ORDERED AND RELIABLE MULTICAST COMMUNICATION

Hector Garcia-Molina
Annemarie Spauster

October 1988

CS-TR-184-88

Ordered and Reliable Multicast Communication

Hector Garcia-Molina
Annemarie Spauster

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

A multicast group is a collection of processes that are the destinations of the same sequence of messages. These messages may originate at one or more source sites and the destination processes may run on one or more sites, not necessarily distinct. A multicast protocol is responsible for the delivery of messages to the appropriate processes. Some applications require that the protocol provide some guarantees on the order in which messages are delivered. In addition, in many cases reliability of delivery is essential. In this paper we characterize three ordering properties and discuss their solutions. We concentrate on the *multiple group ordering* property, which guarantees that two messages destined to two processes are delivered in the same relative order, even if they originate at different sources and are addressed to different multicast groups. We present a new protocol that solves the multiple group ordering problem. We present extensive experimental results that illustrate the performance of our algorithm compared to other techniques for ordering multicasts. We address the issue of reliability by providing several alternatives for handling message loss and site crash recovery. We explore the usefulness of these with a comparison to the reliability guarantees of the other proposed solutions. In many cases our new algorithm solves the problem with greater efficiency than previous solutions without sacrificing reliability.

Ordered and Reliable Multicast Communication

Hector Garcia-Molina
Annemarie Spauster

Department of Computer Science
Princeton University
Princeton, NJ 08544

1. THE PROBLEM

A multicast group is a collection of processes that are the destinations of the same sequence of messages. These messages may originate at one or more source sites and the destination processes may run on one or more sites, not necessarily distinct. Each source message is addressed to the multicast group (as opposed to individual sites or processes). The multicast protocol ensures that the messages are delivered to the appropriate processes. Much research has been done in the area of multicasting. Cheriton and Deering [CD85] have studied the use of multicasting in internetworks and claim that an efficient multicast facility is needed because broadcast (transmission of a message to *all* sites) is not a generally useful facility as there is little reason to communicate with all sites. Gray [Gray88] concurs, saying that the scalability of multicast to very large networks appeals to practitioners.

For some applications, the multicast protocol must provide guarantees regarding the order in which messages are delivered to the destination processes. The properties are usually the following ones, arranged by increasing strength.

(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 all destination processes get them in the same relative order.

(b) *Multiple source ordering.* If messages m_1 and m_2 are addressed to the same multicast group, then all destination processes get them in the same relative order (even if

This work has been supported by NSF Grants DMC-8351616 and DMC-8505194, New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from DEC, IBM, NCR, and Concurrent Computer corporations.

they come from different sources).

(c) *Multiple group ordering*. If messages m_1 and m_2 are delivered to two processes, they are delivered in the same relative order (even if they come from different sources and are addressed to different but overlapping multicast groups).

There are of course applications that do not require all of these (or even any of these) properties. But there are applications where the receipt of messages in differing orders will lead to inconsistency or deadlock problems. To illustrate, consider a bank 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.) The two main computers constitute a multicast group, and each branch office is a potential source site. Transactions should be executed in the same order (property b) 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 our same banking example, consider now a second multicast group 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 separate multicast groups are involved, it is probably still important to process all messages in the same order at the machines in the intersection of the groups (property c).

This example illustrates why the ordering properties for multicasts are important. Birman and Joseph [BJ87] 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]).

It is with this motivation that we present a novel multicast message ordering technique. Our solution guarantees all three of the ordering properties listed above. Of course, in many cases, the multicast protocol must exhibit some *reliability properties*, as well. We view the reliability issue as orthogonal to the ordering issue; we believe that any one of the ordering mechanisms we study here can be made reliable to varying degrees by applying different message loss and site crash recovery techniques. Therefore, we study the two issues separately. In Section 3 we describe our algorithm for providing consistent message ordering. In Section 5 we discuss two reliability alternatives that mesh well with our strategy.

2. EXISTING SOLUTIONS

We begin our study by considering previous solutions that guarantee some or all of the ordering properties presented in Section 1.

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.

Enforcing the multiple source and group properties (b, c) is harder. One solution is to assign a timestamp to each message at the source and then deliver messages in timestamp order [Lamp78, Schn82]. To illustrate, consider the scenario of Figure 1. Sites x and y are sending to multicast group $\alpha = \{a,b,c,d\}$, while site z is sending to $\beta = \{c,d,e,f\}$. We use a, b, etc. to refer to both the destination process and the site where it resides. Suppose that x sends to α message m_1 with timestamp T_1. When site c receives m_1 it cannot immediately give it to its destination process. It first must find out from all

potential sources if there are other messages with smaller timestamps. Only when a site is certain that a message has the smallest timestamp of any undelivered message does it deliver it. For property (b), site *c* must check with all α source sites (e.g., *y*). For property (c), site *c* must in addition check with potential β sources. If the potential sources are unknown, *c* must check with *all* sites in the system.

