Notes On A Reliable Broadcast Protocol

CCA-85-08

Hector Garcia-Molina
Nancy Lynch
Barbara Blaustein
Charles Kaufman
Sunil Sarin
Oded Shmueli

Computer Corporation of America
December 1985

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Air Force Systems Command at Rome Air Development Center under Contract No. F30602-84-C-0112. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, RADC, or the U.S. Government.

CONTENTS

Abstract

The problem of reliable broadcast in various network environments is studied. Special attention is given to utilizing the network links efficiently to achieve timely message delivery.

1. INTRODUCTION

This report summarizes our ideas on the reliable broadcast protocol (or so called "distributor"), ideas developed in the summer of 1984. The report is organized and written as a "working paper" or set of notes, not a polished paper. Our intent is to record the ideas we discussed, so they will not be forgotten and can be used in subsequent papers. We do not present detailed solutions to algorithms; instead we simply discuss issues involved and outline possible strategies. The material is not organized didactically and may hence require reading in several passes. For example, all of our assumptions are listed in one section, although it may be difficult to understand some of them until they are actually used later on in an algorithm. Similarly, most of our examples have been grouped into a single section.

In a *reliable broadcast*, a sequence of messages must be transmitted to a set of computers over a communication network. The messages must all be eventually delivered; in addition we would like for the messages to be delivered as fast as possible, given the actual state of the network.

One application where the need for a reliable broadcast protocol arises is replicated database management. We have been studying such an application, developing a system that provides high data availability in the face of network partitions [Blau83, Garc83, Blau85, Sari85]. In this case, database updates must be reliably broadcast to nodes that hold a copy of the database, as soon as the network permits it and regardless of whether previous updates have already been broadcast successfully. It is this application that motivates our work on reliable broadcasts.

In this report we only deal with the single-source case. Extensions to multi-source should be studied further, but one could run multiple single-source algorithms in parallel. Some additional optimization may be possible in the parallel execution.

We start this report by reviewing our basic model and assumptions. In Section 3, we then discuss our "optional assumptions," i.e., those we have not agreed upon yet. Each choice of optional assumptions may lead to a different algorithm, although different algorithms may be combined into one. Next we discuss the

objectives of the algorithm (Section 4), and some of the general heuristics that may be useful in order to achieve these objectives (Section 5). This is followed by some simple illustrative examples (Section 6), and by a presentation of three "scenarios" (choices of optional assumptions) that we will study in detail (Section 7). The remaining sections discuss each scenario in detail, outlining possible algorithms to solve the reliable broadcast problem.

2. THE MODEL AND BASIC ASSUMPTIONS

(a) *Hosts.* We consider a set of interconnected hosts, each having local non-volatile storage. One of the hosts is the source S. It runs a source process that generates a sequence of messages to be reliably broadcast. Each broadcast message must be received by all other hosts, and must be handed to a recipient process that runs there. We refer to the broadcast messages as *data messages* in order to distinguish them from others in the network (e.g., control messages, point to point messages).

(b) *Servers.* Each host h is attached to a local communications server, $C(h)$. (The server may reside on the same physical computer as the host, but there is a clean interface between them. The server could also be partitioned between the host and a dedicated communications processor, as occurs in the ARPANET.) The servers are tied together by a computer communication network.

(c) *Costs.* Each physical link in the network, whether it is shared or not, has a cost associated with it. This cost reflects the transmission delay and bandwidth of the link.

(d) *Eventual Connection.* For each data message m to be sent from S to host x, there may never be a complete communication path over which to transmit m directly. However, it will always be possible to eventually transmit m through intermediate hosts. That is, there will always be a sequence of hosts and "sufficiently long" (possibly disjoint) time intervals, such that in the first interval S can communicate with the first host, in the second interval the first host can communicate with the second host, and so on, until in the last interval, the last host can communicate with x. We cannot define

"sufficiently long" formally without a specific broadcast algorithm in mind, but intuitively, each period should be long enough to allow the two communicating hosts to discover that they can communicate and to exchange data messages that have not been seen by each other. (Note that this property implies that hosts must have non-volatile storage. If servers have non-volatile storage, then this property could be stated in terms of a sequence of hosts and/or servers.)

(e) *Hosts Do Not Fail.* The hosts are reliable and never fail. (Host crashes can be "simulated" by server failures, as long as critical host data structures are kept in stable storage.)

(f) *No Bogus Messages.* The network does not generate bogus messages.

