Princeton University

RELIABLE BROADCAST IN NETWORKS WITH NONPROGRAMMABLE SERVERS

Hector Garcia-Molina
Boris Kogan
Nancy Lynch

CS-TR-123-87

November 1987

Department
of
Computer Science

DEI SVB NVMINE VIGET

RELIABLE BROADCAST IN NETWORKS WITH NONPROGRAMMABLE SERVERS[†]

Hector Garcia-Molina
Boris Kogan

Princeton University

Nancy Lynch

Massachusetts Institute of Technology

ABSTRACT

The problem of implementing reliable broadcast in ARPA-like computer networks is studied. The environment is characterized by the absence of any multicast facility on the communications subnetwork level. Thus, broadcast has to be implemented directly on hosts. A reliable broadcast protocol is presented and evaluated on several important performance criteria.

November 25, 1987

RELIABLE BROADCAST IN NETWORKS WITH NONPROGRAMMABLE SERVERS[†]

Hector Garcia-Molina

Boris Kogan

Princeton University

Nancy Lynch

Massachusetts Institute of Technology

1. Introduction.

Reliable broadcast of messages in point-to-point computer networks is an important distributed application which has received considerable attention. A simple and obvious way to broadcast a message is to send a separately addressed copy of it to every host in the network and repeat this process until an acknowledgment is received. This solution, however, leaves room for possible improvement in several directions. First of all, this solution is clearly inefficient since it can generate much more network traffic than necessary.

Efficiency could be improved if the network servers were programmed to handle broadcast messages intelligently. This approach is taken in [AwEv84], [DaMe78], [Peac80], [Rose80], and [SeAw83]. Unfortunately, it is not always

[†] 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.

applicable. For instance, Arpanet users cannot program that network's servers (IMPs), nor are the servers preprogrammed to implement broadcast efficiently. However, even when servers are nonprogrammable, one can still achieve better efficiency than with the simple solution. In particular, expensive communication links can be identified and avoided whenever possible.

While the simple solution has reliability provisions in the form of acknowledgments this is not always adequate. Consider, for example, a situation when the broadcasting host gets disconnected form the network after delivering the message only to a portion of all hosts. The rest of the hosts will never (or until the source is reconnected) receive the message. Therefore, we would like to have a broadcast algorithm in which all hosts share the responsibility for reliable message delivery so that in the described scenario the hosts that successfully received the message from the source could then propagate it to others.

Finally, improvement can also come from taking advantage of the fact that broadcast applications usually operate on streams of many messages rather than on a few isolated messages. By ordering messages at the source and keeping track of the messages received so far at every host the algorithm we propose will be able to dynamically make decisions on how to propagate newly generated messages. The benefits gained in terms of reliability and low delay will outweigh the extra communication cost involved when the broadcast stream is sufficiently long (consists of many messages).

It is important to note that reliability is treated here as a relative measure rather than an all-or-nothing property. That is, instead of classifying protocols as reliable or unreliable, we try to estimate to what degree they are reliable (or unreliable). No statement can be made about the reliability of any broadcast protocol without first making some assumptions concerning the reliability of the network itself. For example, if the network stays in a partitioned state for an indefinite period of time, no protocol, no matter how clever, can guarantee reliable delivery of broadcast messages to all destinations. On the other hand, if the partition is repaired for a brief period of time, only to reappear and persist, some protocols might be able to take advantage of this brief opening to complete a broadcast while others might not. Thus it seems more justified to speak of *relative* reliability of a protocol, referring to the degree to which it is capable of utilizing communication opportunities presented by the dynamically changing network. This issue is discussed in greater detail in subsequent sections.

Interestingly enough, not all applications that make use of broadcast require that it be reliable. For example, in adaptive routing it may be necessary to distribute the information regarding queueing delays in different parts of the network. Broadcast could be used for this purpose. However, if a broadcast message is late in coming, due to communication failures, it may just as well not arrive at all because it will soon be outdated by a more recent one anyway.

So it seems useful to keep in mind some specific applications which require reliable broadcast. The main motivating application that has been driving the present work is management of highly available replicated databases. There are several known techniques for solving the problem of high data availability in replicated databases in the face of network partitions, all of which require reliable broadcast of updates. But while the goal of reliable broadcast is to eventually deliver all messages to all destinations, there are some particulars associated with certain approaches. For example, in the type of approaches that forego serializability of transaction execution in order to achieve maximum data availability (e.g., Data-Patch [Garc83], log transformation [BlKa85], [Sari85], and data fragmentation [GaKo87]), it is not absolutely essential that updates be installed in remote copies of the database always in the correct order, i.e., in the order they were generated. Consequently, it is not essential that broadcast messages be always delivered in the order they were dispatched.

In designing a reliable broadcast we take into account this consideration. As a result, the stress is put on delivering messages as promptly as possible, but not necessarily in the strict order. Note that this relaxation of requirements on a reliable broadcast gives potentially more flexibility to the protocol and may improve its average delay characteristic.

2. Basic Assumptions.

In this section, the chosen network environment is described in more detail, and some motivations for considering this environment are introduced.

The network consists of a set of *hosts*, communication *servers*, and communication *links*. Hosts are computers that participate in the broadcast application. Servers are nodes interconnected among themselves by point-to-point bidirectional links into a communication subnetwork. (This study can be extended to the case when some of the links are of the broadcast type, however we choose not to consider this extension here.) Each host is attached to a server. Some servers, however, may have no corresponding hosts, and, therefore, act only as switches.

In reality, a server is either a separate dedicated communication processor (e.g., Arpanet) or a process residing at the same physical computer with the corresponding host (e.g., Bitnet). If the latter is true, a clean interface between the host and the corresponding server is assumed. For our purposes it is both convenient and sufficient to assume that servers are separate nodes.