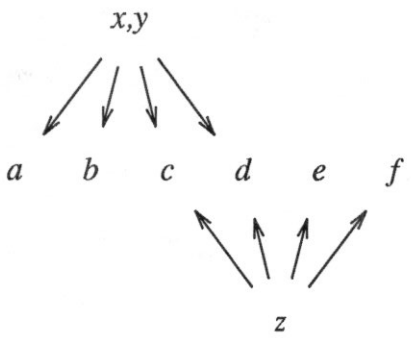


Figure 1

Birman and Joseph have proposed another interesting solution [BJ87] (attributed in part to Dale Skeen) that is similar to two-phase commit. Each site maintains a priority queue per process. The sender sends the message to the multicast destinations who each give it their own priority number, a systemwide unique number higher than any given so far for that process. The message is marked "undeliverable" and put on the queue. Each receiver returns the priority number to the sender. The sender picks the highest priority number it got 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. Whenever a message at the front of the queue is "deliverable", it is delivered. Note that this algorithm guarantees all three ordering properties without requiring receivers to contact all potential sources.

The two approaches we have sketched are fully distributed and may have substantial message overheads. A more centralized approach is suggested in [CM84] to reduce

the synchronization cost. Here all sources transmit to a central site, which assigns sequence numbers to the messages and forwards them to the destination sites. (The central site is identified with a token, and this token circulates through the system. However, from the point of view of ordering, the fact that the central site moves over time is not important.) The paper [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.

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] 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.

3. OUR SOLUTION

In this paper we propose a new solution for guaranteeing property (c) in a multicast environment, called the *propagation algorithm.* Our algorithm is inspired by [CM84] and also attempts to reduce some of the overhead of fully distributed solutions. However, instead of ordering all messages at a single central site, they are ordered by a collection of nodes structured into a *message propagation graph* (in particular, a forest). Each node in the graph represents a computer site. 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, the messages get propagated via a series of sites that order them along the way by merging messages destined for different groups. Eventually, all messages end up at their destinations, already ordered. The key idea is to use sites that are in the intersections of multicast groups as the intermediary nodes.

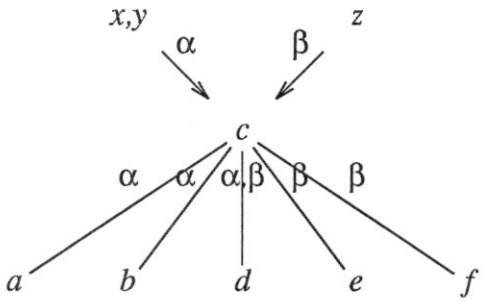


Figure 2

Using the example of Figure 1, a simple propagation graph is indicated in Figure 2. In order to guarantee that all messages are delivered in the same relative order, *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 algorithm has two components: the propagation graph (PG) generator and the message passing (MP) protocol. The PG generator builds the propagation graph for a given set of multicast groups. For simplicity, we assume one site runs the PG generator and transmits the resulting graph to the other sites. Dynamic changes to the set of multicast groups are discussed in Section 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. We call the site that originates a message for a multicast group the *source* and the group that is to receive that message the *destination group*. The source sends the multicast message to one site in the multicast group, called the *primary destination*. (The primary destination could be the source.) Any time a site sends a message to another site, we refer to these sites as the sender and receiver, respectively.

One important requirement for the algorithm is that property (a) of Section 1 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 2. 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 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.

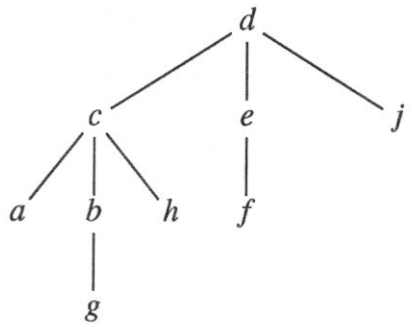


Figure 3

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

3.1 The PG Generator

Our technique must guarantee the following two properties:

(1) Property (c) of Section 1, 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.

Both of these properties 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 4.

To start, 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. 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 4.

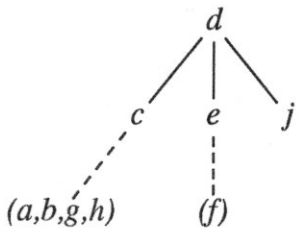


Figure 4

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 $\{c,d\}$, $\{a,b,c\}$, $\{b,c,d,e\}$, $\{d,e,f\}$, $\{e,f\}$, 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.

