# MESSAGE ORDERING IN A MULTICAST ENVIRONMENT

Hector Garcia-Molina
Annemarie Spauster

CS-TR-161-88

June 1988

# MESSAGE ORDERING IN A MULTICAST ENVIRONMENT

*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 ensures that the messages are delivered to the appropriate processes. Some applications require that the protocol provide some guarantees on the order in which messages are delivered. 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 address the issues of performance and reliability by providing comparisons with other techniques for ordering multicasts. In many cases this new algorithm solves the problem with greater efficiency than previous solutions without sacrificing reliability.

# MESSAGE ORDERING IN A MULTICAST ENVIRONMENT

*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.

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 they come from different sources).

(c) *Multiple group ordering.* If messages $m_1$ and $m_2$ are delivered to two processes,

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 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). [KG87] considers another application that relies on ordered multicasts. See also [BJ87] for other uses of ordered multicasting. Reference [CD85] discusses multicasting in internetworks and gives strong justification for the use of multicasting.

The objective of our work is to present a novel message ordering technique that guarantees all the properties listed above. Of course, in many cases, the multicast protocol must exhibit some *reliability properties*, as well. Here, however, we will focus on ordering and not reliability. We view the reliability issue as roughly 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. We therefore study message ordering separately. Eventually, the ordering mechanism must be combined with the reliability mechanisms and in Section 6 we briefly touch on this subject.

## 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 way to do it is to assign a timestamp to each message at the source and to 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 $\alpha$ 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 $\alpha$ source sites (e.g., $y$). For property (c), site $c$ must in addition check with potential $\beta$ sources. If the potential sources are unknown, $c$ must check with *all* sites in the system.

Birman and Joseph have proposed another solution [BJ87] that is especially

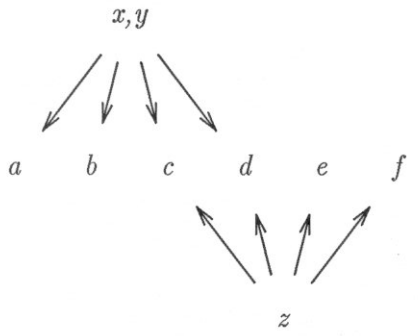



Figure 1

interesting. The algorithm (attributed in part to Dale Skeen) 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 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. 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.

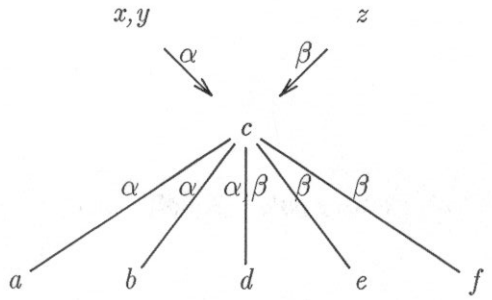



Figure 2

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 $\alpha$ messages from $x$ and $y$ to $a$ and $b$, the $\beta$ messages from $z$ to $e$ and $f$, and the merged $\alpha$, $\beta$ messages to $d$. Thus, all sites deliver their messages in the order defined by site $c$.

It is important to make the distinction between "logical" and "physical" paths. The propagation graph indicates logical paths. For instance in Figure 2, an $\alpha$ message must be sent from node $c$ to destination $a$, but it is not necessarily the case that there is a direct link from $c$ to $a$. The message will follow some physical path from $c$ to $a$ that we do not indicate here.

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 $\alpha$ and $\beta$ 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:

$$\alpha_1 = \{c,d\},\ \alpha_2 = \{a,b,c\},\ \alpha_3 = \{b,c,d,e\},\ \alpha_4 = \{d,e,f\},\ \alpha_5 = \{e,f\},$$
$$\alpha_6 = \{b,g\},\ \alpha_7 = \{c,h\} \text{ and } \alpha_8 = \{d,j\}.$$

Site $d$ is the primary destination for $\alpha_1$, $\alpha_3$, $\alpha_4$ and $\alpha_8$, $c$ is the primary destination for $\alpha_2$ and $\alpha_7$, $e$ is the primary destination for $\alpha_5$ and $b$ is the primary destination for $\alpha_6$. Note that messages do not necessarily flow down to the bottom of the tree. For instance, $g$ only receives $\alpha_6$ messages.