There is no multicast facility provided by the network, and servers cannot handle messages with multiple addresses. The only kind of instruction a host can give to a server is request it to deliver a message to a single destination. Thus if the same message is to be sent to several destinations, the above procedure has to be repeated several times. Servers are assumed to be nonprogrammable as far as the broadcast application is concerned, i.e., the code that is run on the servers cannot be changed to expedite reliable broadcast. That leaves the only remaining alternative: implementing broadcast on the hosts.

The kind of scenario described in the previous paragraph is quite realistic. It may arise in a network of the type of Arpanet (which still does not provide a multicast facility), when (some of) the hosts connected to the network wish to enact efficient and reliable broadcast for a common application.

The host that issues broadcast messages (which will also be called *data messages*) is referred to as the *source.* Here, we study only a single-source broadcast problem. However, a multiple-source broadcast can be performed reliably by running several identical single-source protocols suggested in the present paper. From the point of view of efficiency this option also appears to be a reasonable one.

The hosts are reliable and never fail. The servers and links, however, can fail. In view of this latter assumption, the assumption concerning the reliability of hosts is no longer overly restrictive, for a host crash can now be "simulated" by a server or link failure, provided of course that hosts are equipped with non-volatile data storage.

We make no assumptions about communication failures in the network other than the impossibility of malicious messages being generated. Links can fail and recover at any time. Messages can arrive out of order, have arbitrary delays, be lost at any point (even when the link over which the lost message was sent is perceived to be operational), or be spontaneously duplicated. Moreover, the fact that a message is lost is not automatically detected by the communication subsystem and, therefore, cannot be reported to the application. Similarly, failures of links and their recoveries are not detected either. Thus, the application can never be certain whether a given link is operational at any given moment.

The reason for making as few assumptions as possible about the way the communication network behaves, particularly the way in which it may fail, is to design a protocol that does not depend for reliability on the data link layer of the network [Tane81]. There is a growing feeling among the researchers in the field against such dependency. Moreover, even though most of the existing networks have reliability mechanisms — such as message acknowledgments — implemented at the data link layer, it is likely that future designs will favor pushing these mechanisms up to the application layer. A strong efficiency argument can be made in favor of such arrangement.

The next assumption will be referred to as the *communication transitivity* assumption. It postulates that if during the time interval $(t,\ t')$ it is possible for host x to communicate with host y, and for host y to communicate with host z, then it should be possible, during $(t,\ t')$, for host x to communicate with host z. The significance of communication transitivity will become apparent when we discuss the particulars of the proposed protocol. The assumption seems quite reasonable for networks with adaptive routing since in a situation described there exists at least one communication path between hosts x and z — the one that goes through (the server of) host y.

Hosts possess no knowledge of the network topology or any other static information concerning the network. They do, however, know the identities of other participating hosts. (When this latter assumption is not valid, i.e., some hosts do not know the identities of all other hosts, the problem becomes very different. See [Deme87] for a possible solution.)

Finally, we assume that there is a division of all links into two categories, according to their bandwidth. All links with a high bandwidth are called *cheap*; links with a low bandwidth are called *expensive*. For obvious reasons, it is not specified what high and low mean precisely, but we assume that expensive links are much more expensive than cheap ones. This assumption is motivated by the existence of long haul networks (with low bandwidth links) with local networks (with high bandwidth links) integrated into them. In a global network of this kind some hosts are connected via cheap links while others are connected via expensive links only.

Since they have no static information about the network, hosts do not know which links are cheap and which are expensive. We assume, however, that there is a way for a host to tell whether the message it has just received traversed an expensive link on its way to the destination. (For instance, there could be a special bit in the message format initialized to 0 and set to 1 by a server whenever the message in question traversed an expensive link. Even if the network did not provide this type of service, the service could be implemented on the host level. One way to do this would be to timestamp each message at the time it is sent out. This would allow each host to estimate the time in transit. Since the expected times for cheaply delivered messages and for expensively delivered ones vary significantly, hosts would be able to tell them apart.) The ability to distinguish expensively delivered messages from cheaply delivered ones is the only kind of dynamic information available to hosts.

3. Basic Ideas.

As was mentioned earlier, the goals of our protocol should be low cost, low average delay, and high reliability. In this section we focus on some basic ideas on how to achieve these goals, without going into details of our proposed algorithm.

We start with a fairly obvious observation, namely that optimal cost cannot be achieved for broadcast in our environment. This is illustrated by the example in Figure 3.1. (In all examples, from here on, hosts will be denoted by squares, and servers by circles.)

In this example we have three hosts connected by a network of four servers. Host h_1 is the source of broadcast. Clearly, the most cost efficient (as well as the delay minimizing) way for h_1 to broadcast a message would be as follows. Host h_1 hands the message to its server (s_1). Server s_1, then, sends it to server s_4. Sever s_4 makes two copies of the message and sends one copy each to servers s_2 and s_3. Finally, servers s_2 and s_3 pass the message on to hosts h_2 and h_3, respectively. In this way, no link is traversed more than once (and, obviously, every link has to be traversed in the given example for the broadcast to succeed).

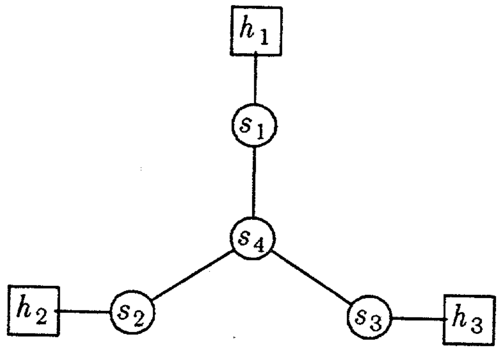

Figure 3.1.

