AN IMPLEMENTATION OF RELIABLE BROADCAST USING AN UNRELIABLE MULTICAST FACILITY

Hector Garcia-Molina
Boris Kogan

CS-TR-170-88

August 1988

AN IMPLEMENTATION OF RELIABLE BROADCAST USING AN UNRELIABLE MULTICAST FACILITY†

Hector Garcia-Molina
Boris Kogan

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Some computer communication networks provide a multicast facility for their users. This means that a host can hand a message to its server with more than one destination address. The network then tries to deliver the message to all specified destinations in an efficient way. However, it does not guarantee a reliable delivery. In this paper the problem of reliable broadcast on top of such a network is considered. The solution proposed makes use of the multicast facility to achieve efficiency.

August 3, 1988

AN IMPLEMENTATION OF RELIABLE BROADCAST USING AN UNRELIABLE MULTICAST FACILITY[†]

Hector Garcia-Molina
Boris Kogan

Department of Computer Science
Princeton University
Princeton, NJ 08544

1. Introduction.

In a *reliable broadcast* a sequence of messages must be transmitted to a set of computers over a communication network. All messages must be eventually delivered to all participating computers. In addition, it is essential that the mechanism designed to achieve this goal be efficient.

In this paper we consider the problem of reliable broadcast in a point-to-point asynchronous network. Such a network consists of *host* computers and a *communication subnetwork*. The latter, in turn, is a collection of *switches* (special purpose computers that have the ability to store and forward messages), interconnected by point-to-point bidirectional communication *links*. The subnetwork is unreliable, i.e., a message is never guaranteed to be delivered in a finite interval of time. Broadcast is to be implemented among the hosts of the network.

There has been a significant amount of research done in the area of reliable broadcast in such networks [AwEv84, Deme87, Garc87, McQu80, Peac80, Rose80, SeAw83]. What makes the present work different are three requirements and one assumption that we have:

(*a*) We require "epidemic" propagation of messages. This means that a broadcast message must be delivered to a destination as fast as possible. A destination will not get a message only if there is no path to a node with a copy.

(*b*) We require efficient recovery from network failures. The network should not be flooded with retransmission attempts and control messages (e.g., acknowledgments) when a partition is repaired.

(*c*) We require that the broadcast algorithm not involve programming the network switches. As a consequence, the algorithm has to be implemented at the host level.

(*d*) We assume that the network provides an efficient but unreliable multicast mechanism for use by hosts. This mechanism can be used to build a reliable broadcast.[‡]

[†] 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.

[‡] Reliable broadcast, which is the subject of this paper, can be easily generalized to reliable multicast. However, to prevent any confusion between the multicast facility provided by the network and the algorithm that we intend to build on top of that facility, we will use the word "multicast" to refer to the former and "broadcast," to the latter.

In Section 2.1 we justify requirements (a) and (b) by describing environments where they arise. In Section 2.2 we argue that non-programmable switches (requirement (c)) are a reality, but that it is reasonable to expect an unreliable multicast facility (assumption (d)). Having done this, we will proceed to describe our algorithm, starting in Section 3.

2. System Setting.

In this section we focus on the assumptions about the kind of distributed systems we consider and on the requirements we would like our algorithm to satisfy.

2.1. The Environment.

We require here, unlike other approaches to the problem, that messages be propagated as fast as possible, even if this means delivering them out of order. So if message i cannot be delivered due to a failure, but for some reason message $i+1$ can be, then the application should get $i+1$. Our justification is that there are applications that need this type of service, so the broadcast should provide it. If other applications require ordered delivery, they can easily delay out of order messages. Thus, our broadcast will provide the most flexible and general type of delivery, that can be tailored to satisfy all applications. To illustrate applications that need out of order delivery, consider the problem of managing highly available replicated databases. There are some approaches to this problem that sacrifice serializability of transaction execution for the sake of unrestricted data availability in the face of communication failures (e.g., Data-Patch [Garc83], log transformation [BlKa85, Sari86]). This means that arbitrary transactions can run independently at distributed sites, even when the network is partitioned. To maintain mutual consistency of replicas, updates are propagated to remote sites whenever possible (hence the need for reliable broadcast). In such a system it would no longer be crucial that a site install remote updates in the same order they were generated. Thus, out of order messages are acceptable and allow the database to reflect the latest information available.

As another example, consider a real-time distributed application which requires a reliable efficient broadcast. Broadcast messages sent in such a system might have different priorities. A high priority message should not be delayed just because it is out of order. For instance, suppose that in a military command and control application an update on the maneuvers planned for next month was broadcast by the HQ. It was then followed by a red alert message. Even if some hosts have not yet received the maneuvers update, it seems unreasonable to require that the red alert message be withheld from them. Note that the assumption of out-of-order messages implies that our broadcast is nonatomic.

In our approach, we require that recovery from network partitions be efficient. In many systems partitions can be quite protracted. In fact, rather than being a failure condition, they can even be a normal functional mode of the system. For example, in the UUCP network computers operate autonomously, in isolation, most of the time. Once in a while they dial up one another to perform message exchange. A similar situation exists in a system which uses satellites for communications among geographically dispersed computer nodes (or clusters of such). Communication between a pair of nodes (clusters) is possible only for a relatively brief period of time when the satellite is in the right position within its orbit allowing it to have radio contact with both nodes (clusters).

