a spectrum of ordering and reliability guarantees. We do not exhaust the possibilities, but we do demonstrate how providing specific types of ordering and reliability affects the efficiency of a protocol and the assumptions required to make it work. In a sense, we are describing correctness criteria. In addition, we are attempting to be clear about what is required from the computing environment in order for delivery to meet the reliability requirements. We also use the model to analyze the performance of multicast protocols. Though we do formally define the three ordering properties mentioned above, we focus on formalizing the spectrum of reliability alternatives. We begin by presenting the model and then defining various guarantees multicast protocols can provide. Next, we apply the model to some simple algorithms for multicasting. As we mentioned earlier, we return to the algorithm studied in previous chapters in Chapter 7.

6.2. The Model

Before presenting formal definitions for ordering and reliability, we establish a model for message delivery. We define a multicast message m as a triple $[s,d,n]$ where s is the source, d is the destination group and n is a systemwide-unique identifier for the message. (The identifier n may bear no relation to the order of messages, but rather, it may simply make a message unique. It could be a concatenation of a unique identifier for s and a sequence number for all messages from s to d.) We use *Group(m)* to denote the set of sites comprising the destination multicast group, d. A site that receives a multicast is responsible for *deliver*ing it to the destination process (running locally). For simplicity, we assume one such process per site.

The system is modelled in terms of events occurring at sites. A global clock is used to specify when events happen. At any one site, no two events occur concurrently, but multiple events can occur at the same time at different sites. (It is important to note that the clock is used merely for the purposes of the model and is not needed by the algorithms that provide ordering and reliability.)

We use $\langle E \rangle_a$ to represent event E at site a. Event E may occur at more than one site. For example, if event E is the delivery of message m and two sites, a and b, deliver m then E has occurred at both sites. The delivery at b is represented by $\langle E \rangle_b$. $T\langle E \rangle_a$ is the time at which E occurs at a. If $T\langle E \rangle_a < T\langle E' \rangle_a$ then the time of event E at a is earlier than the time of event E' at a. If $T\langle E \rangle_a < T\langle E \rangle_b$ then E happened at a before E at b. If $T\langle E \rangle_a = T\langle E \rangle_b$ then E at a and E at b happen at the same time. In this case, $a \neq b$.

Events at one site are totally ordered by the time at which they occur. $\langle E \rangle_a \Rightarrow \langle E' \rangle_a$ iff $T\langle E \rangle_a < T\langle E' \rangle_a$. A partial order can be established for all events in the system. $\langle E \rangle_a \rightarrow \langle E \rangle_b$ iff $T\langle E \rangle_a < T\langle E \rangle_b$. The model thus far is similar to that presented in [Lamp78].

Since we are interested in the relative order of message delivery at the sites, we define an event ordering predicate, O. For two events E and E', both occurring at a and b, $O(\langle E \rangle_a, \langle E' \rangle_a, \langle E \rangle_b, \langle E' \rangle_b) = \mathit{true}$ if the order of the two events at the two sites is the same:

$O(\langle E \rangle_a, \langle E' \rangle_a, \langle E \rangle_b, \langle E' \rangle_b) = \mathit{true}$ when

$$\langle E \rangle_a \rightarrow \langle E' \rangle_a \ \mathit{iff} \ \langle E \rangle_b \rightarrow \langle E' \rangle_b .$$

There are eight types of events that can occur at a site.

$\langle M(m)\rangle_s$	site s initiates a multicast of message m;
$\langle S(m)\rangle_g^a$	site g sends message m to site a;
$\langle G(m)\rangle_g$	site g receives m;
$\langle D(m)\rangle_g$	site g delivers message m to the destination process;
$\langle F^i\rangle_g$	site g fails for the i^{th} time;
$\langle R^i\rangle_g$	site g recovers from failure i;
$\langle A^i\rangle_g$	site g awakens after failure i.

The distinction between an M event and an S event is important. When $\langle M(m)\rangle_s$, then s has initiated a multicast. Site s is the source. The actual physical transmission of m involves one or many S events, possibly not all of which will occur at s. For convenience, we will say that $\langle M(m)\rangle_s$ occurs after s has performed $\langle S(m)\rangle_s^{d_i}$ for all d_i to which it is required (by the protocol under consideration) to send m. The difference between a G event and a D event is also important. When $\langle G(m)\rangle_g$, then site g is the recipient of message m, but this does not imply that m has been delivered to the destination process. $\langle D(m)\rangle_g$ is the event that indicates the destination process executing at g has been given m. A delay at site g between $\langle G(m)\rangle_g$ and $\langle D(m)\rangle_g$ may be due to the multicast protocol in use. Even in a very simple protocol where sites deliver messages in a sequence number order, if a site g receives message m it may not be able to deliver it if it is missing the message(s) that should be delivered before m. The recovery and awake events distinguish between when a node is again functioning after a failure (awake) and when it can resume normal operation (recovered). Note that for failure i, $T\langle F^i\rangle_g < T\langle A^i\rangle_g < T\langle R^i\rangle_g$.

It is convenient to know *if* a message is ever multicast, sent, received or delivered. We define the following:

$M_s(m) = true$ if $T\langle M(m)\rangle_s < \infty$

$S_g^a(m) = true$ if $T\langle S(m)\rangle_g^a < \infty$

$G_g(m) = true$ if $T\langle G(m)\rangle_g < \infty$

$D_g(m) = true$ if $T\langle D(m)\rangle_g < \infty$

Sometimes it is convenient to allow more than one delivery of the same message at the same site. The deliveries of a particular message m are indexed by referring to the i^{th} delivery event of a message m at site g: $\langle D(m)\rangle_g^i$. We let $\langle D(m)\rangle_g^1$ be synonymous with $\langle D(m)\rangle_g$.

6.3. Ordering and Reliability Guarantees

In order to make some guarantees on message delivery, we need to consider various states of the processors and network. For instance, it is obvious that we cannot guarantee delivery of a message at a site whose behavior is erratic, i.e., it goes down often. The same applies to the network. If the network cannot remain up long enough to deliver a message, then message delivery cannot be guaranteed. Also, if a processor or a link is especially slow, only weak properties of delivery can be guaranteed. To explore the spectrum of message delivery we need to define some states for our system. To simplify matters, we divide the system into two components: the network responsible for delivering messages and the processors receiving and delivering messages.

For processors, we need to define what constitutes a down, or halted, processor. We say that there exists a constant ϕ_P for processors that represents the time required to perform a message operation. These operations are broad, e.g., they may include delivering one message, delivering a series of messages, delivering a backlog of messages plus a new one, etc. When a processor, P_i, can no longer meet ϕ_P, then that constitutes a failure event, $\langle F^j \rangle_i$, the j^{th} failure at P_i. When processor P_i can again meet ϕ_P, it awakens, i.e., $\langle A^j \rangle_i$ occurs. Further, we define a period of being awake.

AWAKE(g,τ,δ) = true if

(i) there is no $\langle F^i \rangle_g$ such that $\tau \leq T \ \langle F^i \rangle_g \leq \tau + \delta$ and

(ii) $T \ \langle F^i \rangle_g < \tau$ implies $T \ \langle A^i \rangle_g \leq \tau$.

A site that is awake is capable of executing on behalf of messages within ϕ_p, but it may not be allowed to execute for all processes, e.g., it may need to do some recovery work. (What a site can do in the AWAKE state is algorithm dependent.) Stronger than AWAKE is UP.

UP(g,τ,δ) = true if

(i) there is no $\langle F^i \rangle_g$ such that $\tau \leq T \ \langle F^i \rangle_g \leq \tau + \delta$ and

(ii) $T \ \langle F^i \rangle_g < \tau$ implies $T \ \langle R^i \rangle_g \leq \tau$.

In addition to processor execution time, we also consider a network that is operational to be one that can provide delivery of a message from sender a to receiver b within ϕ_N. When the network cannot make that guarantee, it is not operational. Similar to UP and AWAKE, we define a time period in which two sites can send each other messages that get delivered within ϕ_N.

REACH $(a, b, \tau, \delta) =$ *true* if *AWAKE* $(a, \tau, \delta + \phi_N)$ and *AWAKE* $(b, \tau, \delta + \phi_N)$ imply

(i) if $\tau \leq T \ \langle S(m) \rangle_a^b \leq \tau + \delta$ then $T \ \langle G(m) \rangle_b \leq T \ \langle S(m) \rangle_a^b + \phi_N$.

(ii) if $\tau \leq T \ \langle S(m) \rangle_b^a \leq \tau + \delta$ then $T \ \langle G(m) \rangle_a \leq T \ \langle S(m) \rangle_b^a + \phi_N$.

Now, using UP and REACH (the state of the processors and the network) we can provide guarantees on delivery. The properties are divided into 3 categories: P, Q and R. The P properties make guarantees on delivery of a multicast message m at a destination a given that the source has initiated the multicast of m. The P properties mostly fall into two types: those that provide some guarantee based on the time a message is multicast from the source and those that provide a delivery guarantee based on when the message is received at the destination, a. The Q properties describe message delivery guarantees among group members, that is, they address atomicity of delivery. The R properties define guarantees on consistent delivery orders at the destinations. Although the properties are numbered, there is not a strict hierarchy of property strength; hence, the numbering does not reflect any such hierarchy.

We begin with the P properties. P0 is the weakest P property we present and P2 is the strongest. The others fall in between, but cannot be strictly ordered by strength. To determine the relative strength of a property, we consider from what action the property can guarantee bounded message delivery. For instance, P2 can guarantee delivery based on the original multicast from the source. P1 is weaker because it only guarantees delivery at a destination based on when the message is received (i.e., the get event occurs) at the destination. The properties are not totally ordered because they apply to different types of protocols; for instance, some apply to agreement protocols and others apply to protocols where the ordering decision is centralized. The first P property we present guarantees nothing (as some protocols may).

P0: Let m be a message such that $M_s(m)$ and $a \in$ *Group(m)*. No guarantee is made on the delivery of m at a.

The next two properties are stronger. They provide a delivery bound. P1 is weaker than P2 since its bound is based on when the multicast is received at the destination, rather than based on when it is sent by the source.

P1: Let m be a message such that $M_s(m)$ and $a \in Group(m)$. There exists a constant Δ such that if $T\langle G(m)\rangle_a = \tau$, UP($a,\tau,\Delta$), then $T\langle D(m)\rangle_a \le \tau + \Delta$.

P2: Let m be a message such that $\tau = T\langle M(m)\rangle_s$ and $a \in Group(m)$. There exists a constant Δ such that if UP(a,τ,Δ), AWAKE(s,τ,Δ) and REACH(a,s,τ,Δ) then $T\langle D(m)\rangle_a \le \tau+\Delta$.

The next property is used when sites must be able to contact other destination sites in order to deliver messages. For example, if a majority commit is needed to deliver a message, P3 may apply.