Note, however, that servers cannot handle multiply addressed messages, nor is there any way for host h_1 to explicitly instruct server s_4 to duplicate the message and send the copies to two separate destinations (and even if there were, that would do no good because h_1 knows nothing of the network topology). Hence, broadcast cannot possibly be performed as described above. So, no matter what type of protocol one comes up with for our environment, it will not, in general, have optimal performance.

We have implicitly assumed here that the cost of a protocol is measured in the number of times a message traverses a communication link in the network. However, since, for a protocol implemented on hosts, there is no way of knowing how many links a host-to-host message traverses or even how it is routed by the network, this cost metric does not present a very good basis for comparison of different protocols. Thus we introduce a new metric: the number of packets[†] delivered (at least part of the way) over expensive links. This metric seems acceptable for two reasons. First, it closely approximates the actual cost of broadcast because it is assumed that high-bandwidth (cheap) links are much cheaper to use than low-bandwidth ones. Second, since the only thing that can be said about how a packet has been delivered to its destination is whether it has traversed any expensive links, this seems the *only* way to estimate the real cost.

At any given time, all the hosts in the network can be divided into groups such that within each group hosts can communicate among themselves cheaply, but hosts in different groups can only communicate using expensive links. Such a group of hosts is called a *cluster*. Clustering of hosts can change over time due to failures and repairs of communication links.[*] Note that hosts' views of the constituency of their clusters may not always conform to reality.

Assuming that the network is not partitioned and disregarding for now the possibility of any changes in it, it would be desirable to arrange clusters in a

[†] A packet is a physical embodiment of a message. Each time a message is sent from host to host, it becomes a *new* packet.
[*] In this context, repair can also mean an introduction of a new link.

tree rooted at the cluster containing the source. This tree is referred to as the *cluster tree.* Then broadcast messages could trickle down the tree from parent cluster to child cluster. Barring the possibility of lost inter-clsuter packets, it would take $k-1$ expensive packets, where k is the number of clusters in the network, to broadcast one message. Clearly, this is optimal (in the new metric).

Every cluster, except the one at the root of the cluster tree, has a special host in it, called the *cluster leader.* The cluster leader receives broadcast messages from one of the hosts in the parent cluster, and it is responsible for distributing them to other members of its own cluster (cluster neighbors). Broadcast is initiated when the source sends a message to its cluster members. Figure 3.2 gives an example of a cluster tree.

As was mentioned before, the cluster tree arrangement represents a demontsrably good option for minimizing the cost of broadcast. If we also want low average delay, however, it is not enough to come up with just any cluster tree. The main idea for reducing delays is, for every cluster, to try to find a parent that can deliver new broadcast messages as promptly as possible. Namely suppose that, for a given cluster C, we have a choice of parents C' or C''. Further, suppose that somehow it is known that cluster C' receives broadcast messages ahead of C''. Then C' is a better candidate for a parent than C'', and cluster C should become a child of C'. Note that at a later time, due to changing message traffic, some other cluster can become a more desirable parent for C than C'. Thus, we may have to *dynamically* restructure the cluster tree to minimize delays.

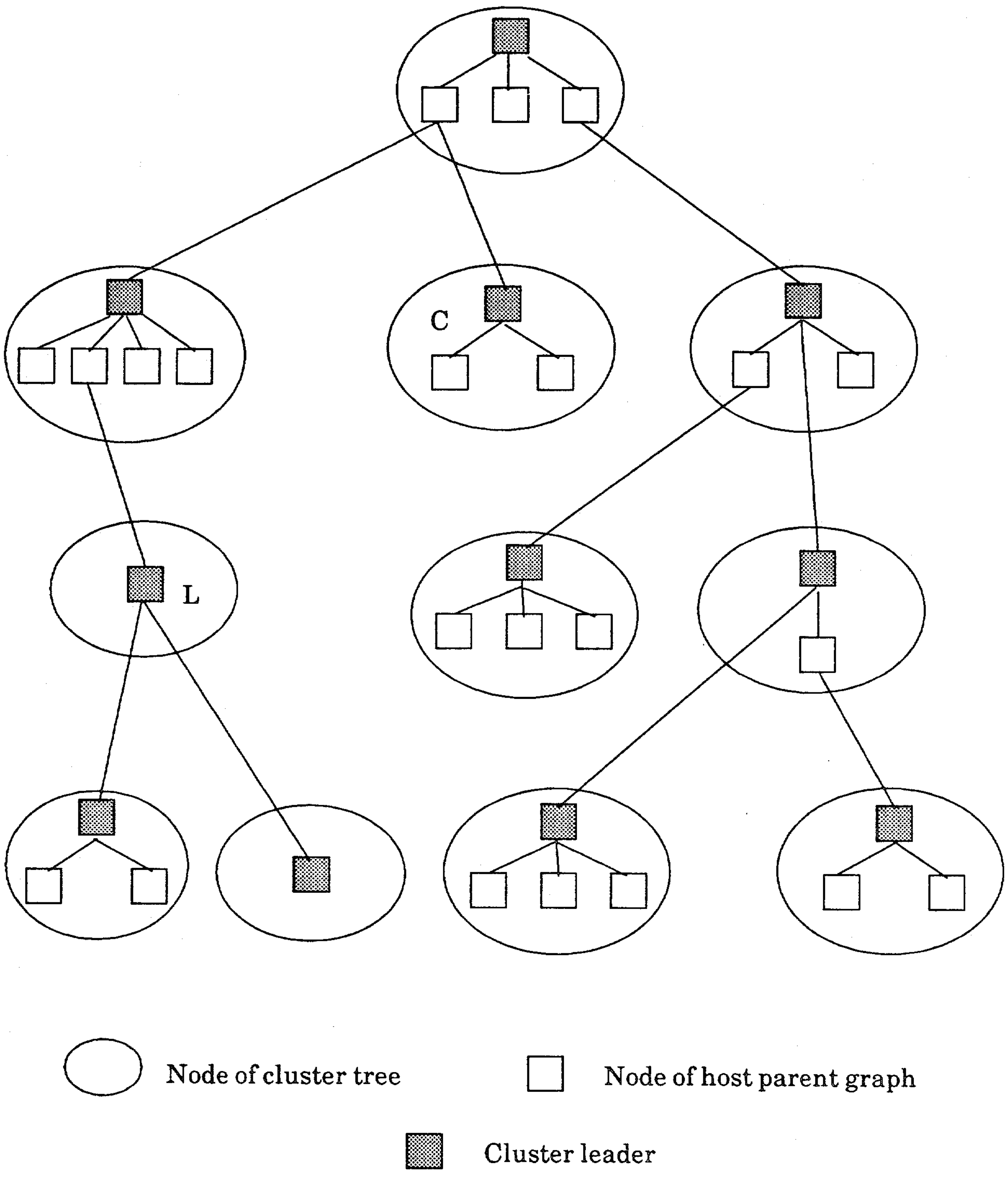
C
L
Node of cluster tree
Node of host parent graph
Cluster leader