While a partition lasts, the source can continue generating broadcast messages. The longer the time till reconnection, the more messages will have accumulated for delivery to those hosts that were disconnected from the source of broadcast. If the system can stay connected for only a short time (as in the examples of the previous paragraph), it is imperative that the (low bandwidth) link (links) providing the connection be utilized as efficiently as possible. Thus, the message traffic needed to transmit the broadcast messages across the connection should be kept to a minimum. Note that this is true even for systems in which partitions occur only rarely. For it is undesirable to flood the newly repaired link (links) with high message traffic because the resulting congestion can bring this link (links) down, thereby partitioning the network again and breeding a vicious circle.

In this paper we only consider the problem of single-source broadcast. However, multiple sources can be accommodated by replicating the same protocol for every source.

2.2. The Network.

Hosts, switches, and links were already mentioned in the introduction as the main components of the network. We assume that every host is connected to a switch, called this host's *server*. A switch can be the server for more than one host, just as there can be switches with no hosts attached to them. Figure 2.1 shows an example network. The squares represent the hosts h_1 through h_5, while the circles represent the switches. The eleven communication links are labeled l_1 through l_{11}. In all future references to this example we assume that host h_1 is the source of broadcast.

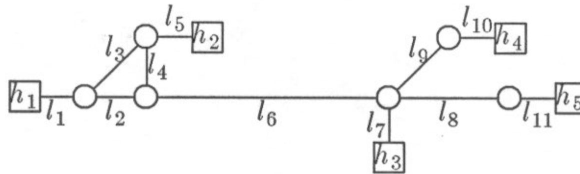

Figure 2.1.

Note that in real networks (e.g., the UUCP network) a server, rather than being a separate piece of hardware connected to the corresponding host by a link, can be a process running on that host. In such an environment, the host computer is directly connected to the communications subnetwork. For the purposes of this paper, however, this distinction is not essential, and the model we present can be a valid abstraction for any type of point-to-point network.

Failure properties of the described network are characterized by the following set of assumptions. Sites (hosts and switches) can exibit *omission* failures, i.e., they can fail to send a message when prescribed to do so. Note that omission failures cover the case of site crashes. It is assumed, however, that no *malicious* failures occur, i.e., messages are not altered or generated when they are not supposed to in order to disrupt the correct functioning of the system.

Communication links can fail at any time and can come back up at any time. Messages can be lost, and delays can be arbitrarily long. Note that the loss of a message due to a buffer overflow can be more easily modeled by a link failure than a site failure (when a message arrives at a site but fails to be processed, it is as if it never was delivered).

We assume that the communication subnetwork provides a multicast facility to the hosts. This means that a host can hand to its server a message with a list of more than one address. It is then the responsibility of the communications subnetwork to expedite the (efficient) delivery of the message to all the addresses listed. The multicast mechanism, however, gives no guarantees that every single message will successfully reach every specified destination. (Such a mechanism could be implemented, for example, on the basis of the ERPF algorithm suggested by Dalal and Metcalfe [DaMe78], which is efficient but not reliable).

The solution to the problem of reliable broadcast proposed here will consist of the existing multicast mechanism in combination with an algorithm, *to be implemented on the hosts*, whose goal is to fill "holes" in the potentially unreliable multicast mechanism. The justifications for such an approach, as opposed to implementing a reliable broadcast on the switches, are twofold. First, in many instances the users of networks will not have administrative control over the switches and, thus, will not be able to create or modify the code that the switches run. For instance, users of ARPANET do not have access to the code running on the IMPs. The only alternative in this case is to program the hosts. Second, even when the switches are programmable, it may be less costly to build a package of reliability provisions on top of an already existing efficient (but unreliable) mechanism than to design a reliable broadcast from scratch (even though the latter option may result in a more efficient broadcast mechanism). Following the first justification we assume that the switches are indeed nonprogrammable.

Why is it reasonable to assume that an unreliable multicast facility is available? Although many networks do not currently provide a multicast facility, it is the only way to achieve trully efficient broadcast. For example, in the network of Figure 2.1, suppose that a message is to be sent from h_1 to h_3, h_4, h_5. A multicast facility makes it possible to send a single copy of the message over link l_6. Thus, since our goal is to study the most efficient way to perform reliable broadcast, it is reasonable to assume that adequate facilities are provided. (There have been concrete proposals for a multicast facility in the DARPA Internet [Agui84, DeCh85]. Moreover, one of them [DeCh85] has been implemented in parts of the Internet.)

But why not assume that a *reliable* multicast facility is provided by the network (and then there would be no problem for us to solve)? The answer is that this would not be a realistic assumption. First of all, reliable multicast is substantially more complex than the unreliable version, so it is difficult to envision it as a network service. Second, applications would probably implement end-to-end checks anyway, so much of the effort expended by the reliable multicast would be duplicated. Third, as we will see, there are several conflicting goals in reliable multicast (e.g., efficiency, reliability). It would be difficult for a general purpose network layer to give the right balance for all applications. If the reliability provisions are implemented at the application level, then the algorithm can be easily tuned.

3. Basic Ideas.

In Section 1 we mentioned some of the work done on the problem of reliable broadcast in asynchronous point-to-point networks. However, as far as we can tell, our algorithm is the first to consider non-programmable switches (a very realistic assumption we believe) and to take advantage of an underlying multicast facility, when one exists.

A simple and obvious way to perform a reliable broadcast of a message is to send a separately addressed copy of it to every host involved. The process is repeated until an acknowledgement is received from each destination. The algorithm we present here

has an improved performance compared to the basic algorithm. The former has the following features that favorably set it apart from the latter.