3.2 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 (given in Appendix I in pseudo-code) 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 Section 5.

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.

4. 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.

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 $nP+L$.

Table 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 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). We present a representative sample of those results here.

Graphs 1 and 2 indicate results for the average depth, with Graph 1 considering a static group size for a varying number of sites participating in the system and Graph 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 1 and 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 1), for which we show results in Graph 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 1 and compare the algorithms in detail. Clearly, for *N*, the number of messages, the propagation 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.

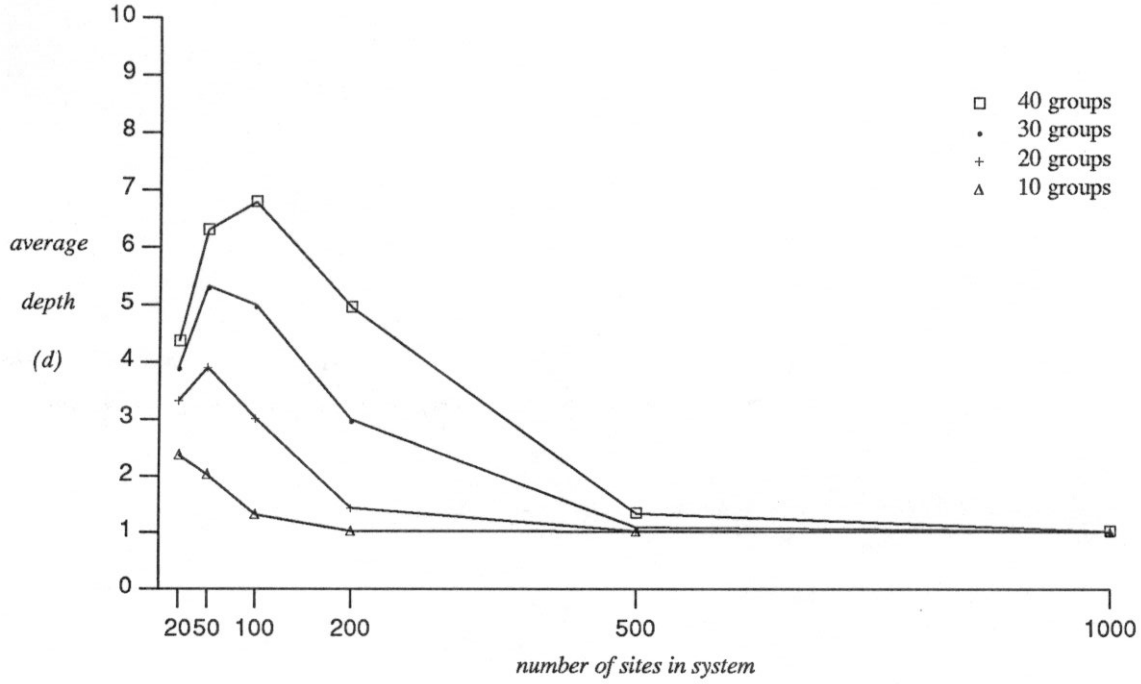
Also, it is important to remember that the depth computed in the experiments is an average over the worst case delay for a group. Thus, for any one group, the delay may be long for one or just a few nodes, whereas the other nodes may get the message with small

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, the 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.

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 the runs.

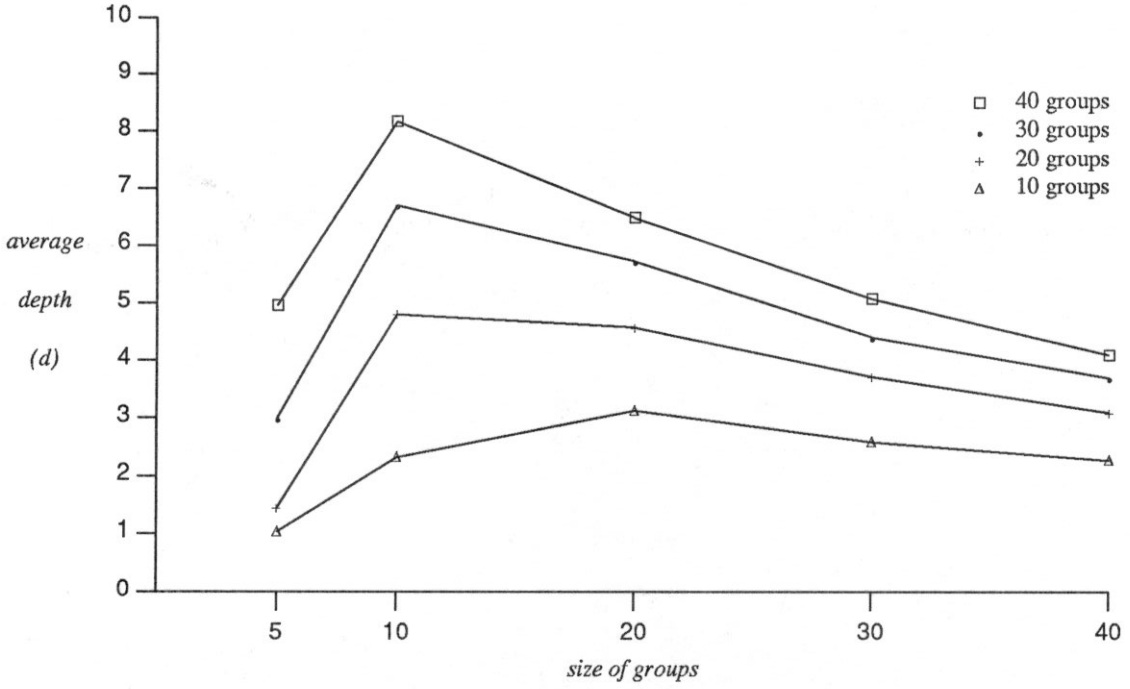
In Graphs 4 and 5 we show the load results, using a confidence interval of $\pm 10\%$ of the mean. For the propagation graph method, load is high when just a few sites are primary destinations and we have short, bushy trees. As trees get longer, load decreases until trees are sparse and load is very 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.