Figure 3.2.

Dynamic changes in the cluster tree may also eb necessary to compensate for component failures. For example, if a cluster leader finds out that it no longer can communicate with its parent, it should try to find a new parent. In another instance, a cluster leader (or its server) may fail, in which case the members of the cluster must come up with a new cluster leader to maintain the connectivity of the tree.

Failures can also cause packets to get lost, and the reliable broadcast algorithm must compensate for this. To detecst lost packets, all broadcast messages are sequenced numbered so that it is easy to tell when a message has failed to be delivered to any given host. When that happens, certain actions are taken to enact a redelivery of a lost packet.

4. The Algorithm.

In the previous section some high level strategies for enacting efficient reliable broadcast were outlined. Nothing was said, however, about any specific ways of implementing those strategies. In particular, it was not explained how a cluster tree can be constructed and maintained. In this section, the actual broadcast algorithm is presented. (For a formal specification of the algorithm see Appendix.)

4.1. The Parent Host Graph.

To enact broadcast, hosts attempt to configure themselves into a tree with the source as its root. In a failure-free environment, such a tree would be stable, and data messages could be sent from parent to child to make broadcast complete. However, because of the possibility of link failures, the tree can become disconnected, and the nodes should be able to reconfigure into a different tree if at all possible. The resulting structure is, therefore, dynamic and referred to as the *host parent graph*, to underscore the fact that connectivity is not always achieved (e.g., during partitions).

(It is important not to confuse communication links of the network with edges of the host parent graph. For the latter correspond to communication paths that, in general, can consist of several links.)

We say that a host parent graph *induces* a cluster tree if (1) the host parent graph is a tree; and (2) children of every cluster leader include all other hosts that are in the same cluster. (For the sake of generality, from now on, we consider the source as a cluster leader.) The relationship between parent host graphs and cluster graphs induced by them is best illustrated by the example in Figure 3.2. There, the little squares (representing hosts) are the nodes of the parent host graph, and all the connections between the squares are its edges. The large ellipses (representing clusters), on the other hand, are the nodes of the cluster tree, whose edge set includes only those edges that go between the ellipses. It is easy to see that the host parent graph in this example induces the cluster tree shown in the picture.

Not every parent graph, though, induces a cluster tree. Consider again the parent graph of Figure 3.2. Suppose, however, that a high bandwidth path has just been repaired between clusters C and L. That means that these two clusters have been joined into one. Hence, the cluster tree is changed, and the parent graph no longer induces the new cluster tree.

To reduce the cost of broadcast, it is desirable to have a host parent graph that induces a cluster tree. Thus the algorithm must maintain a host parent graph in such a way that it dynamically adjusts to changes in the network and tends to assume a configuration that induces a cluster tree.

Broadcast messages are propagated in the host parent graph from the root all the way down. Thus, upon receipt of a broadcast message, a host sends it on to all its children. For reasons explained below, a host can accept a message sequence-numbered higher than any it has received so far, only from its parent. If such a message arrives from any other host, it is discarded.

4.2. The Attachment Procedure.

At the heart of the algorithm is the attachment procedure, which is periodically activated at every host. The purpose of this procedure is to make sure that the host is attached to a "good" parent, and if that is not the case, find a better one.

As was mentioned earlier, broadcast messages are sequence numbered. Every host keeps track of all the messages it has received so far. For each host i, a set $INFO_i$ contains the sequence numbers of all messages received by i. Let us define a partial ordering $<$ on sets of message sequence numbers. We write $A < B$ if the largest element of A is strictly less than the largest element of B, i.e., if $\max_{q \in A}(q) < \max_{q \in B}(q)$. Also, we write $A \simeq B$, if $\max_{q \in A}(q) = \max_{q \in B}(q)$. These sets are used for detection and redelivery of lost packets. They are also used for dynamically maintaining the host parent graph with the goal of maximizing reliability and minimizing delays.

Each host i maintains an array of sets of message sequence numbers, MAP_i. $MAP_i[j]$ represents host i's view of $INFO_j$ (thus, $MAP_i[i] = INFO_i$). Hosts periodically update one another on the current values of their $INFO$ sets. $INFO_s$, where s is the source, gets updated every time a new broadcast message is generated at the source.