(1) *Use of Multicast.* Our algorithm takes advantage of the efficient multicast facility whenever possible (see the example at the end of Section 2.2). The multicast facility can be used not only for regular message delivery but also at recovery time if an entire subgroup of hosts have missed a message.

(2) *Efficient Failure Recovery.* Some network failures can leave a group of nodes isolated. (Such a group is called a *partition.*) The basic algorithm would recover from the failure on a host by host basis. In our algorithm, however, the isolated group will dynamically select a coordinator to handle the recovery more efficiently. For instance, suppose that link l_6 fails. The isolated hosts will form a tree, with h_3 as root and h_4 and h_5 as its children. Only h_3 would probe for host h_1 (and maybe for h_2 too) in order to detect the end of the partition. When h_1 is finally reached, h_3 will hand it the list of hosts in its group so that the transmission of the missed broadcast messages can be done with the efficient multicast facility. Selecting a coordinator is performed by a procedure that is itself fault-tolerant in regard to network partitioning, i.e., when partitions split or fuse with other partitions it makes sure that there always remains exactly one coordinator for every partition. This procedures makes use of *priority lists* — a central concept in our algorithm.

(3) *Stream Communication.* Broadcast applications usually operate on streams of many messages rather than on a few isolated messages (e.g., broadcast of updates in replicated databases). Our algorithm will take advantage of this fact. Broadcast messages will be numbered and these numbers will be used to efficiently detect gaps in the stream and coordinate retransmissions.

(4) *Epidemic Propagation.* Consider the case where the server for h_1 fails just after it has sent a message m_1 to h_2 (along l_3) but before it sends m_1 along l_2 to the rest of the hosts. With a simple algorithm, m_1 would not be received by the remaining hosts until h_1 and its server recover. With our algorithm, on the other hand, m_1 will propagate out of h_2 as long as there is a communication path to other nodes. (In our case, the available path is l_5, l_4, l_6, ...) Thus, messages go out as if they were a "flu epidemic" along any possible "contagion" paths (as in [Deme87]). Epidemic propagation can also be viewed as sharing the responsibility for reliable broadcast among all hosts as opposed to limiting it only to the source, as is the case with the basic algorithm.

(5) *Use of Topological Information.* If information on the network topology is available, our mechanism can take advantage of it to a certain extent. This is done through priority lists. They specify the order in which hosts should probe the network after a failure is detected. For example, when l_6 fails, it is better to have host h_3 be the root of the isolated subgroup. (If h_4 were the root, with h_3 and h_5 as children, the probe messages sent by h_4 would take a longer route to h_1. Similarly, parent-child communications required on the tree would use indirect paths.) The priority lists can ensure that a good tree structure is obtained after a failure. (For instance, h_4 will be forced to ask h_3 if h_3 can become a root. Only if h_3 does not respond would h_4 attempt to become a root.) The priority lists will also tell selected root nodes in what order to probe outside the group for a connection. In our example, a connection from h_3 to h_1 is more desirable than one to h_2 (recall that h_1 is the source of the broadcast). However, if h_1 is unreachable, h_3 should also probe h_2 to try to form a larger tree (and to get any messages the latter may

have; see item (4) above).

4. The Algorithm.

The basic mode of operation for the proposed broadcast mechanism is the posting by the source of multiply addressed data messages to be delivered to all participating hosts, combined with a distributed *redelivery algorithm* that insures that those messages that were lost in transit do eventually arrive at all intended destinations. That the redelivery algorithm should be distributed reflects the notion of shared responsibility for reliable broadcast discussed in the previous section.

Lost messages result from two types of communication failures: *transient* and *persistent.* Transient failures cause isolated cases of lost messages only (e.g., because of a buffer overflow). Persistent failures result in network partitions, when one or more group of hosts get cut off from the rest of the network that contains the source of broadcast. In such a case, these hosts will miss all the messages generated by the source from the time the partition occurs until it is repaired.

We assume that all data messages are sequence numbered at the source. Moreover, there is a predetermined time parameter δ ($\delta > 0$) such that, when δ units of time elapse without a data message being generated at the source, the source broadcasts a *null message.* The above two provisions together simplify the detection of gaps in the stream of data messages received by any given host. Namely, if the host has not received either a data or a null message from the source for over $m\delta$ units of time (where $m \geq 1$ is a parameter introduced to allow for possible discrepancy in delays for two consecutive messages), a failure detection is declared and the redelivery algorithm is initiated at that host. Similarly, the host sounds the alarm when it receives a data message whose sequence number exceeds that of the immediately preceding data message by more than one. The difference between the above two situations is that while the latter always signals a transient failure, the former may also be a result of a persistent failure — network partitioning.

Recovering from a persistent failure is more costly than recovering from a transient one because it requires a redelivery of multiple messages to multiple recipients. Therefore efficient recovery from persistent failures is more essential for good performance of our algorithm. For that reason, as well as for reasons of methodology, we concentrate first on this aspect of the algorithm. Later on it will be shown, using essentially the same procedure as the one outlined in the rest of this section, how to recover from transient failures.