P3: Let m be a message such that $M_s(m)$ and $a,b_1,b_2,...,b_N \in Group(m)$. There exist constants Δ and N such that if $T\langle G(m)\rangle_a = \tau$, UP($a,\tau,\Delta$), AWAKE($b_i,\tau,\Delta$) and REACH($a,b_i,\tau,\Delta$) for all b_i then $T\langle D(m)\rangle_a \le \tau + \Delta$.

Property P4 provides a time bound that is based on when a destination site is *sent* a message by some site, rather than when it is multicast from the source. For instance, some protocols propagate messages through a series of sites in order to deliver the message at all destinations. A delivery guarantee at one site, a, is based on its ability to communicate with the site that sent the message, x. (Note: x may be s.)

P4: Let m be a message such that $M_s(m)$ and $T\langle S(m)\rangle_x^a = \tau$ for $a \in Group(m)$. There exists a constant Δ such that if UP(a,θ,Δ), AWAKE(x,θ,Δ) and REACH(a,x,θ,Δ) for $\theta \geq \tau$ then $T\langle D(m)\rangle_a \leq \theta + \Delta$.

The Q properties provide atomicity of delivery, that is, guarantees among group members. Again, different Q properties apply to different types of protocols (just as the P properties do) so it is not possible to order them strictly from weakest to strongest. But there is some sense of a strong Q property. Strength here is based on which conditions must be true in order to guarantee atomic delivery. Q0 is very weak. Q3 requires that a site be able to communicate with a distinguished site - its parent. Q2 is similar in that the destination must be able to communicate with the source. Q1 is strongest in that the delivery bound at a destination is based on when any other group member delivers the message. Q0, the first property, provides no atomicity guarantee.

Q0: Let m be a message such that $a,b \in$ *Group(m)*. Even if $D_a(m)$, no guarantee can be made so that $D_b(m)$.

Property Q1 guarantees bounded delivery of a message at group member b given that the message was delivered at a and a and b can communicate. The bound is based on when the message is delivered at a.

Q1: Let m be a message such that $a,b \in$ *Group(m)*. There exists a constant ρ such that if $T\langle D(m)\rangle_a = \theta$, if UP($b$,$\theta$,$\rho$), AWAKE($a$,$\theta$,$\rho$) and REACH($a$,$b$,$\theta$,$\rho$) then $T\langle D(m)\rangle_b \leq \theta + \rho$.

Q2 is weaker in that its bounded delivery is not based on when a delivers the message, but rather the time at which b can begin to communicate long enough with the source to get the message delivered at b. It is used for protocols in which the source is solely responsible for getting messages to the group members.

Q2: Let m be a message such that $T\langle M(m)\rangle_s = \tau$ and $a,b \in Group(m)$. There exists a constant ρ such that if $D_a(m)$, UP(b,θ,ρ), AWAKE(s,θ,ρ) and REACH(s,b,θ,ρ) for $\theta \geq \tau$, then $T\langle D(m)\rangle_b = \theta + \rho$.

For the next Q property we need a definition. Many algorithms make use of logical paths for propagating messages, as mentioned for property P4. For instance, the protocol may specify that a message must start at site a, be sent from a to b and then sent from b to c. In such a case, we might consider a to be the parent of b and b to be the parent of c. For such algorithms, we may be able to make a Q guarantee based on the ability of a parent and child to communicate. We refer to the "parent" of a site, say b, as $p(b)$.

Q3: Let m be a message such that $M_s(m)$, $D_a(m)$ and $a,b \in Group(m)$. If $T\langle D(m)\rangle_{p(b)} = \alpha$ then if UP(b,θ,ρ), AWAKE($p(b),\theta,\rho$) and REACH($b,p(b),\theta,\rho$) for $\theta \geq \alpha$ then $T\langle D(m)\rangle_b \leq \theta + \rho$.

The R properties describe ordering guarantees. As mentioned in the introduction, there are several ordering properties that may be required for messages. For two messages, m and m', consistent ordering may be required only if m and m' originate at the same source and are destined to the same group. Or ordering may be necessary if the messages originate at different sources or even if they are destined for different groups. Since the reliability alternatives we describe here are applicable to any of the ordering properties, we generalize these options with the message predicate P. For two messages $m = [s,d,n]$ and $m' = [s',d',n']$, if $P(m,m')$ is true then m and m' are messages that must be ordered consistently. What makes P true depends on which ordering property is required.

(1) *Single Source Ordering*:

$P(m, m')$ is true if $((s = s') \wedge (d = d'))$.

(2) *Multiple Source Ordering*:

$P(m, m')$ is true if $(d = d')$.

(3) *Multiple Group Ordering*:

$P(m, m')$ is true for all m and m'.

For the purposes of presenting the R properties, we do not consider a particular ordering property; rather, we refer to the predicate P.

The R properties presented here can be readily ordered from weakest to strongest. R0 is very weak and guarantees no consistent ordering among sites. R2 is next, and guarantees ordering as long as a site remains up; that is, once a site fails its message delivery order is irrelevant to the remaining sites. R3 is stronger still - it guarantees consistent ordering at all sites since their last recovery event. R1 is the strongest - it guarantees consistent ordering at all sites that have delivered the messages, at any time.

R0: If $P(m, m')$ and $D_a(m)$, $D_a(m')$, $D_b(m)$, $D_b(m')$ for a,b in *Group(m)*, no guarantee is made on the relative order of the deliveries of m and m' at a and b.

R1:

(1) If $P(m,m')$ and $D_a(m)$, $D_a(m')$, $D_b(m)$, $D_b(m')$ for a,b in *Group(m)*, then $O(\langle D(m)\rangle_a, \langle D(m')\rangle_a, \langle D(m)\rangle_b, \langle D(m')\rangle_b)$.

(2) There is no $\langle D(m)\rangle_a^i$, $\langle D(m')\rangle_a^i$, $\langle D(m)\rangle_b^i$ or $\langle D(m')\rangle_b^i$ where $i > 1$.

R2:

(1) If $P\ (m,m')$, $D_a(m)$, $D_a(m')$, $D_b(m)$, $D_b(m')$ for a,b in *Group(m)*, and $\langle D(m)\rangle_a \rightarrow \langle D(m')\rangle_a$, then $\langle D(m)\rangle_b \rightarrow \langle D(m')\rangle_b$ unless

  (a). $\langle D(m)\rangle_b \rightarrow \langle F^i\rangle_b \rightarrow \langle D(m)\rangle_a$ for some i or

  (b). $\langle D(m')\rangle_a \rightarrow \langle F^j\rangle_a \rightarrow \langle D(m')\rangle_b$ for some j.

(2) There is no $\langle D(m)\rangle_a^i$, $\langle D(m')\rangle_a^i$, $\langle D(m)\rangle_b^i$ or $\langle D(m')\rangle_b^i$ where $i > 1$.

A definition is required for R3. The following function defines $F_g(m,t)$ to be the latest delivery (at time t) of message m at g.

$$F_g(m,t) = \langle D(m)\rangle_g^i \text{ where there is no } \langle D(m)\rangle_g^j \text{ such that } \langle D(m)\rangle_g^i \rightarrow \langle D(m)\rangle_g^j \text{ and } \langle D(m)\rangle_g^j < t.$$

This function may be undefined for some values of t, i.e., there may be no delivery of m at g before time t.

R3: If $P(m,m')$ is true, then at any time t for which $F_a^m(t)$, $F_a^{m'}(t)$, $F_b^m(t)$, $F_b^{m'}(t)$ are defined, if UP(a,t,0) and UP(b,t,0) are true, then $O\,(F_a^m(t), F_a^{m'}(t), F_b^m(t), F_b^{m'}(t))$.

6.4. Multicast Protocols

With well-defined properties available to us, we can analyze multicast protocols in terms of the reliability they provide and their performance characteristics. Each protocol is described as a series of code fragments. All fragments are assumed to represent code that can execute within ϕ_P if the site is UP. As mentioned earlier, if ϕ_P cannot be met with the execution of a code fragment, then the site has failed. In terms of the model, typically this simply means that a guarantee cannot be made on delivery. Most protocols include a set of assumptions, numbered in square brackets. At least one P, Q, and R property is shown to be true for each protocol. The protocol consists of a series of code fragments (labeled within parentheses), which we assume are executable within ϕ_P. The fragments include lines of code, plus indication of the occurrence of formal events such as the multicast, send, get or delivery of a message. These events appear in bold face to distinguish them from code. They do not represent an action on the part of the site, but rather show the formal result of the previous action.

We begin with a simple protocol that does not guarantee consistent ordering. (It guarantees R0.) It serves to illustrate two points: (1) the model is straightforward to use; (2) even very simple tasks seem to require strong system characteristics if they are to guarantee strong reliability properties.

6.4.1 Protocol 0

Here is a very simple protocol that guarantees properties P1, Q0 and R0. It consists merely of sources multicasting to groups. Whenever a site receives a message, it simply delivers it.

```
(A) When a message $m = [S, D, n]$ is received at $a$:

        $\langle G(m) \rangle_a$ occurs
        deliver message $[S, D, n]$;
        $\langle D(m) \rangle_a$ occurs
```

```
(B) At $s$, to multicast $m = [s, D, n]$:

        for each $g \in Group(m)$
            send $[s, D, n]$ to $g$;
        $\langle M(m) \rangle_s$ occurs
```

It is easy to see that Protocol 0 guarantees P1, Q0 and R0. To show that P1 is guaranteed, consider message m such that $M_s(m)$, $a \in Group(m)$ and $\tau = T\langle G(m) \rangle_a$. The bound Δ is simply ϕ_P. If a stays up for ϕ_P time, code fragment A will execute and the message will be delivered.

As presented, Protocol 0 guarantees that if the destination site, a, stays up long enough to execute code fragment (A) after receiving message m, then m is delivered at a within the Δ bound. It seems desirable to be able to guarantee some bound based on when the source sends m, not just when a receives m. Such a guarantee is given in P2. In fact, Protocol 0 guarantees P2 also. But note that delivery within a bound in P2 requires some network conditions. It is not sufficient for just a to be UP. The source must be AWAKE and the link between them must be operational. Thus, the conditions under which the stronger property holds are stronger.

6.4.2 Protocol 1

As mentioned above, Protocol 0 does not provide an ordering guarantee. It also does not prevent multiple delivery of the same message at the same site (though some networks provide this service). The next protocol provides a stronger ordering (R) property to address these problems. Observe also that Protocol 0 does not provide atomicity of message delivery among group members, as indicated by the weak Q property. We approach this problem in later protocols.

We call our first non-trivial protocol the fan algorithm. It consists of a single source s multicasting to a single multicast group $D = d_1$ through d_n. Thus, single source ordering is the goal. To multicast a message, the source is responsible for sending the message directly to each member of the destination group (see code fragment D below). This protocol guarantees P2, Q0 and R1. To ensure ordering property R1, the source adds sequence numbers to all messages.

We make the following assumptions.

[A1] The source node S never fails, i.e., $UP(s,0,\infty) = true$. Destination nodes, however, are allowed to fail.