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 5(a). To reduce the load at a and b, we can transform our graph into the tree in Figure 5(b), which is correct and does not use extra nodes. The depth, however, has increased. Further, the propagation graph requires fewer messages than the two-phase algorithm, making it a desirable alternative



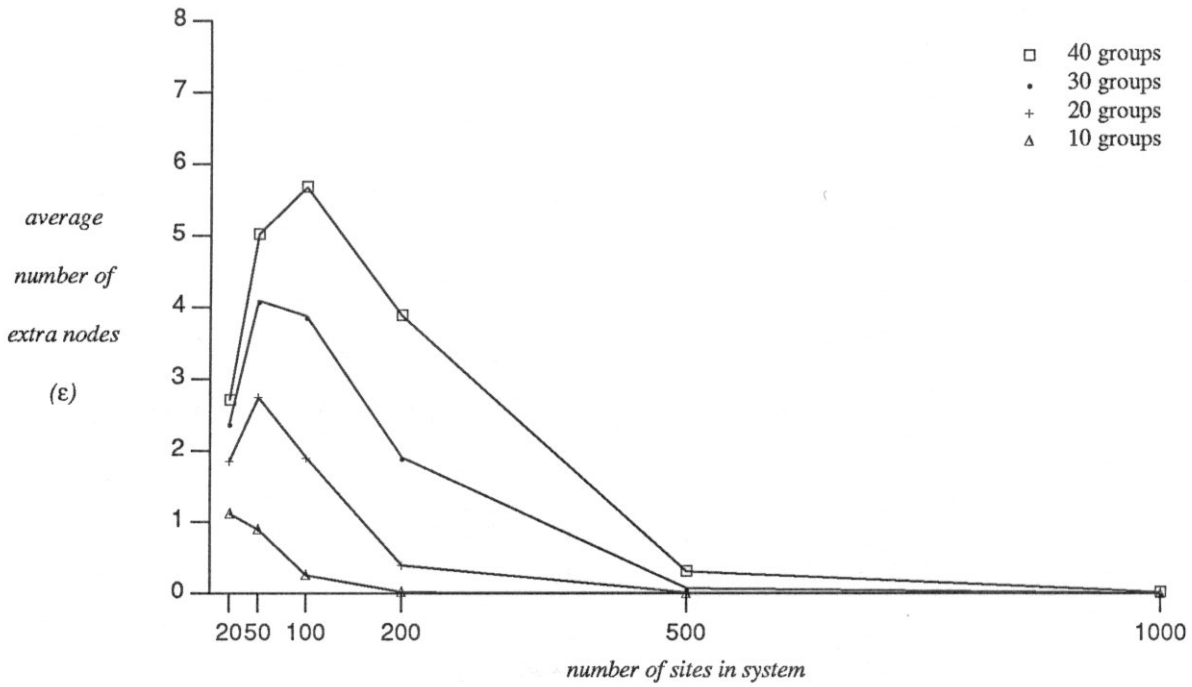
Group size 5

Graph 1



200 sites in system

Graph 2



Group size 5

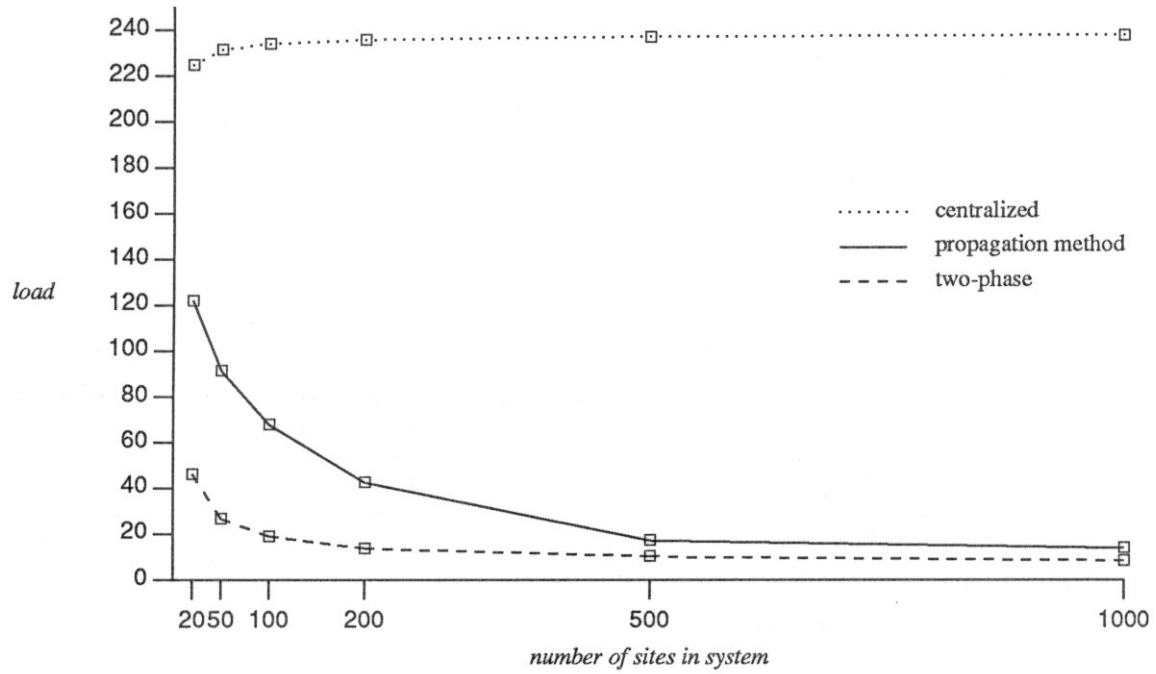
Graph 3

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.