4.1. Priority Lists.

At the heart of the proposed algorithm is a set of structures called *priority lists.* $PL(h_i)$, the priority list of host h_i, is an ordered set of hosts that should be probed by h_i as possible coordinators when h_i discovers that it is missing some data messages. We assume that $PL(h_i) = \emptyset$ whenever h_i is the source, since in that case h_i can never be missing any messages. Otherwise, suppose $PL(h_i) = \{h_{i_1}, h_{i_2}, ..., h_{i_k}\}$. Then h_i tries to contact the hosts starting from h_{i_1} through h_{i_k} until one is found, say h_j, $1 \leq j \leq i_k$, that it can communicate with. If h_j has all the messages that h_i is missing, it sends them to h_i. Otherwise h_j starts probing its own priority list (if it has not been doing so already) on behalf of h_i and any other hosts that may have contacted it for missing messages. In this way a *coordinator tree* is formed. When h_j gets in touch with some other host from its priority list, it delegates to that host the responsibility of the (temporary) coordinator for h_j and all its descendants in the coordinator tree. This process

is halted when the new (temporary) coordinator either can satisfy the demands of all of its descendants for missing messages or is unable to contact any host on its priority list. When the latter is true the host at the root of the coordinator tree remains the coordinator for the group (tree) for the duration of the partition. Periodically, it sends I-AM-COORDINATOR messages to all its descendants in the tree. This helps the descendants to detect when the coordinator gets cut-off from them or goes down, in which case the election process is repeated. When the partition is repaired, the root is finally able to contact one or more hosts that can supply the missing messages. These hosts, in turn, send the messages to the members of the group using the multicast facility. Note that while a partition lasts, there is nothing that should prevent us to allow message redelivery within the partition not only down the coordinator tree but also up the tree. As a result, if at least one host within a partition has a certain broadcast message, all hosts within that partion will eventually have the same message even before reconnection takes place, a situation quite unlike the basic algorithm for reliable broadcast discussed in Section 3. For a detailed specification of the redelivery algorithm see Appendix I.

The same mechanism can be applied for transient failure recovery, as was mentioned above. The host that has discovered a gap in its broadcast message stream (although there is no network partition at the time) can get in touch with one of the hosts on its priority list and request it to retransmit the message in question. If that host does not have the message, it goes to its own priority list to make a request on behalf of the original host. Note that in the case of a transient failure it is unlikely that too many hosts in a row will be missing the same broadcast message, therefore the redelivery should not take long.

4.2. Correctness.

For the algorithm to operate correctly the lists have to satisfy certain conditions. These are *joint completeness* and *acyclicity*. The priority lists for the set of hosts $h_1, h_2, \ldots, h_n$ are jointly complete if for any i and j such that $1 \leq i, j \leq n$, either $h_i \in PL(h_j)$ or $h_j \in PL(h_i)$. Let R be a relation defined as follows: $h_i R h_j$ iff $h_i \in PL(h_j)$. Then a collection of priority lists is said to be acyclic if there is no i, $1 \leq i \leq n$, such that $h_i R^* h_i$, where R^* is the transitive closure of R.[†]

We define correctness as the ability of an algorithm to ensure that in any sufficiently long lasting partition there will be exactly one coordinator. (Note that this is equivalent to having a single coordinator tree involving all hosts in the partition.) We need at least one coordinator to assume the responsibility for recovery. On the other hand, we do not want more than one coordinator because otherwise recovery will not be efficient.

Theorem 4.1. Let $G = \{h_1, \ldots, h_k\}$ be the set of hosts that form a partition that excludes the source of broadcast. Then all hosts in G will eventually configure themselves into a coordinator tree if and only if the priority list assignment is acyclic and jointly complete (provided that the partition is stable for a sufficiently long time).

Proof. See Appendix II.

To illustrate the use of priority lists and their properties, we show one possible way to assign priority lists for the case of the sample network of Figure 2.1. Note that we may or may not know the topology of the network. Let

---
[†] Note that for jointly complete and acyclic lists R is a total order.

$$PL(h_1) = \varnothing,$$
$$PL(h_2) = \{h_1\},$$
$$PL(h_3) = \{h_1, h_2\},$$
$$PL(h_4) = \{h_3, h_5, h_2, h_1\},$$
$$PL(h_5) = \{h_1, h_3, h_2\}.$$

It is fairly straightforward to verify that the lists are jointly complete and acyclic.

Suppose that a partition occurs that leaves the source, h_1, isolated from the rest of the hosts. Detecting the interruption in the stream of messages (by timing out after $m\delta$ units of time since the last data or null message), host h_5 will try to contact the first host on its priority list, h_1. Failing to do that, it will go to the next host, h_3. Since it is in the same partition with h_3, h_5 will be successful this time around, and will become a child of h_3 in the coordinator tree. Similarly, host h_4 will choose h_3 to be its parent in the coordinator tree. Being contacted by hosts h_5 and h_4 (or possibly even before that), host h_3 will start its own search for a potential coordinator. Failing to reach h_1, it will settle for h_2, which is in the same partition with it. Finally, h_2 will not be able to pass on the coordinating responsibility to any other host because the only host on its priority list, h_1, is in a different partition. Thus h_2 becomes the coordinator for its partition. The resulting coordinator tree is shown in Figure 4.1.

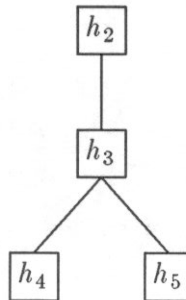

Figure 4.1.

Note that the tree can adjust when the partition boundaries in the network change. For example, if hosts h_5 and h_4 get cut off from h_3 and h_2, thereby forming their own partition, h_4 will attach to h_5 (h_5 is next on its priority list). Thus a new tree is formed, and h_5 becomes the coordinator for this new partition.

Consider the effect on the redelivery algorithm in the case when the lists are not jointly complete or acyclic. Suppose that h_5 is not on the priority list of h_4 (that would make the lists not jointly complete). Then h_4 never attempts to contact h_5 and ask it to be the coordinator. As a result, both hosts remain on their own, despite the fact that they are in the same partition, and the redelivery of data messages will take place separately to each of them, which is inefficient.