[A2] There are no communication failures. That is, for all pair of nodes x, y, $REACH(x,y,0,\infty)$. As given in the definition of $REACH$, the upper bound on message delivery time is ϕ_N. Note that messages may be delivered out of order by the network and duplicate messages may arrive.

[A3] There is stable storage available.

We relax assumption [A2] in the next protocol.

Given these assumptions, the following algorithm guarantees P2, Q0 and R1.

Storage Requirements: The algorithm keeps track of the sequence number of the last delivered message in variable L, initialized to 0. It stores pending messages, those received but not delivered (to the application) in a set P. Variable L is kept in stable storage (available after a failure). It is not necessary to keep P in stable storage. The source maintains a sequence number for messages sent to the group. This sequence number is represented by n for message $[S,D,n]$. Since there is just a single group and source, S and D are constant over all messages.

(A) When a message $m = [S, D, n]$ is received at d_j and d_j is UP, the following is done:

```
$\langle G(m) \rangle_{d_j}$ **occurs**
if $n = L+1$ then begin
        deliver message $[S, D, n]$;
        $\langle D(m) \rangle_{d_j}$ **occurs**
        $L \leftarrow L + 1$;
        while message $[S, D, L+1]$ is in $P$ do begin
                remove message $[S, D, L+1]$ from $P$;
                deliver message $[S, D, L+1]$;
                $\langle D(m) \rangle_{d_j}$ **occurs**
                $L \leftarrow L + 1$; end end
else if $n > L+1$ then add $[S, D, n]$ to $P$;
else ignore message;
```

(B) After a node d_j awakens for the i^{th} time, the following takes place:

```
**event** $\langle A^i \rangle_{d_j}$ **occurs with** $T\langle A^i \rangle_{d_j} = \beta$
/* note that $L$ still represents that last message delivered */
```

(C) When a message $m = [S, D, n]$ is received at d_j and d_j is AWAKE but not UP:

```
if $\phi_N$ has elapsed since $\beta$ and $n > L$ then begin
        $L \leftarrow n - 1$;
        recover from $F^i$, where $F^i$ is the latest failure;
        **event** $\langle R^i \rangle_{d_j}$ **occurs for** $F^i$**, the latest failure**
        process $[S, D, n]$ and following messages as in part (A) above; end
else begin
        $\langle G(m) \rangle_{d_j}$ **occurs**
        place $[S, D, n]$ in $P$; end
```

(D) At s, to multicast message $m = [s, D, N]$:

```
for each $g \in Group([s, D, N])$ do
        send $[s, D, N]$ to $g$;
$\langle M(m) \rangle_s$ **occurs**
```

This algorithm guarantees properties P2, Q0, and R1. To see that property R1 is guaranteed, simply note that a node only delivers messages in strict message order (L is not lost because of failures). So, messages that are delivered must be delivered in the same order. Similarly, messages can never be redelivered.

To show P2, we must consider a message $m = [s, D, n]$ such that $T\langle M(m)\rangle_s = \tau$ and node $a \in Group(m)$. We must then show there is a constant Δ such that if $UP(a, \tau, \Delta)$, then m is delivered at a by time $\tau + \Delta$. We will prove that the constant Δ exists and equals $\phi_N + \phi_P$.

A simple induction on n, the message identifier, proves that our bound Δ is correct if sites never fail.

Basis Clearly $[s, D, 1]$, where $\tau = T\langle M(m)\rangle_s$, is delivered by $\tau + \phi_N + \phi_P$. The network guarantees receipt of $[s, D, 1]$ at a by ϕ_N. (There are no communication failures.) When code fragment B executes, $n = L+1$ (L is initialized to 0), so $[s, D, 1]$ is delivered immediately. Code fragment B requires ϕ_P time.

I.S. Assume that message $[s, D, n-1]$ (multicast at $\tau_{n-1} = T\langle M([s, D, n-1])\rangle_s$) is delivered by $\tau_{n-1} + \phi_P + \phi_N$. Note that $\tau_n > \tau_{n-1}$. The network guarantees receipt of $[s, D, n]$ at a by $\tau_N + \phi_N$. (Again, there are no communication failures.) If $n = L+1$ when code fragment B executes, then a delivers $[s, D, n]$ by $\tau_n + \phi_N + \phi_P$. Otherwise, $[s, D, n]$ is placed in P and will be delivered as part of the execution of code fragment B that delivers $[s, D, n-1]$. But, the execution of B that delivers $[s, D, n-1]$ completes by $\tau_{n-1} + \phi_N + \phi_P$ and $\tau_n > \tau_{n-1}$. Thus, $[s, D, n]$ is delivered by $\tau_n + \phi_N + \phi_P$. $\square$

To show that delivery holds in the presence of site failures, we show that all messages sent within a period in which site a is up are delivered correctly at a. (Recall that there are no communication failures as assumed in [A2].)

Basis Consider $[s,D,x]$, such that $T\langle M([s,D,x])\rangle_s = \tau$, the first message multicast from s after a recovers from failure event $\langle F^i \rangle_a$. Say that $\langle A^i \rangle_a$ occurs at β. We must prove that $[s,D,x]$ is delivered at a within $\tau + \phi_N + \phi_P$. When $[s,D,x]$ arrives, a is up due to the receipt of an earlier message, $[s,D,y]$ that arrived after $\beta + \phi_N$. Thus, $x > y$ since y was sent before x. Message $[s,D,y]$ is delivered immediately since L is set to $y-1$. In addition, all messages $[s,D,z]$ such that $y<z<x$ received before $[s,D,y]$ have been saved in P. Say that $[s,D,y]$ arrived at γ. Then, any message that arrived before γ, but sent after $[s,D,y]$ must have arrived after $\gamma - \phi_N$, due to the network ϕ_N delivery guarantee. Note that $\gamma > \beta + \phi_N$. Thus, a was awake and saved the message in P. Any other messages $[s,D,z]$ $(z < x)$ must arrive no later than $\tau + \phi_N$. Say the last such $[s,D,z]$ does arrive at $\tau + \phi_N$. Then all messages, including $[s,D,x]$ are delivered using code fragment B within ϕ_P of $\tau + \phi_N$, if not before.

I.S. The inductive step is identical to that of the non-failure case. The only messages considered, however, are those sent by s between an R event and an F event at a. □

6.4.3 Protocol 2

In this protocol, we amend Protocol 1 so that communication failures can be tolerated. If a message arrives at d_j out of sequence, d_j requests retransmission of earlier messages. This protocol guarantees P2, Q0 and R1. We use assumptions [A1] and [A3] from Protocol 1.

Storage Requirements: The algorithm keeps track of the sequence number of the last delivered message in variable L, initialized to 0. It stores pending messages, those received but not delivered (to the application) in a set P. Variable L is kept in stable storage (available after a failure). It is not necessary to keep P in stable storage. The source maintains a sequence number for messages sent to the group. This sequence number is represented by n for message $[S,D,n]$. Since there is just a single group and source, S and D are constant over all messages.

(A) When a message $m = [S,D,n]$ is received at a, the following is done:

```
$\langle G(m) \rangle_a$ **occurs**
if $n = L + 1$ then begin
    deliver message $[S,D,n]$;
    $\langle D(m) \rangle_a$ **occurs**
    $L \leftarrow L + 1$;
    while message $[S,D,L+1]$ is in $P$ do begin
        remove message $[S,D,L+1]$ from $P$;
        deliver message $[S,D,L+1]$;
        $\langle D(m) \rangle_a$ **occurs**
        $L \leftarrow L + 1$; end end
else if $n > L+1$ then begin
    add $[S,D,n]$ to $P$
    for all messages $[S,D,x]$ such that $[S,D,x]$ not in $P$ and $L < x < n$
        request retransmission of $[S,D,x]$ from $S$; end
else ignore message;
```

(B) After a node d_j awakens for the i^{th} time, the following takes place:

```
$\langle A^i \rangle_{d_j}$ **occurs with** $T\langle A^i \rangle_{d_j} = \beta$
/* note that $L$ still represents that last message delivered */
```

(C) When a message $[S, D, n]$ is received at d_j and d_j is AWAKE but not UP:

```
if $n > L$ then begin
    set $L \leftarrow n - 1$;
    recover from $F^i$, where $F^i$ is the latest failure;
    **$\langle R^i \rangle_{d_j}$ occurs where $F^i$ is the latest failure**
    process $[S, D, n]$ and following messages as in part (B) above; end
```

(D) At s, to multicast message $m = [s, D, N]$:

```
for each $g \in Group([s, D, N])$ do
    send $[s, D, N]$ to $g$;
**$\langle M(m) \rangle_s$ occurs;**
```

(E) At s, when a retransmission request is received from site a:

```
resend requested messages to $a$;
```

We show that Protocol 2 guarantees Q0, R1 and P2. Once again variable L, along with retransmission of missed messages guarantees R1. To prove P2, we must show that for a message $m = [s, D, n]$ such that $\tau = T\langle M(m) \rangle_s$ and $a \in Group(m)$, there exists a Δ such that if $UP(a, \tau, \Delta)$ and $REACH(a, s, \tau, \Delta)$, then $T\langle D(m) \rangle_a \leq \tau + \Delta$. We show that $\Delta = 3\phi_P + 3\phi_N$.

First we consider the case of no site failures. It takes ϕ_N for site a to receive m. When site a receives message m, either it is deliverable immediately (and Δ obviously holds) or site a informs the source that it is missing messages with sequence numbers less than n. It takes ϕ_P to generate the request for retransmission and ϕ_N for the network to transmit it to s. ϕ_P is required at s to send the missed message and ϕ_N more is needed to ensure they all arrive at a. ϕ_P is required for code fragment B to deliver the last of the messages to arrive. As part of the execution of fragment B, when $[s, D, n-1]$ is delivered, $[s, D, n]$ is taken from P and delivered. Thus, if site a does not fail within $\tau + 3\phi_P + 2\phi_N$

($UP(a,\tau,\Delta)$ is true) and it can contact s within the ϕ_N limit for that time period ($REACH(a,s,\tau,\Delta)$ is true), then the bound is correct.

The proof in the case of site failure is similar to that of Protocol 1. We consider the basis case where we must show that the first message to arrive after a recovers, $[S,D,x]$ where $\tau = T\langle M([S,D,x])\rangle_s$ is delivered within Δ of τ as long as the properties in P2 hold. It is not hard to see that $[S,D,x]$ will be delivered within the time indicated for the non-failure case. Site a will simply request previous messages if it is unable to deliver $[S,D,x]$ immediately.

(Note that although the spirit of Protocol 2 intends that a recovering site not bother to obtain missed messages sent while it was down, this may in fact happen. Because we do not use assumption [A2], it is possible that the first message to arrive initiating code fragment (C) is a message from the distant past - one perhaps sent while d_j was down. As part of fragment (C), L is set so that this newly arrived message will be delivered, as will all following messages for as long as d_j remains UP. Hence, d_j may recover some, and possibly all, of the messages sent while it was down. This does not violate any of the guarantees, but it does mean that more message get delivered than are required by P2 and Q0.)