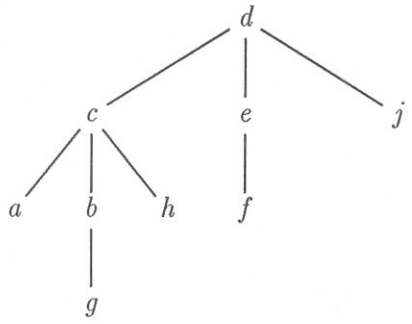



Figure 3

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 4. Once a site knows what the graph looks like, it uses the MP protocol to send, receive, propagate and forward messages.

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 $\alpha$, then $x$ gets all messages destined to group $\alpha$.

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

(PG1) For every group $\alpha$ there is a 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 $\alpha$ is a member of $\alpha$; and

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

When there exists a node $a$ on the path from $p$ to $x$ where $a$ is not a member of $\alpha$, 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 does generate 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 $\alpha_9$. We discuss the impact of extra nodes in Section 5 and in the conclusions we briefly mention how to eliminate some of the extra nodes.

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.

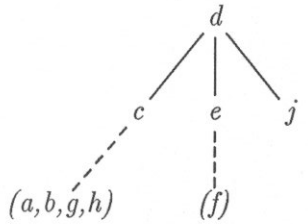



Figure 4

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 one 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. This guarantees that a receiver can order the messages from a sender correctly in case they arrive out of order. Also, each site keeps track of which sequence number it expects next from each sender. In addition, each site maintains a queue for messages destined to its local process, and a wait queue for messages received from other sites that are 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. DYNAMIC MULTICAST GROUPS

The solution presented in Section 3 is correct for static multicast groups. In this section we describe modifications to the algorithm that allow for deleting, adding and modifying groups dynamically. The main goal is to perform such changes without requiring too much coordination among the sites and without preventing the delivery of messages for long periods of time.

One possibility is to have one site compute the new graph and use a type of 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 change to the graph consistently.

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. 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 *Open*s 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. When a site receives an *Open* it incorporates the new tree information, is able to process messages again according to the new graph and propagates the *Open* to its new children.

There are several issues that must be resolved in order for this to be correct. We cannot go into these in detail here, but we briefly mention some of them. First, when the graph changes, so may the primary destination. The old primary destination may have queued messages that it is no longer responsible for ordering. These must be given up to the new primary destination. Second, due to communication delays, *Open* and *Close* messages for different graphs may become intermingled. Graph sequence numbers are required to ensure that all sites have *Open*ed the same graph.

Finally, failure of the manager must be considered. It turns out that this is easily tolerated. The remaining sites can elect a new manager or there can be a backup manager. A simple polling of the roots can determine if there is an *Open* or *Close* 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. If there is no *Open* or *Close* in progress, the new manager then easily takes over. Otherwise, the new manager can complete the *Open*(s) and/or *Close*(s) and then take over.

## 5. PERFORMANCE

In this section we compare the performance of several multicast message ordering algorithms. Due to space limitations, we only consider the following point-to-point network model. 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 $nP+L$. We compare our propagation method to the two-phase algorithm of [BJ87] and to a strictly centralized version of [CM84] †.

We look specifically at two performance measures: $N$, the number of messages required to send a multicast under the multiple group ordering property and $D$, 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. Table 1 indicates the performance of the three methods for the two measures.

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 $n+\epsilon$, where $\epsilon$ 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 $2L+nP$.

† 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.

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.