$CLUSTER_i$ is a set that contains the identities of hosts that, according to host i, are in the same cluster with i. This set can be updated when a message (of any kind, not necessarily a broadcast message) is received from another host j. If the cost bit in the message is 1, and j was a member of $CLUSTER_i$, then j is taken out of this set. Similarly, if the cost bit is 0, and j was not in the set, it is added. $CLUSTER_i$ is initialized to $\{i\}$, i.e., in the beginning each host assumes that it is in a cluster by itself. Of course, if there is some information to the contrary, then $CLUSTER_i$ can be initialized differently.

$CHILDREN_i$ is a set of all the children of host i in the host parent graph and is maintained by host i itself. Also, host i has an array $p_i[]$ such that its j-th element is the supposed parent of host j. Entry $p_i[i]$, of course, is the true parent of i, at all times. Arrays p_i are mutually updated on a periodic basis among the hosts in the same cluster only.

Finally, there is a static linear ordering imposed on all the hosts. The number assigned by the ordering to host i is denoted by $order(i)$.

The attachment procedure consists of a number of options that must be tried by the host, in the order indicated, until either a suitable new parent is found or all options are exhausted without success. In the latter case, the host waits a certain period of time[†] before initiating the same procedure again. If, however, a parent is found, a message is sent to it requesting inclusion in its *CHILDREN* set. If the acknowledgment to this message times out, the procedure is repeated to find another candidate with which the given host can communicate. The old parent, if any, is also notified of the change by an appropriate message.

The options, for each host i, are as follows (the new parent of i is denoted j).

I. For a host currently without a parent:

(1) Attach to a host in the same cluster that has a parent in a different cluster or no parent at all (a cluster leader), and a greater (according to relation $<$) INFO set. Thus j must satisfy the following conditions:

$$j \in CLUSTER_i$$
$$p_i[j] \notin CLUSTER_i$$
$$MAP_i[i] < MAP_i[j]$$

(2) Attach to a cluster leader in the same cluster with an "equal" INFO set and a greater static order number.

$$j \in CLUSTER_i$$
$$p_i[j] \notin CLUSTER_i$$
$$MAP_i[i] \simeq MAP_i[j]$$
$$order(i) < order(j)$$

(3) Attach to a host in a different cluster with a greater INFO set.

$$j \notin CLUSTER_i$$
$$MAP_i[i] < MAP_i[j]$$

† This time period is a parameter of the algorithm.

II. For a host with a parent in a different cluster:

(1) See Case I, Option 1.
(2) See Case I, Option 2.
(3) Attach to a host in a different cluster with an INFO set greater than that of the given host's current parent.

$$j \notin CLUSTER_i$$
$$MAP_i[p_i[i]] < MAP_i[j]$$

III. For a host with a parent in the same cluster:

(1) Attach to the ancestor that is a cluster leader in the same cluster

$$j \in CLUSTER_i$$
$$p_i[j] \notin CLUSTER_i$$
$$j = p_i[\quad \cdots \quad p_i[i]...]$$

Note: It is possible for a cycle to be created in the parent graph that consists of hosts in the same cluster. Such a situation will be detected when this option is tried. In that case the responsibility for breaking the cycle delegated to the host with the highest order number on the cycle. This host becomes a cluster leader and starts looking for a suitable parent outside the cluster.

The procedure is run at all hosts but the source. Note that in the very beginning of broadcast, the host parent graph is just a collection of hosts with no parent—child connections among them. In the process of broadcast those connections are established and changed as appropriate.

4.3. Properties of the Attachment Procedure.

In this subsection, we show that the attachment procedure constructs a host parent graph that induces a cluster dynamically, by adapting to constantly changing network topology and loads.

First of all, we need to show that the attachment procedure constructs a parent graph that is dynamically acyclic, i.e., has no persistent cycles, barring the case of partition. Since hosts accept broadcast messages only from their parents, no host's *INFO* set can be smaller than that of any of its descendants. Therefore the only way to form a cycle is for a host to attach to one of its own descendants with an "equal" *INFO* set. This can be the case only if that descendant is a cluster leader in the same cluster (case I, option 2 and case II, option 2). Hence, that cluster must contribute at least one cluster leader to the cycle. Of all cluster leaders from the cluster considered that are on the cycle, the one with the highest order number will eventually start looking for a new parent outside the cluster (case II, option 3). If it is successful, the cycle will be broken. If not, that will mean that it cannot communicate with any hosts that have greater *INFO* sets. By the transitivity assumption, none of the hosts on the cycles can communicate with such hosts either. Thus, even if the cycle is not broken, it does not matter in this particular case. For a cycle in the parent graph is undesirable only because no host on the cycle can get any new broadcast messages. But in the described situation this is the case even if there were no cycle. This indicates that a partition has occurred in the network. When the partition is repaired the cycle will be broken by the cluster leader.

Note that it is entirely possible that between the time the cycle is formed and the time it is broken, all the hosts that are on it become cluster neighbors (even if it was not the case at the creation of the cycle). If this happens, then it is no longer possible to break the cycle in the manner described in the previous paragraph. However, the provision in option 1 of case II of the attachment procedure will lead to breaking of the cycle. So unless there is a partition in the network, no cycle in the parent graph can be stable.

Options 1 and 2 of cases I and II work towards establishing a single cluster leader for each cluster, by making it a priority to look for a new parent within the cluster. Only when this fails, does the host look for a parent outside its cluster (option 3 of cases I and II). Option 1 of case III (for a host with a parent within the same cluster) attempts to establish a connection with a cluster leader directly, if it is not the case already. As a result, all hosts in the same cluster tend to organize into a single cluster tree node.