6.4.4 Protocol 3

Protocol 3 is much like Protocol 2, except we allow a recovering site to obtain all messages missed while down. Thus, this protocol provides a stronger Q property: Q2. In order to achieve this atomicity, however, sites must be aware of having missed messages. In this protocol, a site that awakens from a failure recognizes it is missing messages by getting a message with a sequence number higher than it is expecting. (Here again the variable L keeps track of the expected sequence number.) But, in order to tell that the site is missing messages, it must receive *some* message with too high a sequence number. Thus, there must be a continual flow of messages from the source. Null messages are used to accomplish this. Once again, we use assumptions [A1] and [A3]. The storage requirements and code fragments (A), (B), (D) and (E) are the same as in Protocol 2. Here, we provide (C) and (F).

```
(C)  When a message $[S,D,n]$ is received at $d_j$ and $d_j$ is AWAKE but not UP:

     if $n > L$ then begin
          recover from $F^i$, where $F^i$ is the latest failure;
          **$\langle R^i \rangle_{d_j}$ occurs where $F^i$ is the latest failure**
          process $[S,D,n]$ and following messages as in part (B) above; end

(F)  At source, if no messages are multicast for $\delta$ seconds:

     multicast null message using (D);
```

It is not difficult to see that Protocol 3 guarantees P2 and R1 just as Protocol 2 does. (Δ is still $3\phi_P + 3\phi_N$.) We show that Q2 holds with $\rho = \delta + 3\phi_P + 3\phi_N$. Say that $\tau = T\langle M(m)\rangle_s$ where $m = [s,D,n]$. In accordance with Q2, say REACH(s,b,θ,ρ) and UP(b,θ,ρ). (The fact that a has delivered m is irrelevant in the proof.) We show that b delivers m within ρ of θ. There are two cases: (1) $\theta = \tau$; (2) $\theta > \tau$.

Case 1:

This case is equivalent to the proof of P2.

Case 2:

Within δ of θ, s must send b some message, even if just a null message, according to code fragment (F). Say this is message $m' = [s, D, n']$. Note $n' > n$ since $T\langle S(m')\rangle_s^b > \theta$. Since REACH($s, b, \theta, \Delta$) and UP($b, \theta, \Delta$) hold, b receives m' by $\delta + \phi_P + \phi_N$. Upon receiving m', b will request, receive and deliver missed messages, including m. As in the proof of P2, this requires $2\phi_P + 2\phi_N$. Thus, $\rho = \delta + 3\phi_P + 3\phi_N$. □

Chapter Seven

Reliability of Algorithms For Multicast Ordering

With a better understanding of reliability, we are prepared now to return to the protocols for ordered multicasts discussed in the earlier chapters; in particular, we can address the issue of reliability in the propagation graph algorithm. We begin by describing the reliability of the PG algorithm, the two-phase solution and the token-passing solution informally. We then apply the model to two versions of the propagation graph algorithm that provide different degrees of reliability.

7.1 Reliability - Informally

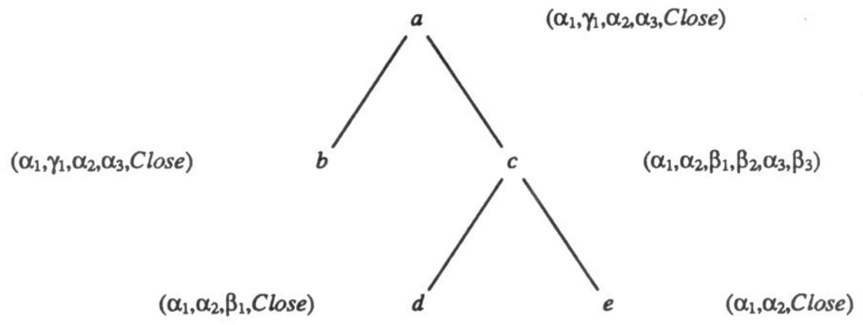
For this informal description of reliability, we make some simplifying assumptions. We assume that all failures are fail-stop [SS83] and there are no partitions. Thus, we assume that all operational sites will respond to messages in a reasonable amount of time and the network delivers messages in a reasonable amount of time. As a result, all failures are detectable. In addition, we assume only one failure occurs at a time and there is sufficient time between failures to run the required recovery procedure and resume normal operation. We address relaxing these assumptions after we discuss reliability of the PG algorithm formally.

As we mentioned in Chapter 6, there are many types of reliability a protocol can provide. Further, we believe that our approach can be made reliable to any desired degree. Here we present two reliability alternatives that mesh nicely with our ordering strategy. We call these two methods for handling failures *atomic ordered delivery* and *non-atomic ordered delivery*. Atomic ordered delivery guarantees that all sites receiving the same messages always deliver them in the same order, but in many cases failure forces sites to block. With this method, the propagation graph is not changed; rather, the other sites wait for the failure to be repaired. Blocking is avoided with the second alternative, but ordered message delivery is not atomic. Non-atomicity occurs infrequently, however, and only a failed site may have delivered messages in the wrong order. We begin with non-atomic delivery.

7.1.1 Non-atomic Delivery

With this alternative, we use a technique similar to that used for dynamic multicast groups. Sites constantly monitor the sites on the propagation graph from which they receive messages. Failures are detected via timeouts. If a child has not received a message from a parent within some predetermined time interval, the child assumes the parent has failed. If the parent has no messages to send, it sends a null message periodically to prevent false failure detection. If a failure is detected, a two-phase process is initiated among the survivors and the manager. In the first phase, the manager is informed of the failure and it closes the group involved. Since the graph may be broken, the manager may have to unicast *Close* messages directly to the survivors, without using the propagation graph. To ensure that all sites order the *Close* messages with the other messages consistently, when each site receives the *Close*, it stops processing messages and reports back to the manager its message history back to the last message it installed for some group. (Note this was not necessary when there were no failures since all messages used the propagation graph.) Knowing the last messages installed per group requires a *group sequence number*, which the primary destination is responsible for assigning to messages. Each new graph starts with the group sequence numbers initialized to 0.

The second phase requires each survivor to install any missing messages that the other sites report. To see how this works, consider the following example. The network has five sites and multicast groups $\alpha = \{a,b,c,d,e\}$, $\beta = \{c,d,e\}$ and $\gamma = \{a,b\}$. The propagation graph is shown in Figure 7.1, along with a history of message delivery at each site up until the time that site c fails, some site detects this and the manager, site a, sends out the *Close* messages. (The first message sent to group α is numbered α_1 by the primary destination, the second is numbered α_2, and so on.)



a
$(\alpha_1,\gamma_1,\alpha_2,\alpha_3,Close)$
$(\alpha_1,\gamma_1,\alpha_2,\alpha_3,Close)$
b
c
$(\alpha_1,\alpha_2,\beta_1,\beta_2,\alpha_3,\beta_3)$
$(\alpha_1,\alpha_2,\beta_1,Close)$
d
e
$(\alpha_1,\alpha_2,Close)$

Figure 7.1

To maintain availability, site c is temporarily omitted from α and β, forming $\alpha' = \{a,b,d,e\}$ and $\beta' = \{d,e\}$. The new tree, PG', is shown in Figure 7.2. Before message processing can resume, all sites in PG' must have consistent message delivery histories. In our example, d and e must deliver α_3 and e must deliver β_1. To accomplish this, the manager determines which messages every site is missing using the group sequence numbers and informs each site, by sending messages down the new tree, of what the history should be (determining a total order if there is not one already implied by the message histories) and which site can provide each missing message.

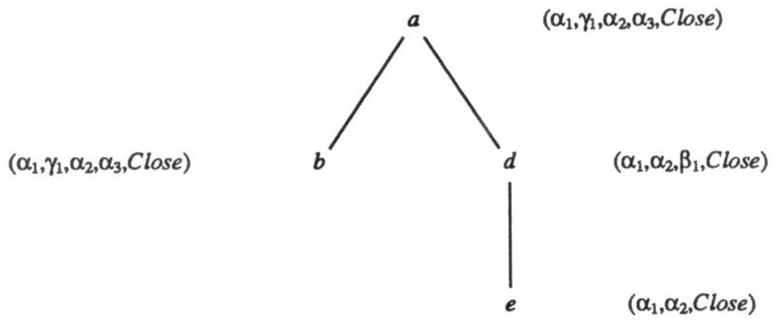
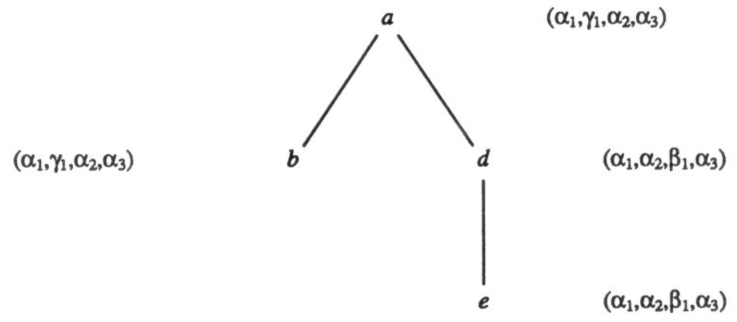

a
($\alpha_1,\gamma_1,\alpha_2,\alpha_3,Close$)
($\alpha_1,\gamma_1,\alpha_2,\alpha_3,Close$)
b
d
($\alpha_1,\alpha_2,\beta_1,Close$)
e
($\alpha_1,\alpha_2,Close$)

Figure 7.2

When this catch up phase is completed, the live sites will have consistent message delivery orders and all live members of the same group will have delivered the same messages. We get Figure 7.3. (The merge of the *Close* messages can be disregarded at this point.) The same is not true of failed sites. Note that site *c* has delivered messages that *d* and *e* have not (β_2 and β_3). Since the message information at site *c* is not available, sites *d* and *e* are not aware of β_2 and β_3. To prevent new messages from the source(s) of β_2 and β_3 from getting delivered at *d* and *e* before β_2 and β_3, a *source sequence number* is used. Each source maintains a sequence number for each group to which it sends messages. When *d* receives another message from the source(s) of β_2 and β_3, it determines that it missed some messages and asks to have them resent. Sources learn of the new primary destination by checking with the manager after detecting the failure of site *c*.