(g) *Transitivity.* If host x receives a message from host y, then at that instant it is as likely for y to be able to communicate with a node z as it is likely for x to be able to communicate with z. This assumption will be useful in the following scenario. Suppose that a host x in unable to communicate with host z. To know when communications are restored, x must periodically send messages to z, and wait for a response. However, if there is a host y in the same situation as x, and both x and y can communicate, then it is not necessary for both to periodically probe z. Whatever host establishes communications with z can inform the other. The assumption is reasonable because the network performs adaptive routing. That is, suppose x is probing z, but it is $C(y)$, not $C(x)$, that is reconnected to $C(z)$. When x sends a message to z, the message gets through because the network will route it through $C(y)$.

(h) *Communication Failures.* We make no other assumptions about message delivery. Specifically, we assume that messages can arrive out of order, can have arbitrary delays, can be duplicated, or can be lost at any point.

(i) *Frequency of Failures.* Failures are not going to be very frequent. As a consequence, the performance of the reliable broadcast algorithm is mainly going to be a function of how fast messages are sent during periods when no failures occur. ("No failure occurs in time period P" means that no system component changes its state from functioning to failed, or vice versa, in this time. However, there could be components in either state in P.) How fast

the protocol adapts to a failure (or repair), or how efficient it is during the adaptation phase is not as critical.

(j) *Down Time.* The length of time between the failure and the repair of a component may be significant.

(k) *Number of Hosts.* The number of hosts participating in the reliable broadcast protocol is relatively small, especially compared to the number of computers in the network. Thus, strictly speaking, we are attempting to solve the reliable *multicast* problem, not the reliable broadcast problem. We do not want to have the data messages delivered to all computers in the system; only to hosts, i.e., to those computers participating in our application (replicated database management). However, since we have been using the word broadcast in our discussions, we will continue to use it in this report.

3. OPTIONAL ASSUMPTIONS

We now list the assumptions that are not fixed. Note that there are dependencies among these assumptions. Specifically, a given choice in one of the categories may restrict the choices in other categories.

(1) *Communication Lines.*

  (a) Lines between servers are point to point only.

  (b) Servers are only linked by broadcast (e.g., Ethernet) lines.

  (c) Lines can be both point to point and broadcast.

(2) *Programmability of Servers.*

  (a) The implementors of the reliable broadcast protocol can modify (or rewrite entirely) the code running at the servers and at the network switching nodes. Thus, we are free to do our own routing at the servers.

  (b) The code and routing strategies of the servers and switches is fixed.

(3) *Diskless Servers.*

  (a) Servers and network switches do not have non-volatile (e.g., disk) storage available. (Or alternatively, they may have it but it is so limited that we cannot use it in our protocol.)

(b) Servers and switches have non-volatile memory. This alternative is only of interest if the servers are programmable.

(4) *Clustering.*

(a) Servers fall into clusters, where the available bandwidth between cluster members is relatively high, and the intercluster bandwidth is limited. Using cost terminology, cluster members are connected by low cost links, while intercluster links are high cost.

(b) There are no clusters.

(5) *Host/Server Interface.*

(a) To send a message, a host gives it to its server, together with one or more destination addresses. The server then attempts to deliver the message to all addresses.

(b) The host can only hand the server one address with each message.

(6) *Broadcast Facilities.* (only applicable for option (5a) above)

(a) The servers have and utilize an efficient multicast algorithm. If server h receives a data message for delivery to hosts A,B,C,D, and hosts A,B can be reached via line 1 and hosts C,D can be reached through line 2, then a single copy of the message will be sent over each line. At some later point, each copy will be converted into 2 copies, one for each intended host. If h and the servers for A,B,C are on an ethernet, then h will broadcast a single message on the net, that will be received by each destination server.

(b) A network-wide multicast algorithm is not available. However, if a source server and the destination servers (or a subset of them) are on the same broadcast line, then the source will transmit a single message on this line.

(c) Servers have no multicast facilities. A multi-destination message is simply converted into multiple copies of the message.

(7) *Information on Arriving Message.*

(a) When a server delivers an incoming message to a host, the identity of the sending host is given, but no additional information as to what path

was followed is provided.

(b) When a server delivers an incoming message to a host, the identity of the sending host is given. In addition, a single bit is provided describing the cost of the path followed by the message. For this, all network links are classified as *cheap* or *expensive*. A link is cheap if its cost is below a given threshold, and is expensive otherwise. If the message arrived via cheap links only, the cost bit is set to cheap. If the message traversed at least one expensive link, then the bit is set to expensive. No additional information as to what path was followed is provided.