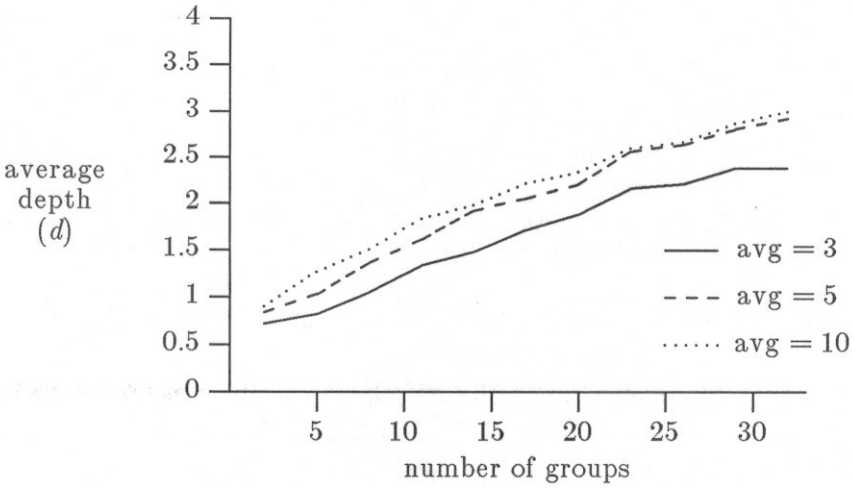
The total delay in this case, then, is the sum of the delays from the source to the primary destination and the delay from the primary destination to the group member that is furthest away (at depth $d$). The delay from the source to the primary destination is simply $L+P$. It is simple to show that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

Clearly, the performance of the propagation algorithm depends on the values of $\epsilon$ and $d$. It turns out that in most cases of interest, $\epsilon$ and $d$ are small. We established this via experiments on randomly generated multicast groups. For a given number of total sites, a number of groups, and an average size of a group (exponentially distributed), we generated a random set of multicast groups and computed their propagation graph. Graphs 1 and 2 show results for a fixed number of sites (20) and sizes of groups ranging from 2 to 30. The three curves represent average group sizes 3, 5 and 10. Each data point is an average over 100 runs of the average value of $d$ or $\epsilon$ for all the groups in a run.

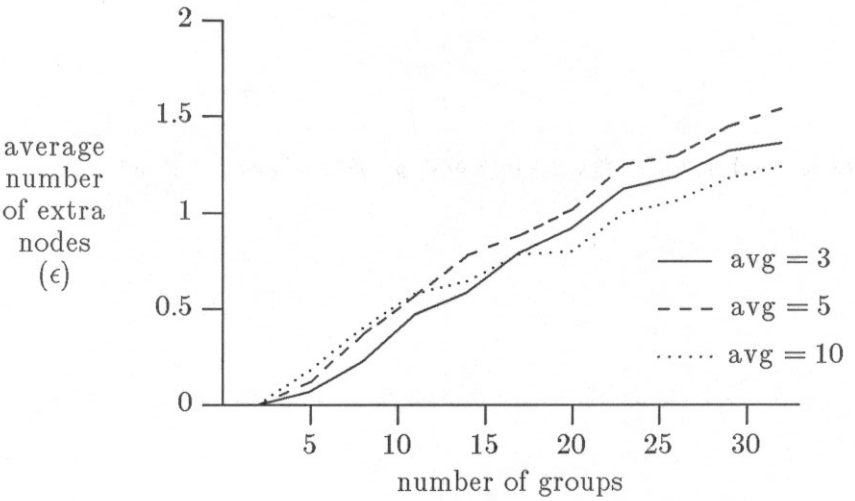
With $\epsilon$ and $d$ small, the propagation algorithm performs similarly to the centralized algorithm and both of these are substantially better than the two-phase method for both $N$ and $D$. Of course, since the work of ordering and propagating messages is distributed over many sites, the propagation algorithm does not have the bottleneck problem of the centralized case. So in this sense the propagation algorithm is superior to the centralized algorithm. Although we did not discuss broadcast networks here, we note that a similar analysis shows that the propagation method works well in most cases for that model, too.

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

Table 1




Graph 1




Graph 2

Finally, the propagation strategy (as well as the centralized one) does have one important drawback: 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.