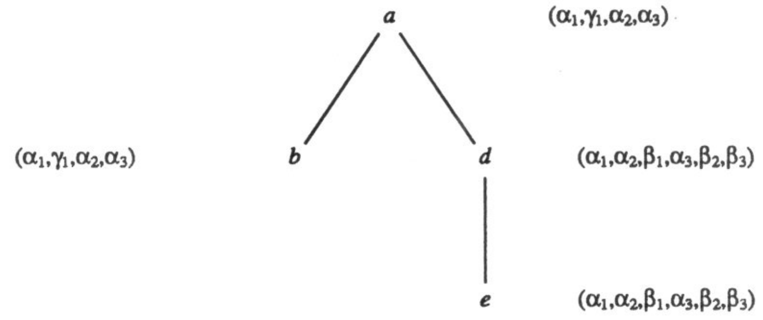
Of course, although site *d* eventually receives β_2 and β_3, it is too late to have them delivered before α_3, as *c* did. Instead, site *d* finds a new spot in the message history for these messages, as indicated in Figure 7.4. Thus, if and when *c* revives, the tail end of its message deliveries are out of order with respect to the other sites. If required, site *c* can rollback delivery of β_2 , α_3, and β_3. Site *c* may redeliver them, using the order defined by



a
$(\alpha_1,\gamma_1,\alpha_2,\alpha_3)$
$(\alpha_1,\gamma_1,\alpha_2,\alpha_3)$
b
d
$(\alpha_1,\alpha_2,\beta_1,\alpha_3)$
e
$(\alpha_1,\alpha_2,\beta_1,\alpha_3)$

Figure 7.3

d along with any other messages it has missed while down. Some applications may not require such rollback and recovery of missed messages. In that case, site *c* can just rejoin the tree and start over at the current message.



a
$(\alpha_1,\gamma_1,\alpha_2,\alpha_3)$
$(\alpha_1,\gamma_1,\alpha_2,\alpha_3)$
b
d
$(\alpha_1,\alpha_2,\beta_1,\alpha_3,\beta_2,\beta_3)$
e
$(\alpha_1,\alpha_2,\beta_1,\alpha_3,\beta_2,\beta_3)$

Figure 7.4

We also must consider failure of the manager. A hierarchy of managers can be set up so that there is always a backup manager ready to take over in the event of manager failure.

Finally, it is important to note that it is only a failed site which may deliver messages in the wrong order; even in that case, it is only the tail end of its message history that may be incorrect. In addition, there are two big advantages with this method. One is that during failure-free operation, no additional messages have to be sent (e.g., no two-phase commit). Although there is some extra bookkeeping as messages are propagated, the performance of the reliable and unreliable versions during normal operation is roughly the same. The second is that message delivery can continue after the recovery, even though the site is still down.

7.1.2 Atomic Delivery

If rolling back messages is not satisfactory and atomicity is desired, then sites that detect a failure can simply block on messages destined for groups that include the failed site. Blocking is not necessary in all cases. For example, if a leaf site of the propagation tree fails, the other sites in its group(s) can continue, assuming the failed site can get its missed messages upon recovery. If a non-leaf site fails, however, the sites to which it propagates messages must block on the messages they ordinarily receive from that node. When the failed site recovers, it can continue forwarding messages from where it left off.

7.1.3 Reliability of other solutions

Both the algorithms of [BJ87] and [CM84] address the reliability issue. In the original algorithm for the centralized solution (not the simplified version discussed here), fault tolerance is achieved by committing the message ordering via token passing. When this is taken into account, the delay as measured in Chapter 4 increases considerably as noted by Ken Birman [Birm88]. The reliability alternatives as presented here can be applied to the simplified centralized solution and thus face the same tradeoff of rollback vs. blocking.

The reliability of the algorithm in [BJ87] is an inherent part of the two-phase nature of the protocol and suffers the same problem of blocking as does two-phase commit. In fact, it can block under the same conditions as the propagation method and the centralized solution (e.g., the source fails before it can send the second phase messages). A three-phase protocol may be adequate to prevent blocking, but this is even less efficient. Thus, the blocking propagation method (the second reliability solution we described) provides the same reliability as the two-phase protocol. Intuitively, it may seem that having a second broadcast phase is necessary for atomic delivery. However, since sites never can refuse to process messages, the propagation graph approach achieves atomic delivery in a single phase by making centralized ordering decisions (enforced via sequence numbers) and blocking sites when failures occur.

7.2 Reliability - Formally

We now consider the reliability alternatives of the PG algorithm formally, using the model of Chapter 6. We present two versions, the first provides atomic ordered delivery by providing guarantee R1. The second version provides R2, non-atomic ordered delivery. In order to present the protocols formally and clearly, we have made some simplifications. The algorithms presented here provide multiple source ordering, not multiple group ordering. We do not consider the deletion and insertion of group members for any reason other than failure or recovery.

7.2.1 Protocol 4: Propagation Graph Algorithm Version 1

In this first version of the propagation graph algorithm, we provide a strict R property: R1. It is "strict" because it does not ever allow two messages to be delivered in contrary orders at two different sites, therefore it adheres to the atomic ordered delivery alternative. For the PG algorithm, this requires that all message propagation stop until the failed node recovers (unless the node is a leaf in the tree).

Properties P4, R1, and Q3 are guaranteed. This protocol tolerates site and communication failures. We use assumption [A3] and make the following other assumptions:

[A4] The initial logical tree is known by all sites, in particular all sites know who their parent is and who their children are.

[A5] The value of n for a message $[S,D,n]$ is a concatenation of the source id, S, and a sequence number assigned by S. Thus, n is $S.x$, where x is determined by a local counter. (Further, $n+1 = S.x+1$ if $n = S.x$.) Including S in the sequence number is redundant, but this assumption is needed because the model stipulates that n uniquely identify the message.

Storage Requirements: Several sequence numbers are required at tree sites for messages. All tree sites except the root maintain a variable, L_p, for ensuring that messages from the parent are not missed. Each parent (including the root) maintains a variable L_c to tag sequentially messages being sent from the parent to its children. In addition, each tree site maintains a local array *children*(D), to store which sites in group D are children of that site. The root maintains a variable L_s^i for sequence numbers of messages from source i (assigned according to [A5]), for each source i. At all sites, messages must be kept in a LOG for recovery purposes. A queue, Q, for undelivered messages is also used. Both the LOG and Q must be on stable storage.

Sources must maintain a sequence number L_r, to tag messages to the root. L_r is initialized to 0. In addition sources must record the messages they send on a LOG in stable storage. Each source maintains the variable *root*(D), the id of the root of the tree for group D.

Tree protocol

(A) At source s, to multicast a message ($[s, D, n]$), the following is done:

```
$n \leftarrow L_r$;
$L_r \leftarrow L_r + 1$;
LOG($[s, D, n]$;
send $[s, D, n]$ to $root(D)$;
$\langle S([s, D, n])\rangle_s^{root(D)}$ occurs
```

(B) At $root$, upon receiving $[S, D, n]$:

```
$\langle G(m)\rangle_{root}$ occurs
if $n = L_s^s + 1$ then begin
    $\langle D([S, D, n]\rangle_{root}$ occurs
    $L_s^s \leftarrow L_s^s + 1$;
    $L_c \leftarrow L_c + 1$;
    LOG($[S, D, n]$,$L_c$);
    deliver message $[S, D, n]$;
    for each $g$ in $children(D)$ do begin
        send ($[S, D, n]$,$L_c$) to $g$;
        $S([S, D, n])\rangle_{root}^{g}$ occurs end
    while message $[S, D, L_s^s+1]$ is in $Q$ do begin
        remove message $[S, D, L_s^s+1]$ from Q;
        $L_s^s \leftarrow L_s^s + 1$;
        $L_c \leftarrow L_c + 1$;
        LOG($[S, D, L_s^s]$,$L_c$);
        deliver message $[S, D, L_s^s]$;
        $\langle D([S, D, L_S^S]\rangle_{root}$ occurs
        for each $g$ in $children(D)$ do begin
            send ($[S, D, L_s^s]$,$L_c$) to $g$;
            $S([S, D, n])\rangle_{root}^{g}$ occurs   end end
else if $n > L_s^s+1$ then begin
    add $[S, D, n]$ to Q;
    for all messages $[S, D, x]$ such that $[S, D, x]$ not in Q and $L_s^s<x<n$ do
        request retransmission of $[S, D, x]$ from $S$; end
else ignore message;
```

(C) At non-root group member, a, upon receiving message $([S, D, n], p)$ from site b, parent of a:

```
$\mathbf{G\langle[S, D, n]\rangle_a}$ **occurs**
if $p = L_p + 1$ then begin
    $L_p \leftarrow L_p + 1$;
    $L_c \leftarrow L_c + 1$;
    LOG($[S, D, n], L_c$);
    deliver message $([S, D, n], p)$;
    $\mathbf{\langle D([S, D, n]\rangle_a}$ **occurs**
    for each $g$ in $children(D)$ do begin
        send $([S, D, n], L_c)$ to $g$;
        $\mathbf{\langle S([S, D, n])\rangle_g}$ **occurs**   end
    while message $([S, D, x], L_p + 1)$ is in Q do begin
        $L_p \leftarrow L_p + 1$;
        remove message $([S, D, x], L_p)$ from Q;
        $L_c \leftarrow L_c + 1$;
        LOG($[S, D, x], L_c$);
        deliver message $([S, D, x])$;
        $\mathbf{\langle D([S, D, x])\rangle_a}$ **occurs**
        for each $g$ in $children(D)$ do begin
            send $([S, D, x], L_c)$ to $g$;
            $\mathbf{\langle S([S, D, n]\rangle_g}$ **occurs**   end end

else if $p > L_p + 1$ then begin
    add $([S, D, n], p)$ to Q;
    for all messages $([S, D, x], y)$ such that $([S, D, x], y)$ not in Q and $L_p < y < m$ do
        request retransmission of $([S, D, x], y)$ from $b$; end
else ignore message; end
```

(D) After a node a awakens from a failure:

```
$\mathbf{\langle A^1\rangle_a}$ **occurs**
wait for next message $([S, D, n], y)$ to arrive;
$\mathbf{\langle R^1\rangle_a}$ **occurs**
process $([S, D, n], y)$ and following messages using (B);
```

```
(E) At each site with children, if no messages have been sent for $\delta$ seconds:

    $L_c \leftarrow L_c + 1$;
    for each child do
        send null message to child: ([$null$],$L_c$);

(F) At each source, if no message is multicast for $\delta$ seconds:

    multicast null message using (A);

(G) At a site $b$, when a retransmission request is received from a site $a$:

    resend requested messages to $a$;
```

Proof of Correctness

It is not difficult to see that the PG algorithm guarantees R1. All sites deliver messages in strict sequence number order. This numbering is determined by the root. The message order is maintained by the parent/child sequence numbers.

It would be nice to say that Protocol 4 guarantees the strong P property, P2. It does not, though, because Protocol 4 propagates messages from the source via a series of sites. Thus, it is not possible to guarantee delivery based on a destination's ability to reach the source when it should only receive the message via its parent. Instead, P4 is used, which guarantees delivery at a site x that depends on the availability of the site responsible for sending the message to x.

The algorithm as described guarantees P4, where $\Delta = \delta + 3\phi_P + 3\phi_N$. Say that $\tau = T\langle S(m)\rangle_s^a$ where $m = [s, D, n]$. For convenience, assume that s is the source of m. That is, in P4, s assumes the role of x. Site a is the root of the tree. We will then address the case of s being a non-source. There are two cases: $\theta = \tau$ and $\theta > \tau$.