Suppose that h_5 is returned to $PL(h_4)$, but h_4 is now inserted into $PL(h_2)$ after h_1. Now the lists are jointly complete but not acyclic. When the first partition occurs (the one that isolates h_1 from the rest of the hosts), h_2, after unsuccessful attempts at comunicating with h_1, contacts host h_4, which is next on its list. Now a cycle is formed in the coordinator tree. As a result, every host in the partition thinks that somebody else is the coordinator, and no host will be probing h_1 to see when it is reconnected to

the rest of the system.

There is an obvious algorithm for constructing collections of priority lists that are acyclic and jointly complete. Start with any host. Include in its priority list all the rest of hosts (in any order). For each host h_i after that take the priority list of the previous host minus h_i (the order of the list can be changed arbitrarily). Thus with each step the size of the list is decreased by one. Complete the algorithm with the source, in which case it is guaranteed to get assigned $\emptyset$. This algorithm runs in $O(n^2)$ time[†].

The reader may have noticed that the relative order of hosts on a priority list did not play any role in terms of the correctness of the redelivery algorithm. In fact, the algorithm for constructing priority lists above does not even control that order. In other words, until now priority lists were treated as if they were sets rather than ordered sets. When it comes to performance, however, the order becomes important. In the next section we will show how to control the performance of the redelivery algorithm by properly ordering the lists.

5. Using Priority Lists to Encode Topological Information.

Given different possible priority list assignments that satisfy the properties of acyclicity and joint completeness, are there any reasons to favor one over another? If we assume that nothing is known about the topology of the network, then the answer to the question we have posed is no. However, if topology information is available (perhaps even including the cost of using particular communication links), the answer will be yes.

To illustrate consider once again the example of Figure 2.1, but with the following addition. Suppose that link l_6 has a small bandwidth, while all other links have substantially higher bandwidths. Thus, the cost of transmitting over l_6 is higher than over other links. Consider two alternative assignments of priority lists, A and B.

Assignment A:

$$PL(h_{1)} = \emptyset$$

$$PL(h_2) = \{h_1\}$$

$$PL(h_3) = \{h_1,\ h_2\}$$

$$PL(h_4) = \{h_3,\ h_1,\ h_2\}$$

$$PL(h_5) = \{h_3,\ h_4,\ h_1,\ h_2\}$$

Assignment B:

$$PL(h_{1)} = \emptyset$$

$$PL(h_2) = \{h_1,\ h_4,\ h_5\}$$

$$PL(h_3) = \{h_1,\ h_2,\ h_4,\ h_5\}$$

$$PL(h_4) = \{h_1,\ h_5\}$$

$$PL(h_5) = \{h_1\}$$

[†] Note that the running time of the algorithm is not of great importance because the assignment is static and has to be done only once.

Suppose link l_1 goes down. This will leave the source isolated from the rest of the hosts. The redelivery algorithm will be started in the resulting partition, and nodes h_2 through h_5 will organize themselves into a coordinator tree. Using assignment A, they will arrive at the tree shown in Figure 4.1. Assignment B, on the other hand, will result in the tree of Figure 5.1.†

Figure 5.1.

Let us compare the two trees from the point of view of relative cost of recovery that they entail. First, we notice that the construction of the second tree would require more message transmissions over the expensive link (l_6) than would the first. For there are two child—parent connections in the tree of Figure 5.1 ($h_3 \rightarrow h_2$ and $h_2 \rightarrow h_4$) that traverse l_6, compared to only one in the tree of Figure 4.1. Furthermore, assignment A will result in electing h_2 as coordinator, and assignment B in electing h_5. In both cases, the coordinator will be responsible for periodically probing the source, h_1, to determine when the source is reconnected so that the retransmission of messages can take place. This probing can be done less expensively when h_2 acts as coordinator since h_2 is much "closer" to h_1 than h_5 is.

Thus, in the example above, one priority list assignment (A) turns out to be more cost efficient than the other (B). Therefore, we would like to have an algorithm that produces not just a correct but also an efficcient assignment for a given network. Unfortunately, the problem of finding an optimal assignment given the topology of the network appears to be very difficult. In fact, it is not even quite clear how to define precisely what constitutes an optimal solution. Among the factors influencing the solution that were not mentioned above are (*i*) possible overloading of servers when too many hosts are trying to attach to the same parent (the parent's server becomes overloaded); (*ii*) the necessity to consider all possible ways in which the network can partition, when optimizing the priority list assignment (the knowledge of probability with which each partition can occur becomes essential here because one would like to have hosts that are more likely to end up in the same partition with the given host to be near the beginning of the priority list for that host); (*iii*) the possibility that some links may fail within a partition, but without causing the partition to split further, thereby possibly

† We have implicitly assumed that if hosts x and y are in the same partition, then x will not fail to respond to probing by y. Thus, if x precedes all other hosts in that partition on the priority list of y, x will become y's parent. In reality, due to a transient failure, this might not always be the case (then y would have start probing the next host on its priority list). We disregard this not very likely possibility for now (given the assumption of low frequency of transient failures).

increasing the cost of communication between some pairs of hosts.

In the rest of this section, we describe a heuristic that produces good priority list assignments, when supplied with the full information on the topology of the network and bandwidths of (or costs associated with) every link.