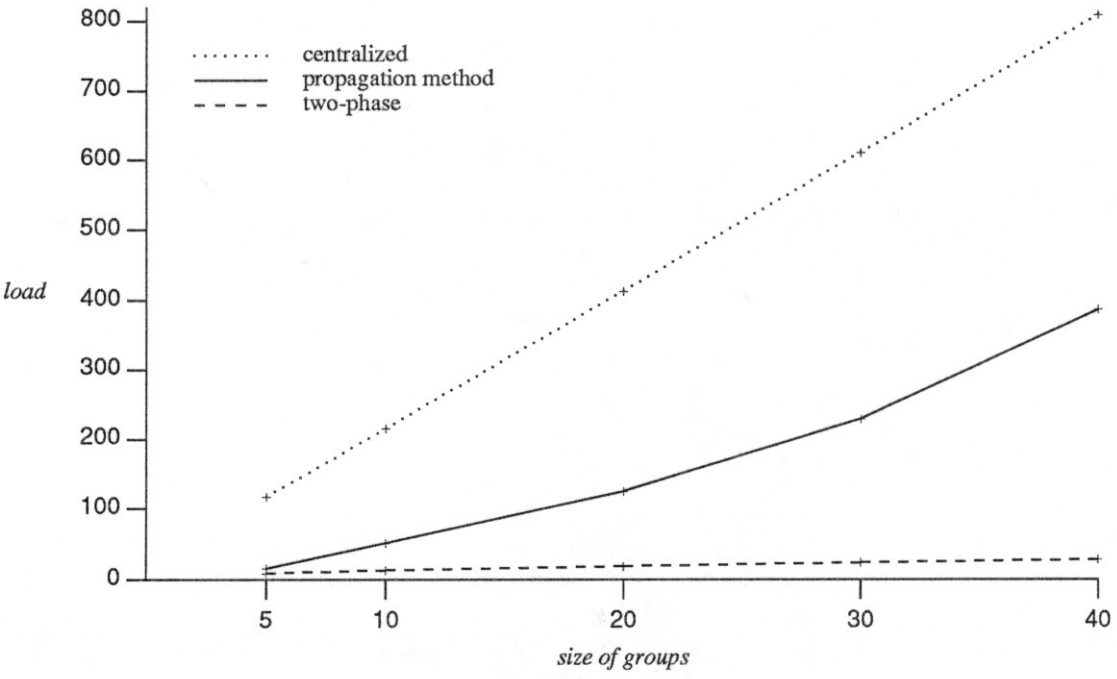
4.2. Broadcast Model

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



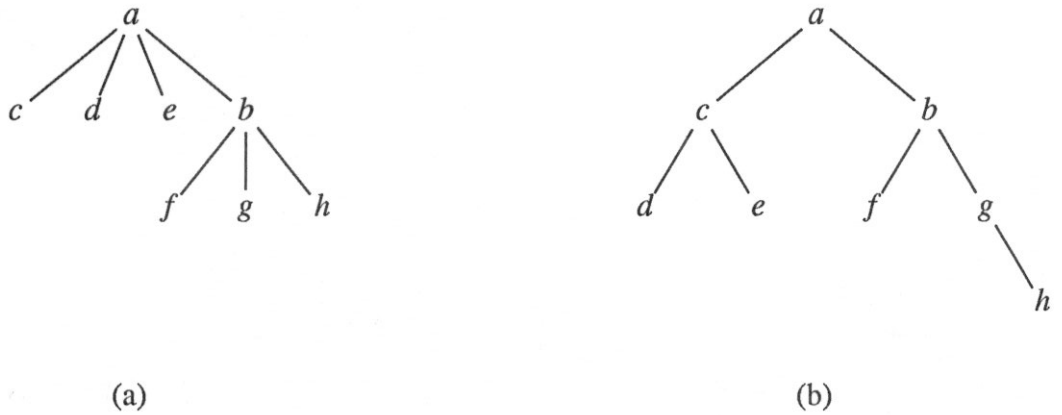
Group size 5

Graph 4



200 sites in system

Graph 5



(a) (b)

Figure 5

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 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 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 $d+1$.

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

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

Table 2

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 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. Performance Conclusions

It is evident that the propagation strategy provides a viable alternative for ordered multicasts in both point-to-point and broadcast networks. However, it (as well as the centralized solution) has one important drawback in both network cases: there is a substantial cost in setting up the propagation graph. Thus, the propagation approach will

only be of interest if a relatively large number of messages are sent during the lifetime of each group. We expect this to be the case in most of the applications that require multicast, e.g., updates to replicated data, software distributions and mailing lists.

Finally, we comment briefly on one optimization we have considered that attempts to reduce the number of extra nodes, thereby reducing the delay and the number of messages for the propagation strategy. This method involves amending the propagation graph in such a way that the tree structure is lost, but the solution is still correct. Unfortunately, it is not always possible to eliminate extra nodes and often the solution requires additional messages. Since the expected number of extra nodes is small according to our experiments, we do not elaborate further on this technique.

5. DYNAMIC MULTICAST GROUPS AND RELIABILITY