If $\theta = \tau$, then a receives m within ϕ_N of τ. If it cannot deliver m immediately, it requests any missed messages within ϕ_P (code fragment (C)). $2\phi_N + \phi_P$ is the maximum time expired before a receives the missed messages. Another ϕ_P time is needed to process them, as well as deliver m. Thus, P4 holds, since $3\phi_N + 3\phi_P < \Delta$.

Else $\theta > \tau$. Within $\delta + \phi_P$ of θ, s sends some message, $m'=[s,D,n']$ to a, even if just a null message, due to (F). Note $n' > n$. Site a receives it within ϕ_N, processes it within ϕ_P, requesting any missed messages, including m. Another ϕ_N is required for the request to reach s, and $\phi_P + \phi_N + \phi_P$ is required for a to finally receive and deliver m, as in the first case. Thus, $\Delta = \delta + 3\phi_P + 3\phi_N$. The case of s not being the source is equivalent, except that code fragment (E) is responsible for the transmission of null messages.

The proof that the protocol guarantees Q3 is almost identical to that of P4, so we do not go into detail. The value of ρ is also $\delta + 3\phi_P + 3\phi_N$.

Although P4 and Q3 seem similar, both are needed to make the protocol correct. If the protocol guaranteed just P4, then delivery of a message m only at the root would be sufficient for correctness. If just Q3 were guaranteed, then delivery at no group members would be correct, since delivery is only guaranteed within ρ if m has been delivered at some site in the group, in particular, at a parent site. However, the root has no parent.

The P and Q properties this algorithm provides are necessarily very weak. This is because the logical tree structure is static. If a node other than a leaf fails, no action is taken to bypass that node and send messages to its children. Hence, since such a node stays down for what is likely an indeterminate amount of time, a guarantee cannot be made concerning any pair of sites a and b. Instead, a bound can only be provided between parent and child pairs in the tree, and from source to root.

7.2.2 Protocol 5: Propagation Graph Algorithm Version 2

In this next version of the propagation graph algorithm the R property is weakened in order to provide stronger P and Q properties. Instead of shutting down tree operation when an interior node fails, the remaining sites reconfigure the tree and go on with message delivery. Note, though, that if it is the root that has failed, it is possible that the root delivered some messages that did not get propagated to the remaining sites. As part of this protocol, these messages will get redelivered via the new tree. It is possible that this delivery order will not match that of the failed root (non-atomic delivery). This situation yields a weaker ordering property than the previous version of the tree protocol. This second version provides R2. We will show that P4 is again guaranteed. We will also show something stronger: P2. In addition, the Q property provided by this protocol is stronger: Q1. In order to make the code easier to follow, the algorithm differs somewhat from the informal description of Section 7.1.1. The manager (root) does not use the tree to inform sites that they should stop processing and participate in reconfiguring the tree. Instead, the root informs the sites directly.

In order to simplify the formal analysis of the protocol, we make a number of assumptions. At the end of the chapter, we discuss how to relax these assumptions. We use assumptions [A2] (there are no communication failures), [A3], [A4] and [A5] and make the following other assumptions:

[A6] The root is the only site that fails. (For multiple source ordering, this is the interesting case.)

[A7] Failed sites do not recover.

[A8] Once the root fails, at least $\delta+3\phi_P+2\phi_N$ seconds pass before there is a failure of the new root, where δ is a constant of the algorithm and is used for timeouts. (This provides enough time for the new tree to be in place and for operation to be proceeding normally.)

Storage Requirements: For this protocol, each parent/children set does not need its own sequence numbering system. Instead, all sites maintain L_D, to keep track of the sequence numbers for all messages that have been sent to the group D. (The root assigns L_D.) As before, the root maintains a variable L_s^i for messages from source i, for each source i. In addition, each tree site maintains the local array *children*(D), as before. Again, stable storage at all sites is needed for the LOG and for the queue, Q. Also, a temporary queue, Q_temp, is needed at the root when it first becomes the root, also in stable storage. Each site is assigned a rank for group D to be used in the event of root failure. The rank is kept in $rank_i(D)$ at site i. In addition, each site maintains the rank of the root in $rank_{root}(D)$. The root maintains a variable called *state* which is initialized to CLOSED when the root is new and is set to OPEN when the new root can proceed normally.

Sources must maintain the same storage as in Protocol 4.

Tree protocol

(A) At source s, to multicast $[s, D, n]$, the following is done:

```
    $n \leftarrow L_r$;
    $L_r \leftarrow L_r + 1$;
    LOG($[s, D, n]$);
    send $[s, D, n]$ to $root(D)$;
    $\langle S([s, D, n]) \rangle_s^{root(D)}$ occurs
```

(B) At *root*, if $state = OPEN$, upon receiving $m = [S, D, n]$ from S:

```
$\langle \mathbf{G(m)} \rangle_{\mathbf{root}}$ occurs
if $n = L_s^s + 1$ then begin
        $L_s^s \leftarrow L_s^s + 1$;
        $L_D \leftarrow L_D + 1$;
        LOG($[S, D, n], L_D$);
        deliver message $[S, D, n]$;
        $\langle \mathbf{D([S, D, n])} \rangle_{\mathbf{root}}$ occurs
        for each $g$ in $children(D)$ do begin
                send ($[S, D, n], L_D$) to $g$;
                $\langle \mathbf{S(m)} \rangle_{\mathbf{root}}^{\mathbf{g}}$ occurs   end
        while message $[S, D, L_s^s+1]$ is in $Q$ do begin
                remove message $[S, D, L_s^s+1]$ from Q;
                $L_s^s \leftarrow L_s^s + 1$;
                $L_D \leftarrow L_D + 1$;
                LOG($[S, D, L_s^s], L_D$);
                deliver message $[S, D, L_s^s+1]$;
                $\langle \mathbf{D([S, D, L_s^s])} \rangle_{\mathbf{root}}$ occurs
                for each $g$ in $children(D)$ do begin
                        send ($[S, D, L_s^s], L_D$) to $g$;
                        $\mathbf{S([S, D, L_s^s])} \rangle_{\mathbf{root}}^{\mathbf{g}}$ occurs   end end
else if $n > L_s^s + 1$ then begin
        add $[S, D, n]$ to Q;
        for all messages $[S, D, x]$ not in Q such that $L_s^s < x < n$ do
                /* for recovery of source messages at new root */
                request retransmission of $[S, D, x]$ from $S$;  end
else ignore message;
```

(C) At root, a, if $state = CLOSED$, upon receiving message $([S, D, n], d)$ (not from S):
/* catching up to siblings */

```
$\langle \mathbf{G}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_a$ **occurs**
if $d = L_D + 1$ then begin
        LOG($[S, D, n], d$);
        deliver message ($[S, D, n]$);
        $\langle \mathbf{D}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_a$ **occurs**
        for each $g$ in $children(D)$ do begin
                send ($[S, D, n], d$) to $g$;
                $\langle \mathbf{S}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_g$ **occurs**   end
        while message ($[S, D, x], d+1$) for any $x$ is in Q_temp do begin
                remove message ($[S, D, x], d+1$) from Q_temp;
                $L_D \leftarrow L_D + 1$;
                LOG($[S, D, x], L_D$);
                deliver message ($[S, D, x]$);
                $\langle \mathbf{D}([\mathbf{S}, \mathbf{D}, \mathbf{x}]) \rangle_a$ **occurs**
                for each $g$ in $children(D)$ do begin
                        send ($[S, D, x], L_D$) to $g$;
                        $\langle \mathbf{S}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_g$ **occurs**  end end
else if $d > L_D + 1$ then begin
        add ($[S, D, n], d$) to Q_temp; end
else ignore message; end
```

(D) At non-root group member, a, upon receiving message $([S, D, n], d)$ from parent:

```
$\langle \mathbf{G}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_{\mathbf{a}}$ occurs
if $d = L_D + 1$ then begin
    $L_D \leftarrow L_D + 1$;
    LOG($[S, D, n], d$);
    deliver message ($[S, D, n]$);
    $\langle \mathbf{D}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_{\mathbf{a}}$ occurs
    for each $g$ in $children(D)$ do begin
        send ($[S, D, n], d$) to $g$;
        $\langle \mathbf{S}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_{g}$ occurs end
    while message ($[S, D, x], y+1$) for any $x$ is in Q do begin
        remove message ($[S, D, x], y+1$) from Q;
        LOG($[S, D, x], y$);
        deliver message ($[S, D, x]$);
        $\langle \mathbf{D}([\mathbf{S}, \mathbf{D}, \mathbf{x}]) \rangle_{\mathbf{a}}$ occurs
        for each $g$ in $children(D)$ do begin
            send ($[S, D, x], y$) to $g$;
            $\langle \mathbf{S}([\mathbf{S}, \mathbf{D}, \mathbf{n}]) \rangle_{g}$ occurs   end end
else if $d > L_D + 1$ then begin
    add ($[S, D, n], d$) to Q; end
else ignore message; end
```

(E) At site i where $rank_i(D) = rank_{root}(D) + 1$, if no messages received from root for δ seconds:

```
/* take over as root */
directly inform all remaining sites of new tree and $L_D$;
for all $j$ such that $j$ is a source do
    $L_D^j \leftarrow$ greatest $x$ such that ($[j, D, x], y$) in LOG, for any $y$;
$state \leftarrow CLOSED$;
clear Q; /* remove any pending messages */
```

(F) At site j, upon receiving new tree and value d from i:

```
$root \leftarrow i$;
$rank_{root}(D) \leftarrow rank_{root}(D) + 1$;
while ($[S, D, x], d + 1$) for any $x$ in LOG do begin
        send ($[S, D, n], d + 1$) to $root$;
        $\langle S([S, D, n]) \rangle_j^{root}$ **occurs**
        $d \leftarrow d + 1$; end
```

(G) At source, if no message is multicast for δ seconds:

```
multicast null message using (A);
```

(H) At root, after $state = CLOSED$ for $2\phi_N + 2\phi_P$ seconds:

```
/* by now the new tree is in place and new root is caught up to other sites */
$state \leftarrow OPEN$;
inform sources of new root;
```

(I) At source, upon hearing from i of new root:

```
$root(D) \leftarrow i$;
```

(J) At source s, upon receiving request for retransmission of $[s, D, x]$ from root:

```
resend $[s, D, x]$ by retrieving it from LOG;
```

(K) At a site b, when a retransmission request is received from a site a:

```
resend requested messages to $a$;
```

Proof of Correctness

It is not difficult to see that this version of the PG algorithm guarantees R2. Essentially, R2 requires that if two messages are delivered in contrary orders at two sites, then it must be the case that the pair was delivered at one site, that site failed, and then the pair was delivered at the other site. This is the only situation that allows inconsistent delivery.

Because of assumption [A2], the only situation that can cause inconsistent delivery in Protocol 5 is if the root delivers messages locally before it fails that are not sent to its children. In that case, the children will recover those messages from the sources and possibly deliver them in a different order than the old root did. But these messages are not recovered by the children until well after the failure event at the root, so R2 holds.