5.1. A Heuristic Algorithm.

A *communication distance* between a pair of hosts is defined as the sum of the costs of all links on the shortest path between them. For example, if the cost of every link in Figure 2.1 were 1, then the communication distance between hosts h_4 and h_5 would equal 4.

The priority lists are assigned to hosts one by one, i.e., the algorithm does not start assigning a list to the next host before the previous host's list is completed. The hosts are considered for assignments in the decreasing order of their distance from the source, with ties broken arbitrarily. Thus, we start with the most remote host and finish with the source (which gets an $\emptyset$ assignment). The hosts to be included in the lists are determined as in the assignment algorithm of Section 4, i.e., all hosts are included in the first list; from every succeding list we exclude only those hosts that have already received their priority lists. That will guarantee that the resulting lists are acyclic and jointly complete. In this new algorithm, however, we should also concern ourselves with the order of hosts within every list. That order is determined according to the proximity of the hosts on the list to the owner of the list, with the closest host being in the first position. Ties are broken in favor of the host that is at the shortest distance from the source. We call this the Shortest-Distance-First algorithm, or SDF (see Figure 5.2).

Two of the criteria of the efficient failure recovery (which were used to compare two alternative priority list assignments in the beginning of this section) are the cost of the coordinator tree, defined as the sum of communication distances between every child—parent pair, and the proximity of the coordinator to the source of broadcast. The SDF algorithm exibits good (although suboptimal behavior) with respect to both criteria. Note, for example, that assignment A above can be produced using this algorithm.

It is intuitively clear that the cost of the tree is kept low by SDF because hosts at a close distance are favored for parent selection over those that are far away. Similarly, a host closest to the source is more likely to become the coordinator. In fact, when communication distances are static, the coordinator is always the closest host to the source among all the hosts in the same partition. This is so because because the closest host to the source will be the last among the hosts from its partition to be assigned a priority list by SDF. Therefore, none of those hosts will appear on its list, which implies that it has to be the root of the coordinator tree.

Priority lists produced by SDF do not have any control over the number of children that a node in the coordinator tree may have. Thus, there are no provisions in SDF, as it is presented, to prevent potential overloading of servers during the execution of the redelivery algorithm. Note that SDF can be extended to take this problem into consideration by modifying the defenition of the coordinator tree cost so that it includes additional penalties for a node with too many children. However, such enhancements are not considered here.

The heurisitic algorithm we have presented works best when communication distances are static. However, the assumption of static distances may not be true to

begin
    $H \leftarrow \{h_1, \ldots, h_n\}$;
    for $i = 1$ **to** n **do**
        $PL(i) \leftarrow$ *empty*;
    $Q \leftarrow$ queue built of hosts in H in
        decreasing order of communication
        distance from h_1 (source);

    while Q is not empty **do**
    begin
        $h_i \leftarrow$ next host from Q;
        $H \leftarrow H - \{h_i\}$;
        $q_i \leftarrow$ queue built of hosts in H in
            increasing order of
            communication distance from h_i;
        while q_i is not empty **do**
        begin
            $h_j \leftarrow$ next host from q_i;
            append h_j to $PL(i)$;
        end
    end
end

Figure 5.2. The SDF algorithm.

reality in many networks. Communication distnaces may change dynamically due to changes in topology. Moreover, even when they remain the same according to our definition, the real cost of communication between a pair of hosts may change, due to changing loads, for example. Any algorithm that assigns priority lists statically has no hope of optimizing efficiency under such conditions. Therefore, it is worthwhile exploring possibilities for dynamic reassignment of lists, or at least dynamic reordering of hosts within a list. One approach is to maintain the hosts in the increasing order of the cost of communication with the owner of the list. This will be essentially a dynamic version of SDF. The situation is analogous to that of dynamic routing versus static routing of messages in networks.

6. Performance.

As was mentioned above, the algorithm proposed here is the only known alternative (given our requirements and assumptions) to the obvious basic solution described in Section 3. Therefore, it is natural to compare the performance of our algorithm to that of the basic one. Rigorous performance analysis, however, appears to be very difficult in this case and could easily constitute a subject for a separate study. Hence, we limit this section to a qualitative discussion of the performance issues.

In the absence of any failures, our algorithm performs with the highest efficiency and lowest delays possible because it uses the efficient multicast facility to dispatch messages. In comparison, transmitting broadcast messages from the source separately

to every recipient host, as done by the basic algorithm, would be far inferior.

When a transient, i.e. isolated, failure occurs, the basic algorithm recovers from it by having the source retransmit the message to the host that lost it. In our algorithm the redelivery takes place not necessarily from the source. The host that redelivers the message is on the recipient host's priority list and probably has a high priority on that list. If the list assignment is done by the algorithm in Section 5, then in all likelihood that will be a "near-by" host. Thus, our algorithm is better at minimizing the cost of redelivery of isolated lost messages.

In Section 2.1 it was argued that handling of network partitions by the reliable broadcast algorithm is of great importance. First, let us compare the two algorithms in terms of their behavior during partitions. Since, in general, there is no reliable way to detect partitions, the basic algorithm will continue sending broadcast messages to all hosts even while the partition lasts (obviously, with no success). Since new messages might be generated during that interval, and each of them might be sent out repeatedly (because acknowledgments do not arrive), this can be rather wasteful. In the proposed algorithm, however, the source never retransmits a message unless explicitly requested to do so, and therefore, no unnecessary traffic results. Instead, the hosts that are isolated from the source select a coordinator, and only the coordinator periodically probes the network to detect when the partition is repaired.