The solution presented in Section 3 is correct when there are no failures and the multicast groups are static. It is likely, however, that the multicast groups will change over time, thus requiring alterations to the propagation graph. 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, 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 describe extending the propagation algorithm to allow for dynamic multicast groups. This initial description applies when the changes to the groups are not due to failures. We follow this with two alternatives for providing reliable message delivery in the face of failures, one which makes use of dynamic multicast groups (with slight changes to account for failed sites) and one which does not. For the purposes of this paper, we do not provide all the details of these algorithms; instead, we present a high-level description that captures the essential features.

5.1 Dynamic Multicast Groups

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.

5.2 Reliability

Failures are inevitable in networks and we must consider how to ensure reliable message delivery despite them. As we mentioned in the Introduction, we believe that our approach can be made reliable to any desired degree. In this section we present two reliability alternatives that mesh nicely with our ordering strategy. Here we consider only fail-stop processor failures and assume there are no network partitions. The two methods for handling failures are atomic delivery and non-atomic delivery. Atomic 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 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.

5.2.1 Non-atomic Delivery

With this alternative, we make use of the 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 the last message it installed per 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 6, 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.)

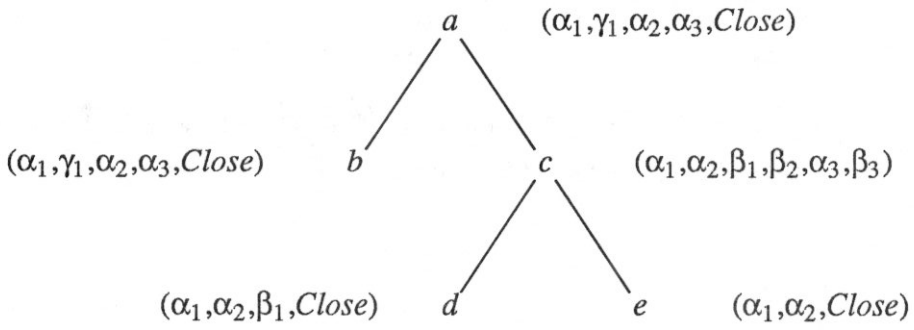


Figure 6

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. 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 what 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 and which site can provide each missing message.