The algorithm as described guarantees P4, where $\Delta = \phi_P + \phi_N$. Say that $\tau = T\langle S(m)\rangle_x^a$ where $m = [s, D, n]$. Note first the types of send events that may occur.

(1) Sources send messages to the root during normal tree operation.

(2) Sources send messages to a failed root.

(3) During normal tree operation, parents send messages in sequence number order to their children.

(4) During tree reconfiguration, some tree sites send messages to the root in sequence number order so that the root can catch up.

Note that it is not possible for sources to send a message to a root that is still in the midst of reconfiguring the tree. Sources are not informed of a new root until the new tree is in place. The tree reconfiguration requires $2\phi_P + 2\phi_N - \phi_N$ for all sites to be informed of new tree, ϕ_P for these sites to execute (F), ϕ_N for the new root to receive all missing messages, and ϕ_P for the new root to process those messages. By assumption [A8], there will not be another site failure within this time, so the new root does indeed catch up to the other sites. By (H), the sources are not informed of the new root until after this catching up process completes.

Messages sent by a source to a failed root do not concern us, as UP(a,τ,Δ) is not true in this case. This leaves (1), (3) and (4). These cases are similar to the scenario in Protocol 1 for the proof of P2. Note that in all three cases, the site x which sends m to a has logged (and delivered, in the non-source case) all previous messages. Thus, once a receives m, if it cannot deliver it due to missing messages, those messages can be retrieved from x, as long as REACH(a,x,θ,Δ), UP(a,θ,Δ) and AWAKE(x,ϕ,Δ). $\phi_P + \phi_N$ time suffices.

The protocol guarantees Q1, with $\rho = \delta + 3\phi_P + 2\phi_N + n{-}1(\phi_P + \phi_N) + \phi_P$, where n is the number of sites in D. The value $n{-}1(2\phi_P + \phi_N)$ is merely the cost of propagating a message from the root to a leaf if the tree is linear. The longest delay between deliveries at two group members occurs between the root and the site of greatest depth. If there are n sites in the group, $n{-}1$ is the maximum this depth could be. For the root to deliver m and propagate it requires ϕ_P time. At each intermediate hop, $\phi_P + \phi_N$ is needed to get the message to the next node and processed. It is possible, though, that the root will deliver m and fail before propagating it to all its children. The rest of ρ covers the case of failure of the root and is determined by the time it takes for a new root to recover a message sent by the old root to some other child. δ time is needed for the new root to detect failure of the old root. Execution of (E) requires ϕ_P seconds, ϕ_N is needed to get the new tree information to the other sites. They require ϕ_P to execute (F). ϕ_N is needed to send any missed messages. ϕ_P is needed for the new root to process those messages, deliver them and propagate them. (By [A8], we do not need to consider root failure in the midst of tree reconfiguration.) The propagation requires $n{-}1(\phi_P + \phi_N) + \phi_P$, as shown above.

By proving that Protocol 5 guarantees P4, Q1 and R2, we have shown that all UP sites deliver messages in the same order, given some conditions about the system.

However, we have not shown that every message from a source gets delivered at the destination group. We have only shown there is some atomicity of delivery at UP sites. It turns out that Protocol 5 guarantees a P property stronger than P4. Not only does it guarantee that messages sent from particular sites to particular sites are delivered if UP and REACH hold, it also guarantees that every message from a source gets delivered by the operational group members. This is property P2. It is not too difficult to see this is the case. If a source initiates a multicast by sending it to a failed root, its messages will be retrieved by the next root that stays operational long enough. In the worst case, the last site to get such a message is the last site to become the root after a string of successive failures of new roots. Each reconfiguration requires $\delta + 3\phi_P + 2\phi_N$ as shown above. Another $2\phi_P + 2\phi_N$ is needed for the source to learn of the new root. Within δ of that time, the source must send some message to the new root (code fragment (G)). When that message is processed at the root using (A), old missed messages are retrieved. This requires $3\phi_N + 3\phi_P$. Thus, the value of Δ is $n-1(2\delta + 7\phi_N + 8\phi_P)$.

7.3 Relaxing the Assumptions

The formal proofs of Section 7.2 give us more confidence in the correctness of the PG protocol. They also, however, point out its limitations. The P and Q properties guaranteed by Protocols 4 and 5 rely on certain properties of the computing environment. In particular, UP and REACH requirements are stringent. The UP property insists that a site be able to process code fragments in a certain amount of time and REACH requires conditions for communication between pairs of sites. Without these properties, P and Q guarantees cannot be made. This is not unreasonable behavior, though. We cannot expect a system with unreliable sites and communications links to deliver messages reliably. We do need to consider, however, the behavior of the protocol when we relax our assumptions and allow all kinds of failures.

There are several issues concerning the protocols for the PG algorithm presented in Section 7.2 that must be addressed:

(1) How do we handle multiple failures that are not conveniently staggered in time?

(2) How do we incorporate multiple group ordering?

(3) What happens when there are communication failures and/or the fail-stop model is not applicable?

(4) What happens if sources fail?

(5) How do we incorporate new or recovering sites into the tree?

We address these questions informally, applied to Version 2 of Section 7.2.2, since this is the more complex protocol.

Tolerating multiple, closely-spaced failures does not require much change to the protocol. For now, we will continue to assume there are no communication failures. Consider first a non-root failure in the midst of tree reorganization due to an earlier non-root failure. Since roots inform the other tree sites directly of the new tree, the failure does not prevent operational sites from hearing of the new tree. Assuming the newly-failed site has children in the new tree, these children will eventually detect the failure and request the tree be reorganized again. If the root fails in the midst of reorganizing the tree for a non-root failure, then this is eventually detected by the site responsible for taking over as root. Reorganization is restarted. If consecutive roots fail, then a problem ensues because a decision must be made as to which site becomes the next root. A one-time prioritizing of the nodes can be used for this (e.g., the level-order of the original tree), but sites must know when it is time to take over. A timeout takes care of this, with the length of the timeout set proportional to the position of the site's spot in the order.

The incorporation of multiple group ordering complicates the protocol since not all messages enter the tree at the root. Non-failure operation is essentially the same as for multiple source ordering. Reorganizing the tree due to failure is more complicated. As for multiple source ordering, children detect failures of parents and inform the root. The root closes the tree (queuing new messages), decides on a new tree and informs all the sites directly. The operational sites that are primary destinations also close and queue new messages from sources. All sites send the root their message histories dated back far enough to include the last message delivered per group the site is in. The root must reconcile these histories to create a total order of message delivery. Each site must be informed of its missing messages so that all sites can be brought up-to-date for the start of the new tree. (This was described in Section 7.1.) It is less reasonable now to expect that reorganization will attach an "orphaned" site to some ancestor which can provide missed messages. Missed messages, then, must be obtained from other sites, just like the new root must do in the case of multiple source ordering. Sources are informed of the new root, as in Protocol 5.

This algorithm requires no more phases than did Protocol 5. However, local processing is more complex, especially at the root. Also, since missed messages do not get propagated down the tree, more messages may be sent in this version. (A more sophisticated algorithm might make message recovery via the new tree possible, but we do not consider this here so as not to complicate the description.) Note that multiple failures during reorganization can be handled as above.

It is not reasonable in practice to expect [A2] to hold. So, we must address how to handle communication failures in the protocol. By the same token, our use of the definition of UP makes failures fail-stop, so that slow sites are down sites. In practice a site that is functioning slowly may not just halt; it may just continue at its slow pace. Taken together, fail-stop processing and reliable communication between pairs of sites gives us the ability to detect failures accurately. Though in practice this is impossible, most protocols do rely on some notion of an "operational" site. One way to decide which site is operational in practice is to use the concept of a *logical failure* , as described in [BJ87]. Such a failure occurs when enough sites in the system (e.g., a majority) deem another site to have failed. The site, for instance, may be so overloaded that no other sites have heard from it in a while. Even though it may not be completely down, the system acts as though it is. Similarly, the site may have lost its communication link with the other sites (in the case of a partition). When the "down" site again establishes communication with the others, it is told that it is dead. Such an approach can be used in the propagation graph algorithm. When a site is deemed down, tree reorganization can proceed as described in the protocols. However, we now insist that in order to form a new tree, a majority of sites must participate. This will prevent two new trees from forming at once. Of course, without a majority of sites, no tree will form; this is similar to the situation in the ISIS system, where protocols block if a majority becomes unavailable.

There is an anomalous case that this "majority tree" can lead to. In the event of false failure detection (in particular, due to a partition) it is possible that two trees will exist at once. This can happen if a subtree with a majority of sites breaks off and forms a new tree, while the intact remainder of the old tree continues the propagation of messages. This situation is illustrated in Figure 7.5. This is a highly unlikely situation, but one that should be considered. Nothing can be done to prevent this from happening, but it is reasonable to insist that every tree periodically check to ensure it still contains a majority of members. Thus, the remains of the old tree eventually determine they should gracefully die. In the meantime, they are "logically" failed sites, as the new tree is treating them that way.

As a result, there is a period of time during which different sites may be delivering messages inconsistently, though this period can be made very short and includes sites

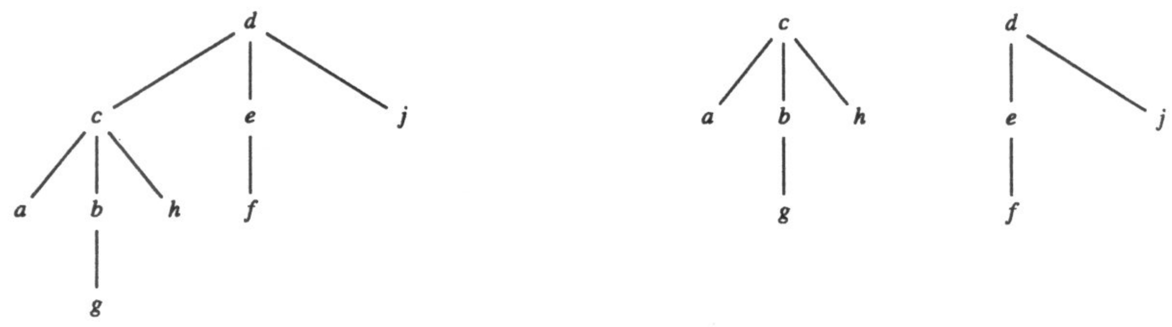


Figure 7.5

that will expire soon, leaving no inconsistencies among up sites. Other protocols work in this fashion. In particular, the two-phase protocol of [BJ87] yields a similar situation if the source and a destination become unreachable in the midst of running the protocol for a message, m. If the source has finalized the priority number of the message and the unreachable destination has delivered it based on that number and the remaining destinations do not hear of the final priority number of m, then inconsistency will result when the remaining sites pick a new priority number for m. (This is the scenario that causes the well-known two-phase commit protocol to block.) Instead of blocking, the remaining sites determine a new order for the messages and go on with delivering more. For some period of time, there are inconsistent delivery orders. Of course, if this is intolerable to the application, then both protocols can simply use blocking to prevent inconsistent delivery, as atomic ordering reliability guarantees.