Recovery from partitions is also handled by our algorithm more efficiently. For simplicity, suppose that two partitions are reconnected by one link only (e.g., in Figure 2.1 link l_6 could fail, partitioning the network, and then come back up). Then, if there were k hosts in the partition not containing the host, and m broadcast messages that were generated while the partition lasted, it would take at least km transmissions over the critical link to recover from the partition using the basic algorithm. In contrast, our algorithm would need only m transmission because it uses the underlying efficient multicast, which is presumably clever enough to make k copies of each broadcast message only after the message crosses the link in question. Moreover, as in the case of transient failures, it is very likely that the redelivery will take place from a host in close proximity, not necessarily from the source.

Since, when the basic algorithm is used, every broadcast message goes out from the source separately to every host, congestion of the source's server is possible. This is less likely to occur when our algorithm is used because an efficient multicast facility can be designed to avoid congestions.

It should be pointed out that our algorithm relies more heavily on the use of control messages, i.e., messages other than those containing broadcast data. (A null message sent by the source is an example of such a message. The basic algorithm uses control messages in the form of acknowledgments.) However, in the basic algorithm there is always one control message (acknowledgment) per broadcast message per host, whereas the amount of traffic generated by control messages in our algorithm is totally independent of the number of broadcast messages. Moreover, the amount of such traffic can be adjusted according to the needs of the application at hand, fine-tuning the desired trade-off between cost and recovery speed. For instance, keeping the frequency of null messages down (large δ) will also keep down the cost of the broadcast algorithm. On the other hand, increasing the frequency will allow the hosts to detect missing messages faster and, as a consequence, will result in faster recovery.

Finally, as was already pointed out in Section 3, by distributing the responsibility for reliable broadcast among all hosts we are able to achieve better reliability, in addition to improved performance, compared to the basic algorithm (see Section 3, *Epidemic*

Propagation).

7. Acknowledgments.

We would like to thank Barbara Blaustein, Charles Kaufman, Nancy Lynch, Sunil Sarin, and Oded Shmueli for their helpful comments. Some of the ideas developed here were originally introduced in [Garc85].

8. Bibliography.

[Agui84] Aguilar, L., "Datagram Routing for Internet Multicasting," *Proc. ACM SIGCOMM Symp. on Communications Architectures and Protocols*, 1984, pp. 58-63.

[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.

[DeCh85] Deering, S.E., and D.R. Cheriton, "Host Groups: A Multicast Extension for Datagram Internetworks," *Proc. 9th Data Communications Symp.*, 1985, pp. 172-179.

[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.

[Garc85] Garcia-Molina, H., et. al., "Notes on a Reliable Broadcast Protocol," Computer Corporation of America, Technical Report CCA-85-08, July 1985.

[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.

[Garc87] Garcia-Molina, H., Boris Kogan, and Nancy Lynch, "Reliable Broadcast in Networks with Nonprogrammable Servers," Technical Report CS-TR-123-87, Princeton University, November 1987.

[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.

[Sari86] Sarin, S.K., "Robust Application Design in Highly Available Distributed Databases," *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, pp. 87-94, January 1986.

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

9. Appendix I.

The redelivery algorithm is specified as a collection of event—action pairs such that each action is triggered when the occurrence of the corresponding event has been detected. The same code runs at every host i. In addition there are two procedures that are invoked as part of some actions: *fill*() and *probe*(). *probe*() is a concurrent procedure, i.e., it runs concurrently with the code in which it is invoked. For example, in Action 4 the **end** statement that follows the invocation of *probe*() executes without waiting for the latter to complete.

When invoked at host i, *fill*($DESC(j)$) causes i to send the broadcast messages available to it to fill the gaps in the streams received by hosts in $DESC(j)$. For example, let $DESC(j) = \{M_k, M_j\}$, where $M_k = <5,2,4>$ and $M_j = <4,3>$, i.e., host k has received messages numbered 1, 3, 5 and host j messages 1, 2, 4. Also, suppose that host i has received messages 1, 2, 3, 5, 6. Now, if statement *fill*($DESC(j)$) is executed at host i, i will end up sending messages 2 and 6 to k, and messages 3, 5, and 6 to j. $DESC(j)$ will be updated accordingly. This procedure, of course, uses multicast whenever possible.

Call $probe(PL(i))$ results in host i sending a probing message to the first host on its priority list. If a response is not received promptly, a probing message is sent to the next host on the priority list, etc. The procedure is terminated when one of the hosts responds to the probe. Note that a wrap-around on the list is allowed. This guarantees that eventually contact will be established if at least one of the hosts in $PL(i)$ is in the same network partition as i, even if some probes are lost. As was mentioned above, the probing process does not block the execution of any code that may follow the *probe*() call in the body of an action.

$COORD(i)$ is the current coordinator of host i. Initially $COORD(i) = \text{NIL}$. $DESC(i)$ is defined as follows. Let h_j be the highest sequence number of all messages received by host j. Let g_j be the set of all the "gaps" in the message stream received by j, i.e., $g_j = \{m: m < h_j$, message m has not been received by $j\}$. Finally, let $M_j = <h_j, g_j>$. Then $DESC(i) = \{M_j$: j is a descendant of i in the coordinator tree$\}$. Thus $DESC(i)$ contains the identities of all descendants of i (including i itself). It also tells us which messages each host is missing and/or at which point it has stopped receiving any new broadcast messages(h_i). Before the redelivery algorithm is initiated, $DESC(i) = \{M_i\}$.