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 8. (The merge of the *Close* messages can be disregarded at this

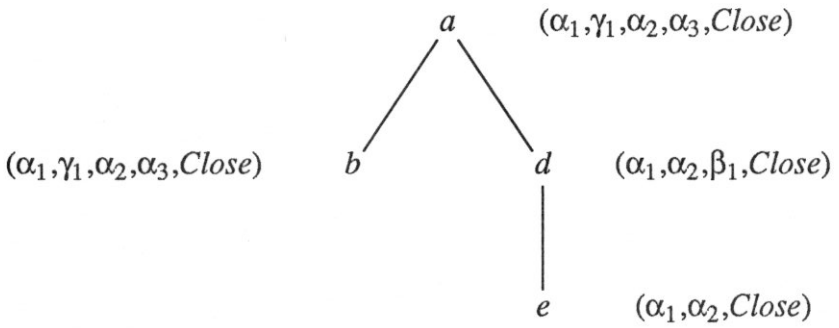


Figure 7

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*.

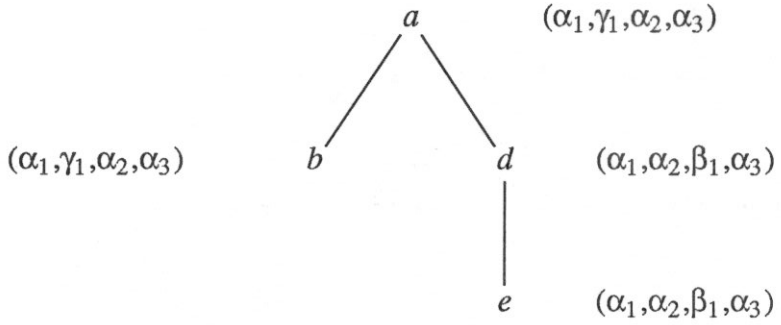


Figure 8

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 8. Thus, when *c* revives, the tail end of its message deliveries are out of order with respect to the other sites. It is necessary in this example to rollback delivery of β_2 , α_3, and β_3 at site *c*. Site *c* may redeliver them, using

the order defined by *d* along with any other messages it has missed while down.

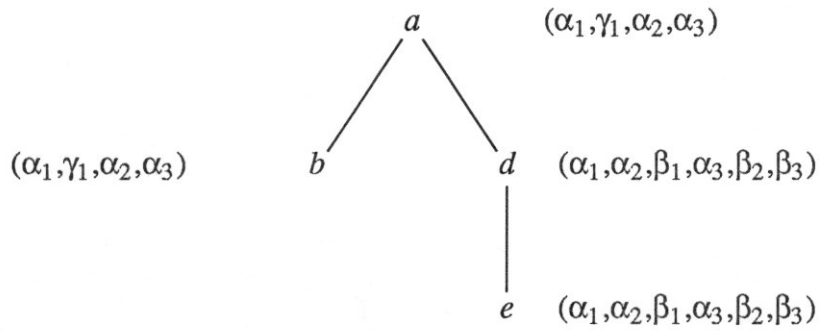


Figure 9

We also must consider the case of a site failure during the recovery protocol. Failure of the manager necessitates electing a new manager [Garc82] or maintaining a backup manager. (These are also used if the manager fails when there is no recovery in progress.) A simple polling of the roots can determine that an *Open* or *Close* is in progress. At the same time, the roots can be directed not to accept any more messages from the old manager which may be en route. The new manager completes the *Open*(s) and/or *Close*(s). If a participant site fails during recovery, each descendant of the failed site either has received the *Close* or has not received the *Close*. If not, if that site receives a new *Open* it will queue it until the *Close* appears (i.e., the site blocks). This will happen if either the failed site recovers or the manager runs recovery again (which it will do for the newly failed site). If it has received the *Close*, it may receive the *Open*, in which case it is keeping up with the other *Open* sites.

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.

5.2.2 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.

5.2.3 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 Section 4 increases considerably [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.

6. CONCLUSIONS

The propagation algorithm provides an efficient and distributed method for guaranteeing single source, multiple source and multiple group ordering properties of multicast messages. In most cases, it is superior to the two-phase algorithm in terms of number of messages. It is competitive with the two-phase method in terms of delay, and can surpass the two-phase method when the size of the groups is small, as we would expect in many applications. At the same time, the propagation algorithm alleviates the bottleneck associated with a centralized solution.