It is not difficult to handle source failure in the PG protocols. The only complication arises if the primary destination is expected to maintain the order of messages from a source. It may, then, receive messages out of order and must queue later messages to recover earlier messages. But, the source may fail in the meantime. If the source it not expected to recover, the later messages must be discarded. Else, they are held up until earlier messages are recovered.

It has been convenient (and, for many applications, useful) not to consider recovery of failed sites. Similarly, we have not considered entry of new sites into multicast groups. As in [BJ87], we can consider recovering sites to be new sites, thus the cases are the same. To incorporate new sites, the tree must be restructured as in the site failure case. There is little difference from the technique used for failures.

If we want recovering sites to deliver the messages missed while down, then we might employ R3, where sites are allowed to re-deliver messages. Remember that the tail end of the recovering site's message history may be inconsistent with the other sites. With R3, the message history of the recovered site would show it backing out its old deliveries that may conflict with the order that the operational sites had decided upon. Then, these messages get redelivered in the new order. Plus, the messages completely missed by the recovering site are delivered.

Chapter Eight

Conclusions and Future Work

Here we have considered the problems of ordered and reliable multicast delivery both informally and formally. As mentioned in the Introduction, multicasting has become a popular model for message passing in distributed systems. In addition, many researchers claim that ordering multidestination messages consistently at the receivers simplifies many distributed applications. The need for reliable message delivery is well-documented.

The most general form of multicast ordering we have considered here is multiple group ordering. The propagation graph algorithm we have detailed provides this property (as well as single source and multiple source ordering) efficiently. During normal operation it is essentially a "one-phase" algorithm. That is, once a site receives a message it can deliver it without further contact with other destinations of the message. Performance results indicate that the propagation graph algorithm strikes a good compromise between minimizing delay and distributing load and is flexible. It does not require as many messages as other solutions and avoids bottlenecks. Many heuristics can be used for generating the graphs, including heuritics that take into account the network topology.

However, given the cost of setting up the propagation graph, it is only appropriate when the multicast groups are expected to last a long time.

Providing reliability along with ordering properties is not straightforward. (In fact, even without ordering, reliable message delivery is a formidable task.) In order to be truthful about the reliability of a protocol, we claim that precise models of the system and message delivery algorithm are required. In Chapter 6, we presented such a model. We then applied it to the propagation graph algorithm, albeit under favorable system conditions. Regardless, the model illustrates that it is really only under such favorable conditions that a guarantee on delivery can be made. What our model does is clear up the notion of "eventual" delivery at "operational" sites by showing that delivery happens not "eventually," but when the conditions are right. For instance, delivery can occur when the necessary links are up and the necessary sites are available.

Unfortunately, it is very difficult to prove formally anything about a protocol such as the propagation graph algorithm when all possible system events are considered. Instead, we have restricted the system to the types of events we can handle for the formal analysis. However, based on that analysis, we can make reasonable conclusions when we relax the assumptions, as in Chapter 7.

The work presented here has many future directions. We did not have the opportunity to implement our algorithm. Much can be learned from putting an idea into practice and we would benefit from such a task. Enhancements to the algorithm are possible in several areas. Further inquiry into the issue of extra nodes might lead to interesting observations. More interesting, though, would be to study rigorously how to tailor the propagation graph to the topology of the network. This is applicable to both point-to-point and internetwork topologies. Careful positioning of the nodes in the forest could lead to a very efficient graph. Work has been done in this area to provide for reliable delivery when ordering is not needed [McKL90]. Techniques used there might be applicable to the ordering case.

The model developed in Chapter 6 is really only a start in what is a very complex problem. By formally considering other types of algorithms (e.g., those that use a phased commit protocol or those that use tokens to commit) many other P and Q properties could be developed. There are many simple algorithms which lend themselves to formal analysis by the techniques in Chapter 6. Lamport's timestamp algorithm [Lamp78] and the Fault-Tolerant Broadcast algorithm of Schneider, Gries and Schlichting [SGS84] are good candidates. Such an analysis could give us insight into the efficiency of these protocols under favorable conditions and also give us a new way to view their correctness. Comparison of various algorithms is much easier when they are considered using a common model.

APPENDIX I

PG Generator Pseudo-code

The Propagation Graph (PG) Generator

```
main()
begin

groups ← the set of multicast groups;
sites ← the set of sites;
unmarked_groups ← groups;
unmarked_sites ← sites;


while unmarked_groups <> ∅
{
   root ← s | s occurs most frequently in unmarked_groups;
   new_subtree(root);
}

end
```

```
new_subtree(current_subroot)

begin

intersecters ← ∅;

/* Mark site since it has been placed in forest. */
mark_site(current_subroot);

/* Determine the sites that are in groups with the subroot. */
for each $s \in$ unmarked_sites
    if $\exists g \in$ unmarked_groups such that ($s \in g \wedge$ current_subroot $\in g$)
    then
        intersecters ← intersecters $\cup$ s;

/* Mark all groups that contain subroot since now we have a primary destination
for them. */
for each $g \in$ unmarked_groups
    if current_subroot $\in g$
    then
        mark_group(g);

/* Partition groups so that no group in a partition intersects a group in another partition
and some site in some group of each partition is included in a group with the subroot (is in
intersecters). */
$G \leftarrow \{g \mid g \in$ unmarked_groups $\wedge \exists s \in g$ such that $s \in$ intersecters$\}$;
repeat
    $S \leftarrow \{s \mid \exists g \in G$ such that $s \in g\}$
    $G \leftarrow G \cup \{g \mid g \in$ unmarked_groups $\wedge \exists s \in G$ such that $s \in S\}$
until no change to G
$P_1 \cdots P_k \leftarrow$ partition of G so that no group in a partition intersects a group
                in another partition;

/* If s is in a group with the root but is not in a partition, make it a child. */
for each $s \in$ intersecters
    if s is not in a $P_i$
        current_subroot → s;  /* make s a child of current_subroot */

/* Determine a child from each partition. */
for i := 1 to k
{
    newsite ← s | s occurs most frequently in $P_i \wedge s \in$ intersecters;
    current_subroot → newsite;  /* make newsite a child of current_site */
    new_subtree( newsite );
end
```

```
mark_site(s)
begin

unmarked_sites ← unmarked_sites - s;

end;

mark_group(g)
begin

unmarked_groups ← unmarked_groups - g;

end;
```

BIBLIOGRAPHY

[Ahuj89] M. Ahuja, "Implementation and Use of *Flush* Primitives for Asynchronous Concurrent Systems," Technical Report OSU-CISRC-9/89 TR41, Ohio State University.

[Birm88] *Personal Communication.*

[BJ87] K. P. Birman, T. A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February 1987, pp. 47-76.

[CD85] D. R. Cheriton, S. E. Deering, "Host Groups: A Multicast Extension for Datagram Internetworks," *Proceedings of the 9th Data Communications Symposium, ACM SIGCOMM Computer Communications Review*, Vol. 15, No. 4, September 1985, pp. 172-179.

[CM84] J. Chang, N. F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 251-273.

[CZ85] D. R. Cheriton, W. Zwaenepoel, "Distributed Process Groups in the V Kernel," *ACM Transactions on Computer Systems*, Vol. 3, No. 2, May 1985, pp. 77-107.

[DC90] S. E. Deering, D. R. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Transactions on Computer Systems*, Vol 8, No. 2, May 1990, pp. 85-110.

[FLP85] M. J. Fischer, N. A. Lynch, M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 34, No. 2, April 1985, pp. 374-382.

[FWB85] A. J. Frank, L. D. Wittie, A. J. Bernstein, "Multicast Communication on Network Computers," *IEEE Software*, Vol. 2, No. 3, May 1985, pp. 49-61.

[Garc82] H. Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Transactions on Computers*, C-31, No. 1, January 1982, pp. 48-59.

[Gray78] J. N. Gray, "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, G. Goos, J. Hartmanis, ed. Springer-Verlag, New York, 1978, pp. 393-481.

[Gray88] J. Gray, "The Cost of Messages," *Proceedings of the Seventh Annual Symposium on Principles of Distributed Computing, August 1988, pp. 1-7.*

[GA87] J. J. Gray, M. Anderton, "Distributed Computer Systems," *Proceedings of the IEEE*, Special Issue on Distributed Database Systems, Vol. 75, No. 5, May 1987, pp. 719-726.

[GKL88] H. Garcia-Molina, B. Kogan, N. Lynch, "Reliable Broadcast in Networks with Nonprogrammable Servers," *Proceedings of the Eighth International Conference on Distributed Computing Systems*, June 1988.

[KG87] B. Kogan, H. Garcia-Molina, "Update Propagation in Bakunin Data Networks," *Proceedings Sixth ACM Symposium on Principles of Distributed Computing*, August 1987, pp. 13-26.

[KTHB89] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummell, H. E. Bal, "An Efficient Reliable Broadcast Protocol," *Operating Systems Review*, Vol. 23, October 1989, pp. 5-19.

[Lamp78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp. 558-565.

[LSP82] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.

[LG88] S. Luan, V. D. Gligor, "A Fault-Tolerant Protocol for Atomic Broadcast," *Proceedings Seventh Symposium on Reliable Distributed Systems*, October 1988, pp. 112-126.

[LG90] S. Luan, V. D. Gligor, "A Fault-Tolerant Protocol for Atomic Broadcast," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 271-285.

[McKL90] P. K. McKinley, J. W. S. Liu, "Multicast Tree Construction in Bus-Based Networks," *Communications of the ACM*, Vol. 33, No. 1, January 1990, pp. 29-42.

[MMA90] P. M. Melliar-Smith, L. E. Moser, V. Agrawala, "Broadcast Protocols for Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*," Vol. 1, No. 1, January 1990, pp 17-25.

[MMA90b] L. E. Moser, P. M. Melliar-Smith, V. Agrawala, "On the Impossibility of Broadcast Agreement Protocols," to be published.

[NCN88] S. Navaratnam, S. Chanson, G. Neufeld, "Reliable Group Communication in Distributed Systems," *Proceedings Eighth International Conference on Distributed Computing Systems*, June 1988, pp. 439-446.

[Schn82] F. B. Schneider, "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, April 1982, pp. 125-148.

[Schr90] M. D. Schroeder, et. al., "Autonet: a High-speed, Self-configuring Local Area Network Using Point-to-point Links," Technical Report 59, DEC Systems Research Center, Palo Alto, April 1990.

[SGS84] F. B. Schneider, D. Gries, R. D. Schlichting, "Fault-Tolerant Broadcasts," *Science of Computer Programming*, 4(1984), pp. 1-15.

[SS83] R. D. Schlichting, F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computing Systems*, Vol. 1, No. 3, August 1983, pp. 222-238.

[Tarj83] R. E. Tarjan, "Data Structures and Network Algorithms," Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Wuu85] T. Wuu, "Reaching Consistency in Unreliable Distributed Systems", Ph.D. thesis, Department of Computer Science, State University of New York at Stony Brook, August 1985.