Option 3 of case I is for a host that has been unable to find a parent within its own cluster and, therefore, has to look elsewhere. This host, then, becomes a cluster leader.

Option 3 of case II is for a cluster leader that tries to improve its situation in terms of the delay with which it receives broadcast messages, by switching to a parent that has received more recent messages (with greater sequence numbers). This idea for reducing delays is similar to the one proposed by Awerbuch and Even [AwEv84]. In their work, however, it was applied in a different network setting (programmable severs, more restricted failure assumptions, disallowed acceptance of out-of-order messages).

Besides being an instrument for reducing delays, option 3 of case II can help a host to detect when its parent has become disconnected from it. For, in that case, the old parent's *INFO* set will fall behind those of other out-of-cluster hosts with which the given host can communicate. Note, however, that for hosts other than cluster leaders the attachment procedure does not provide an automatic way of detecting the failure or disconnection of the parent. Therefore, we need a separate provision to help detect this situation. One way to this would be to time out on a parent that fails to send messages such as the ones containing its *INFO* set and the identity of its own parent, which are being routinely exchanged by hosts in the same cluster. When this occurs, the host sets its parent pointer to NIL and goes through options 1 to 3 of case III.

4.4. Gap Filling.

The attachment procedure presented above is a way for the hosts participating in broadcast to adjust to component failures as well as to the changing loads in different parts of the network. The part of the protocol discussed here deals with compensating for lost broadcast messages (or filling gaps in *INFO* sets). Note that loss of messages can result not only from unreliable behavior of the communication subnetwork, but also from the workings of the attachment procedure. In particular, after a host has attached to a new parent, it may receive a broadcast message from its old parent (if the old parent never got the message requesting detachment from its former child), but in compliance with the restriction introduced above it is forced to discard it.

One type of gap filling actions takes place among parent graph neighbors. When a host attaches to a new parent, the parent examines its new child's *INFO* set and forwards to the child all those messages that the child is missing and that the parent has. When a host receives a gap filling message (a broadcast message with a sequence number less than the largest it has already seen), it forwards it to all those of its parent graph neighbors (its children and its parent) that according to its *MAP* do not have it. In addition to the above, every host periodically tries to fill its parent graph neighbors' gaps using messages that it has seen. This is done more frequently for the members of the same cluster and less frequently for the members of different clusters. The restriction that a host can accept broadcast messages only from its parent does not have to apply to gap filling messages because they do not alter the relationships among *INFO* sets.

Gap filling among parent graph neighbors only is not sufficient in that it fails, in some cases, to fill all the gaps or at least as many gaps as the current communication status of the network would allow. To illustrate consider the following example. Let there be three hosts in the network: s (the source), i, and j (in different clusters). The parent graph is shown in Figure 4.1. In it, s is the root, and i and j are its children. Suppose that a network partitioning occurs that leaves s isolated from the rest of the network. But i and j can still communicate with each other. Suppose, further, that three data messages (numbered 1, 2, and 3) were issued by s before the network partitioned; message number 2 has not reached node i, and message number 1 has not reached node j. Since neither $INFO_i < INFO_j$ nor $INFO_j < INFO_i$, hosts i and j will not be able to reconfigure themselves into a new parent graph until the partitioning is repaired. And thus, as i and j are not parent graph neighbors, they will not be able to fill each other's gap even though they can communicate with each other.

In order to deal with this kind of situations we introduce another type of gap filling actions. These take place periodically among hosts that are parent

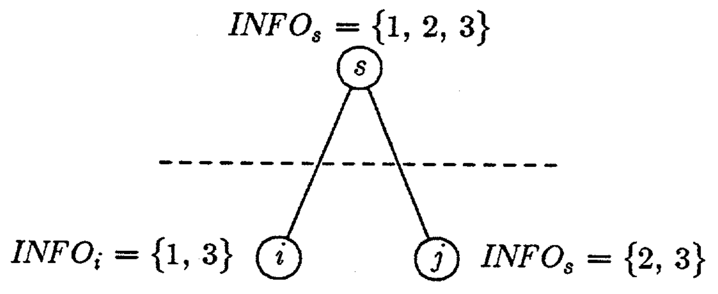

Figure 4.1.

graph neighbors. Frequency of this type of exchanges should be relatively low since these hosts belong to different clusters, and therefore the communication cost is high.

5. Conclusions.

We have presented a broadcast protocol for networks with nonprogrammable servers that appears to have good cost, delay, and reliability characteristics. We wish to emphasize, however, that our protocol is based on heuristics and, therefore, cannot be expected to perform optimally. The problem of efficient reliable broadcast in networks with nonprogrammable servers is a hard one, and solving it in a truly optimal way appears to be difficult.

There is one performance aspect that has not yet been discussed. It is the trade-off between reliability and cost-delay characteristics. That such should exist is no surprise. Reliability is understood to mean the ability of the algorithm to utilize as much as possible the communication opportunities presented by the network. Thus, if there is even a brief interval during which hosts h_1 and h_2 can communicate, and h_1 has a broadcast message that h_2 does not, a reliable protocol will detect this fact and have h_1 send this message (repeatedly if necessary) to h_2. But to achieve this, including the detection of the existence of the communication path between the two hosts, hosts have to exchange messages. The more frequently this is done, the more chance we will have to use the brief interval to deliver the message, and, at the same time, the more costly the algorithm will be.

In the algorithm presented here, these trade-offs are examplified by the frequency with which hosts enact *INFO* exchange, parent pointer exchange, and gap filling. These can be tuned according to specific cost-reliability requirements.