In addition, the propagation method provides flexibility by allowing different graphs for the same set of groups to accomodate the requirements of the network or application. Also, different degrees of reliability are achievable. Changes to the multicast groups are done quickly and easily.

Finally, the major weakness of this method is the set up cost of the propagation graph; thus, the technique should only be used for multicast streams that are long-lived, thereby amortizing the set up cost over many messages. Another disadvantage is that sometimes sites are required to handle messages which they do not need to deliver locally. These "extra" nodes, however, do not occur frequently according to our experiments.

7. REFERENCES

[Birm88] K.P. Birman, Private communication, July 1988.

[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.

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

[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 and N. Lynch, "Reliable Broadcast in Networks with Nonprogrammable Servers," *Proceedings of the Eighth International Conference on Distributed Computing Systems*, June 1988.

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

[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.

[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.

[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.

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

[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.

8. APPENDIX I

Here we specify the PG Generator and the MP Protocol in pseudo-code form.

The Propagation Graph (PG) Algorithm

```
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 ∈ unmarked_sites
   if ∃g ∈ unmarked_groups such that (s ∈ g ∧ current_subroot ∈ g)
   then
      intersecters ← intersecters ∪ s;

/* Mark all groups that contain subroot since now we have a primary destination
for them. */
for each g ∈ unmarked_groups
   if current_subroot ∈ 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 ← {g | g ∈ unmarked_groups ∧ ∃s ∈ g such that s ∈ intersecters};
repeat
   S ← {s | ∃g ∈ G such that s ∈ g}
   G ← G ∪ {g | g ∈ unmarked_groups ∧ ∃s ∈ G such that s ∈ S}
```

```
until no change to G
P_1 ··· P_k ← 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 ∈ 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 ∧ s ∈ 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;
```

Message Passing (MP) Protocol

```
message m = RECORD
{
   originator
   sender
   seq#
   contents
   dest_group
   receiver
};

/* At each site, array of next expected sequence numbers, one per sender. */
integer next_seq#_in[ ];

/* At each site, array of next sequence number to use for sending, one per receiver. */
integer next_seq#_out[ ];

/* At each site, a wait queue for out of sequence messages, one per sender. */
queue wait_queue[ ];

/* At each site, a local delivery queue. */
queue local_queue[ ];

/* At each site, a timeout to indicate when to send null messages, one
   per site to which it sends messages. */
clock time_msg_out[ ];

/* At each site, a timeout to determine when it has been too long
   since a sender has been heard from, one per sender. */
clock time_msg_in[ ];


/* The following handle processing a message at site me */


receive_message(m)
begin

time_msg_in[m.sender] = current_time;
if m.seq# = next_seq#_in[m.sender]
{
   if me ∈ m.dest_group
      queue_for_delivery(m);
   send_message(m);
   next_seq#_in[m.sender]++;
   check_queue(m);
}
else
   queue_for_waiting(m);

end
```

```
send_message(m)
begin

for all s such that s is a child of me in propagation graph and s is an ancestor of a site in m.dest_group in prc
{
   m.seq# ← next_seq#_out[s];
   m.sender ← me;
   m.receiver ← s;
   send(m) to s;
   next_seq#_out[s] ++;
}

end


originate_message(contents,gp)
begin

m.originator ← me;
m.receiver ← primary destination of gp;
m.dest_group ← gp;
m.contents ← contents;
m.seq# ← next_seq#_out[m.receiver];
m.sender ← me;
send(m) to m.receiver;

end



check_queue(m)
begin

m' ← head of wait_queue[m.sender];
if m'.seq# = next_seq#_in[m.sender]
{
   delete m' from wait_queue[m.sender];
   receive_message(m');
}

end



queue_for_delivery(m)
begin

insert m in local_queue;

end


queue_for_waiting(m)
begin
```

```
insert m in wait_queue[m.sender];
order wait_queue[m.sender] by m.seq#

end

say_I'm_alive()          /* Run periodically */

begin

for s such that me → s
  if (current_time - time_msg_out[s]) > threshhold_out
    send_null to s;

end

timeout()          /* Run periodically */

begin

for s such that s → me
  if (current_time - time_msg_in[s]) > threshhold_in
    recover

end
```

9. APPENDIX II

Correctness

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 $m_1 m_2$ and *b* delivers them in the order $m_2 m_1$. 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 vice-versa. 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 10, where *a* and *b* are supposed to receive type α messages, but *b* is not on a path for α messages. Figure 10 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.

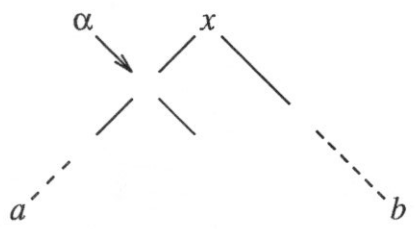


Figure 10