Event 1: failure detection or timing out on coordinator
Action 1: **if** $COORD(i) \neq i$ **then** $COORD(i) \leftarrow i$;
$probe(PL(i))$

Event 2: probed by host j
Action 2: **if** $COORD(i) = \text{NIL}$ **then** $COORD(i) \leftarrow i$;
send "coordinator is $COORD(i)$" to j

Event 3: message "coordinator is x" received from host j
Action 3: **if** $COORD(i) = i$ **and** $x \neq i$ **then**
begin $COORD(i) \leftarrow x$;
send $DESC(i)$ to x
end

Event 4: $DESC(j)$ received
Action 4: $DESC(i) \leftarrow DESC(i) \bigcup DESC(j)$;
fill$(DESC(j))$;
if $COORD(i) =$ NIL **then**
if $DESC(i)$ not completely filled **then**
begin $COORD(i) \leftarrow i$;
probe$(PL(i))$
end
else if $COORD(i) \neq i$ **then** send $DESC(j)$ to $COORD(i)$

Event 5: expiration of timer
Action 5: **if** $COORD(i) = i$ **then** send "I AM COORDINATOR" to each host in $DESC(i)$

Event 6: message "I AM COORDINATOR" received from host j
Action 6: **if** $j \neq COORD(i)$ **then** $COORD(i) \leftarrow j$

When host i detects a failure, i.e., an out of sequence broadcast message or no messages for a long period of time ($m\delta$ time units), it initiates the redelivery procedure by setting $COORD(i)$ to i (recall that originally $COORD(i) =$ NIL) and probing the hosts on its priority list ($PL(i)$). Similarly, when timing out on i's current coordinator occurs, which indicates a probable communication failure or a failure of the coordinator itself, i starts looking for a new coordinator. This is accomplished by event—action pair 1.

When probed by another host searching for a coordinator (Event 2), host i joins the redelivery procedure in progress (unless it already has been participating, in which case $COORD(i) \neq$ NIL). Then it sends back to j the identity of its current coordinator (which is i itself, if i has just joined the redelivery procedure).

Action 3 is prompted by the arrival of a message specifying the indentity of the current coordinator. Host i sets its coordinator pointer accordingly, unless it already had a non-trivial coordinator ($COORD(i) \neq i$), in which case the message can be ignored. Then i proceeds to send to $COORD(i)$ set $DESC(i)$ to inform the coordinator of new nodes in the coordinator tree and their message redelivery needs.

When host i receives a message containing $DESC(j)$ (Event 4), it is a sign that a group of hosts (j and its descendants) wants to join the tree of which, they think, i is the root (coordinator), or wants i to become the root of their tree. Another possibility is that a network partition has been repaired, and a previously isolated group of hosts, headed by j as coordinator, requests redeliver of missing messages. In either case i

attempts to fill the message gaps of those hosts as best it can, using the messages it has seen. If i has been inactive until now ($COORD(i) =$ NIL) and did not have all the messages wanted by the hosts, it joins the process and starts probing its priority list on behalf of the hosts on $DESC(j)$. If, on the other hand, i is already a node in a coordinator tree and it has a non-trivial coordinator, then it will forward $DESC(j)$ (properly updated after a dispatch of the missing message) to $COORD(i)$.

$DESC(j)$ from a remote host can serve still another purpose. Namely, from this message the recipient can figure out whether any of the hosts in $DESC(j)$ have some of the messages that it is missing. If so, those hosts can be asked to deliver them.

Every host that is currently a coordinator periodically sends "I AM COORDINATOR" to all its descendants, using the multicast facility. A timer is used for this purpose, whose expiration is Event 5. The timer is reset after the message is dispatched.

Arrival of message "I AM COORDINATOR" (Event 5) is intended to reassure host i that it still has connection to its coordinator. If, however, this message is received from a host other than $COORD(i)$, it must mean that the former coordinator has relegated its responsibilities to a new one. In that case the pointer should be reset accordingly.

10. Appendix II.

Theorem 4.1. Let $G = \{h_1, ..., h_k\}$ be the set of hosts that form a partition that excludes the source of broadcast. Then all hosts in G will eventually configure themselves into a coordinator tree if and only if the priority list assignment is acyclic and jointly complete (provided that the partition is stable for a sufficiently long time).

Proof. Let $PL(h_j)$ for each j, $1 \leq j \leq n$, be an acyclic and jointly complete assignment of priority lists. First we show that the parent—child structure that we call a coordinator tree indeed has no cycles. Suppose to the contrary and let h_c be a node on a cycle. Let h_p be the parent of h_c, which is, of course, on the same cycle. Since a host can attach to another host only if the latter is on the priority list of the former, we must have $h_p R h_c$. Furthermore, since h_p is on a cycle, it must be its own (non-trivial) descendant. Therefore, we have $h_p R^* h_p$, which contradicts the acyclicity assumption.

Now we show that the parent—child coordinator structure eventually becomes a single tree (as opposed to a forest of trees). As before, suppose to the contrary. Let h_r and h_t be any two roots in the forest. They become roots some time after the partition is formed and remain such as long as the partition is stable. Since the priority lists are jointly complete, by our assumption, one of the roots must be on the priority list of the other, say $h_r \in PL(h_t)$. Since both are in the same partition, h_t will eventually respond to probing by h_r. At that point h_r will attach to h_t, which contradicts the assumption that h_r will remain a root for the duration of the partition.

If the priority list assignment is not acyclic or jointly complete, we might have either cycles or multiple roots (coordinators), which is demonstrated by the examples at the end of Section 4. □