Throughout this paper we assumed that hosts have access to dynamic information concerning clustering. Note that even if such information is unavailable, but instead there is a static knowledge of clusters, the latter can be used in the algorithm, albeit with less satisfying performance results. Furthermore, if no cluster information at all is available, the algorithm still can be used, with the assumption that every host is in a separate cluster by itself, at any given moment.

A number of fairly obvious optimizations can be incorporated in the actual implementation of the algorithm. For instance, some control messages that are dispatched by the same host at about the same time can be piggybacked in one packet. As another example, *INFO* sets can be pruned of messages with sequence numbers 1 through n when it becomes known that all hosts have safely received them.

6. Acknowledgments.

We would like to thank Barbara Blaustein, Charles Kaufman, Sunil Sarin, and Oded Shmueli for their helpful comments.

7. Bibliography.

[AwEv84] Awerbuch, B., and S. Even, "Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network," *Proc. 3rd Symp. Principles of Distributed Computing*, 1984, pp. 278-281.

[BlKa85] Blaustein, B.T., and C.W. Kaufman, "Updating Replicated Data During Communications Failures," *Proc. 11th VLDB*, 1985, pp. 1-10.

[DaMe78] Dalal, Y.K., and R.M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM*, 1978, Vol. 21, Num. 12, pp. 1040-1048.

[Deme87] Demers, A., et.al., "Epidemic Algorithms for Replicated Database Management," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 1-12.

[Garc83] Garcia-Molina, H., et. al., "Data-Patch: Integrating Inconsistent Copies of a Database after a Partition," *Proc. 3rd Symp. Reliability in Distributed Software and Database Systems*, 1983.

[GaKo87] Garcia-Molina, H., and B. Kogan, "Achieving High Availability in Distributed Databases," *Proc. 3rd International Conf. Data Engineering*, 1987, pp. 430-440.

[McQu80] McQuillan, J.M., et. al., "The New Routing Algorithm for the

ARPANET," *IEEE Trans. on Communications*, 1980, Vol. COM-28, Num. 5, pp. 711-719.

[Peac80] Peacock, J.K., et. al., "Synchronization of Distributed Simulation Using Broadcast Algorithms," *Computer Networks*, 1980, Vol. 4, Num. 1, pp. 3-10.

[Rose80] Rosen, E.C., "The Updating Protocol of Arpanet's New Routing Algorithm," *Computer Networks*, 1980, Vol. 4, Num. 1, pp. 11-19.

[Sari85] Sarin, S.K., "Robust Application Design in Highly Available Distributed Databases," Computer Corporation of America, Technical Report, May 1985.

[SeAw83] Segall, A., and B. Awerbuch, "A Reliable Broadcast Protocol," *IEEE Trans. on Communications*, 1983, Vol. COM-31, Num. 7, pp. 895-901.

[Tane81] Tanenbaum, A.S., *Computer Networks*, Prentice Hall, Englewood Cliffs, N.J., 1981.

8. Appendix.

In this section, the proposed broadcast algorithm is formally specified. It consists of procedures that are all event driven, except INITIALIZE, which is run once at every host i at the very beginning of broadcast.

```
procedure INITIALIZE

begin
    $CHILDREN_i \leftarrow \emptyset$;
    $CLUSTER_i \leftarrow \emptyset$;
    for all hosts $j$ do
        $p_i[i] \leftarrow$ NIL;
    for all hosts $j$ do
        $MAP_i[j] \leftarrow \emptyset$;
    set timers NP1(i), EIC(i), EIN(i), GFC(i), GFN(i), EPP(i)
end
```

```
procedure UPDATE-CLUSTER
event receipt of an arbitrary message M from host $j$
```