## 6. RELIABILITY

As we mentioned in the introduction, we believe that our approach can be made reliable to any desired degree. In this section we outline two of the reliability alternatives that mesh nicely with our ordering strategy (assuming fail-stop processor failures and no network partitions). In addition, we make some remarks regarding the performance of reliable ordering.

With our first alternative, sites do not block but message delivery is not atomic. Sites constantly monitor the sites on the propagation tree from which they receive messages and to which they forward messages. If a failure is detected, a two phase process is initiated among the survivors and the manager (see Section 4). 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 send Close messages directly to the survivors.) Each site reports back to the manager the last message that was installed before termination of each group. In the second phase, each survivor installs any missing messages that the other sites reported. At the end of the protocol, all survivors have delivered to their local processes exactly the same set of messages, in the same relative order, to the terminated groups. After the recovery, new multicast groups can be defined (involving only the survivors) and transmissions can resume.

We have omitted many details (e.g., how sequence numbers are used to identify the missing messages, the fact that the recovery messages might flow along the remains of the propagation graph, etc.). However, there are two important characteristics we

note. The first 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 non-reliable versions during normal operation is roughly the same. The second characteristic is that messages are not delivered as an atomic operation. When a site recovers from a failure (and only in this situation), it may discover that it installed messages in the wrong order. It will have to roll back the installation (e.g., undo a transaction), and then redo the missed messages in the correct order.

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 site that is not a subroot 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 subroot fails, however, the sites to which it propagates messages must block on the messages they ordinarily receive from the subroot. When the failed subroot recovers, it can continue forwarding messages from where it left off.

Note that the other solutions, in particular the two-phase protocol of [BJ87], may block under the same conditions (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 (second 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 by making centralized ordering decisions (enforced via sequence numbers) and blocking sites when failures occur.

## 7. CONCLUSIONS

The propagation algorithm provides an efficient and distributed method for guaranteeing the multiple group ordering property for multicast messages. In most cases, it is superior to the two-phase method in terms of number of messages and delay. At the same time, it alleviates the bottleneck associated with a centralized solution. In addition, it allows for changing the multicast groups quickly and easily. It is flexible enough to provide differing degrees of reliability. Its major weakness is the set up cost of the propagation graph, so it should only be used for multicast streams where the set up cost can be amortized over many messages.

Finally, we point out that one disadvantage of the technique 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 experiments presented here. Further, we are studying ways to alter the propagation graph so that the number of extra nodes can be reduced. This involves adding edges to the graph that violate the tree property but still guarantee consistent ordering.

## 8. REFERENCES

[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.

[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.

[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.

[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.

## 9. 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 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[ ];


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


receive_message(m)
begin

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 me → s and s is an ancestor of a site in m.dest_group
{
    m.seq# ← next_seq#_out[s];
    m.sender ← me
    send(m) to s;
```

```
    next_seq#_out[s] ++;
}

end


originate_message(contents,gp)
begin

m.originator ← me;
m.receiver ← primary destination of gp;
m.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
```

## 10. 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 $\alpha$, then initially they are ordered by the primary destination for $\alpha$. 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 $\alpha_1$ and $m_2$ is destined for $\alpha_2$. 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 $\alpha_1$. 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 $\alpha_1$ to $a$ and only one path from $\alpha_2$ 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 5, where $a$ and $b$ are supposed to receive type $\alpha$ messages, but $b$ is not on a path for $\alpha$ messages. Figure 5 indicates that the PG generator places nodes $a$ and $b$ in different subtrees of $x$, even though they are in the same group ($\alpha$). This is an impossibility since the partitioning step of new_subtree puts all sites that share groups in one subtree of the current subroot.

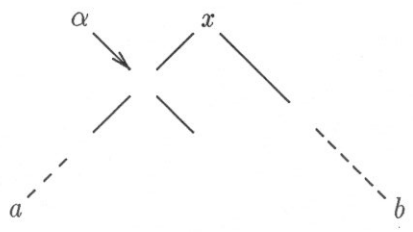



Figure 5