/* (M).*cost* denotes the value of the cost bit in the format of message M. $CLUSTER_i$ is updated according to this value. */

```
begin
    if (M).cost $> 0$ and $j \in CLUSTER_i$ then
        $CLUSTER_i \leftarrow CLUSTER_i - \{j\}$;
    else if (M).cost $< 0$ and $j \notin CLUSTER_i$ then
        $CLUSTER_i \leftarrow CLUSTER_i \cup \{j\}$;
end
```

```
procedure PARENT-ALIVE
event receipt of an arbitrary message M from host $j$

/* Whenever a message is received from the parent the timer
gets reset. So when it expires, that indicates that the parents
is out of reach. */

begin
    if $j = p_i[i]$ and $j \in CLUSTER_i$ then
        reset timer $\mathbf{PA}(i)$
end
```

```
procedure NEW-PARENT
event: expiration of timers NP1(i) or NP2(i) or PA(i)


/* This is the attachment procedure. It is initiated either after a regular
predetermined interval (timer NP1(i)), or if a newly selected parent
fails to respond (timer NP2(i)), or when an in-cluster parent dies
(timer PA(i)). */

begin
    if PA(i) expired then $p_i[i] = \text{NIL}$;
    if $p_i[i] = \text{NIL}$ then $h = p_i[i]$
    else $h = p_i[p_i[i]]$;
    if $p_i[i] = \text{NIL}$ or $p_i[i] \notin CLUSTER_i$ then
    begin
        if there is a host $j$ s.t. $j \in CLUSTER_i$
                            and $p_i[j] \notin CLUSTER_i$
                            and $MAP_i[i] < MAP_i[j]$ then
                  $p_i[i] \leftarrow j$;
        else if there is a host $j$ s.t. $j \in CLUSTER_i$ and $p_i[j] \notin CLUSTER_i$
                            and $MAP_i[i] \simeq MAP_i[j]$ and $order(i) < order(j)$ then
            $p_i[i] \leftarrow j$;
        else if there is a host $j$ s.t. $MAP_i[h] < MAP_i[j]$ then
            $p_i[i] \leftarrow j$;
        else if there is a host $j$ s.t. $MAP_i[h] \simeq MAP_i[j]$ and $order(i) < order(j)$ then
            $p_i[i] \leftarrow j$;
        send DECLARE to $p_i$;
        reset timer NP(i)
    end
    else if
    begin
        repeat $anscestor \leftarrow p_i[p_i[i]]$
        until $anscestor = i$ or $p_i[anscestor] \notin CLUSTER_i$;
        if $anscestor = i$ then
            if $order(i) = \max_{j \text{ on cycle}} order(j)$ then $p_i[i] \leftarrow \text{NIL}$
        else
        begin
            send DECLARE to $p_i$;
            reset timer NP(i)
        end
    end
end;
```

```
procedure NEW-CHILD
event receipt of DECLARE from host $j$


/* Host $j$ indicates that it wants to attach. It is included in
$CHILDREN_i$. Also, the gaps that it might have are filled. */

begin
    $CHILDREN_i \leftarrow CHILDREN_i \cup \{j\}$;
    for all $r$ s.t. $r \in MAP_i[i]$ and $r \notin MAP_i[j]$ do
        send message $r$ to host $j$;
    send ADOPT to host $j$
end
```

```
procedure DETACH-A-CHILD
event receipt of DETACH from host $j$


begin
    $CHILDREN_i \leftarrow CHILDREN_i - \{j\}$
end
```

```
procedure      NEW-PARENT-ACKNOWLEDGED
event receipt of ADOPT from host $j$

/* The new parent has acknowledged adoption. If this does not happen for
a certain period of time (timer NP2($i$)), NEW-PARENT is invoked again.
If ADOPT comes from a host which is not the parent, that host must be the old
parent that never got DETACH. In that case DETACH is sent again. */

begin
    if $j = p_i[i]$ then
    begin
        freeze timer NP2($i$);
        reset timer NP1($i$)
        if $j \in CLUSTER_i$ then reset timer PA($i$)
    end
    else send DETACH to $j$
end
```

```
procedure EXCHANGE-INFO-CLUSTER
event expiration of timer EIC($i$)

/* For exchange of INFO sets among members of the same cluster. */

begin
    for all $j \in CLUSTER_j$ do
        send $MAP_i[i]$ to host $j$;
    reset timer EIC($i$)
end
```

```
**procedure** EXCHANGE-INFO-NONCLUSTER
**event** expiration of timer **EIN**($i$)

/*For exchange of *INFO* sets among hosts from different cluster. Executed
with low frequency than the previous procedure. */

**begin**
    **for** all $j \notin CLUSTER_i$ **do**
        send $MAP_i[i]$ to host $j$;
    reset timer **EIN**($i$)
**end**
```

```
**procedure** UPDATE-MAP
**event** receipt of $MAP_j[j]$ from host $j$

/* *MAP* is update after receiving *INFO* set from another host. */

**begin**
    $MAP_i[j] \leftarrow MAP_j[j]$;
    **if** $j \in CHILDREN_i$ **then**
        **for** all $r$ s.t. $r \notin MAP_i[j]$ **and** $r \in MAP_i[i]$ **do**
            send message $r$ to host $j$
**end**
```

```
**procedure** EXCHANGE-PARENT-POINTERS
**event** expiration of timer **EPP**($i$)

/* Tell your cluster neighbors who your parent is. */

**begin**
    **for** all $j \in CLUSTER_j$ **do**
        send $p_i[i]$ to host $j$;
    reset timer **EPP**($i$)
**end**
```

```
procedure UPDATE-PARENT-POINTERS
event receipt of $p_j[j]$ from host $j$


/* Update the $p$ array. */

begin
        $p_i[j] \leftarrow p_j[j]$
end
```

```
procedure GAP-FILLING-CLUSTER
event expiration of timer GFC($i$)


/* Fill gaps of the parent graph neighbors within the same cluster. */

begin
        for all $j$ s.t. $j \in CLUSTER_i$ and ($j = p_i[i]$ or $j \in CHILDREN_i$)
                for all $r$ s.t. $r \in MAP_i[i]$ and $r \notin MAP_i[j]$ do
                        send $r$ to $j$;
        reset timer GFC($i$)
end
```

```
procedure GAP-FILLING-NONCLUSTER
event expiration of timer GFN($i$)


/* Fill gaps of the parent graph neighbors that are in a different cluster. */

begin
        for all $j$ s.t. $j \notin CLUSTER_i$ and ($j = p_i[i]$ or $j \in CHILDREN_i$)
                for all $r$ s.t. $r \in MAP_i[i]$ and $r \notin MAP_i[j]$ do
                        send $r$ to $j$;
        reset timer GFN($i$)
end
```

```
procedure PROPAGATION
event: receipt of broadcast message $r$ from host $j$

/* A broad message has been received. It is discarded if it comes from
a host other than the parent and has a sequence number greater than any
seen so far. Otherwise, it is accepted and forwarded to parent graph
neighbors. */

begin
    $MAP_i[j] \leftarrow MAP_i[j] \cup \{r\}$;
    if $r \in MAP_i[i]$ then
        discard message $r$
    else if $j \neq p_i[i]$ and $r > \max_{q \in MAP_i[i]} (q)$ then
    begin
        discard message $r$;
        send DETACH to host $j$
    end
    else
    begin
        store message $r$;
        $MAP_i[i] \leftarrow MAP_i[i] \cup \{r\}$;
        for all $k$ s.t. $k \in CHILDREN_i \cup \{p_i[i]\} - \{j\}$ and $r \notin MAP_i[k]$ do
            send message $r$ to host $k$
    end
end
```