(c) When a server delivers an incoming message to a host, all information on the path followed, including cost of the links, is given.

(8) *Static Information.* The following static network information may be available to the reliable broadcast protocol:

(a) Each host knows the static topology of the full network.

(b) Each host only knows its local topology, i.e., the servers (and switches) that are directly connected to its server.

(c) In addition to the topology (either full or local), a host knows the cost of the links.

(d) Same as (c), except that only the classification of links into cheap or expensive is known. (As discussed earlier, a link is cheap if its cost is below a given threshold, otherwise it is expensive.)

(e) The topology (local or full) is known, but no cost information is available.

(f) No static information is available.

(9) *Dynamic Information.* The following dynamic network information may be available to the reliable broadcast protocol:

(a) The routing information *routing*[p,q], but only where p is the server of this host. (*routing*[p,q] gives the outgoing link of server p that should be used to send a message to q.)

(b) The routing information *routing*[p,q], for all servers p.

(c) The up/down status of the servers and hosts.

(d) Other statistics like queue sizes at the nodes, and transit times between nodes.

(e) Combinations of the above.

(f) None of the above.

4. OBJECTIVES OF ALGORITHM

(a) *Reliability.* The broadcast should be reliable, i.e., all messages should be eventually delivered to the recipient processes.

(b) *Low Average Delay.* The transmission delay, averaged over all broadcast messages, should be minimum. The fact that we are interested on all broadcast messages is important: we do not want to broadcast one message really fast, at the expense of following messages.

(c) *Limited Interference.* The transmission delays of non-broadcast messages should also be kept reasonable. We do not want to clog the network just to get our messages through fast. For example, if broadcast messages arrive at the source every hour, a hot potato algorithm may be tempting: it gets the broadcast message to the destinations fast. It also floods the network, but by the time the next broadcast arrives, the congestion would have subsided. Yet, we reject this strategy because it delays non-broadcast messages. At the other extreme, we do not want to minimize delay of non-broadcast messages exclusively, because then we could come up with a broadcast algorithm that is slow but sends few messages.

(d) *Prompt Delivery.* The reliable broadcast of a message should not be delayed simply because a previous message has not been delivered yet. For example, if a host receives data message n from the source, it should be handed to its recipient process, even if this process has not seen all data messages that were generated before n.

In summary, the performance objectives are difficult to define precisely, and even harder to achieve. A truly optimum broadcast algorithm would have to have perfect knowledge of the topology, of the failures, of the current loads and

queues at each link, and of the future message arrivals. Since this is not possible, our algorithm will use heuristics to make decisions. They are rules that are likely (but not certain) to yield good performance.

5. DISCUSSION AND GENERAL HEURISTICS

(1) It is useful to break up the reliable broadcast problem into two components:

(a) The unreliable broadcast problem (UB), and

(b) The gap filling problem (GF).

UB only deals with getting new data messages from the source to the destinations efficiently. A protocol that solves this part of the problem, does not make any guarantees about message delivery. GF deals with filling in gaps of data messages that are left by UB. The protocols for both parts may or may not be combined into one, but we have found this conceptual division of the problem to be useful in all cases.

(2) If the network provides (unreliable) multicast facilities, we should use them as much as possible, especially for UB. During no-failure periods (which are the important ones from the point of view of performance), it is unlikely we can do better than the network facilities, so let's use them.

(3) If network multicast facilities are not provided, and our algorithm will perform broadcasts in no failure periods, then its best to use a tree (rooted at the source) for this operation (i.e., for UB). Non-tree strategies (e.g., hot potato [Rose80]) generate too many messages. Let us call the unreliable broadcast tree the UB tree. The nodes of this tree are the hosts, unless the servers are programmable, in which case servers may also be nodes.

(4) From our discussion of the objectives, we know that this UB tree must adjust to failures (i.e., we want to return to efficient broadcasts after a failure occurs). Of course, it is also desirable to adjust to the changing loads in the network. To make these adjustments, dynamic information must be exchanged among the nodes that make up the UB tree. Nodes can either request this information (polling), or it can be sent to them periodically. The following information about node j could be useful to node i in trying to adjust the UB tree (specifically, when i is trying to decide if j should be its

parent in the tree):

(a) The up/down status of j.

(b) The sequence number of the latest data message that j has received. (Awerbuch's algorithm [Awer84] uses only this information.)

(c) The load (e.g., queue sizes) at j.

(d) The cost of the line(s) that connect (or are most likely to connect) i and j.

(e) The cost of the line(s) that connect (or are most likely to connect) j to the source. That is, is a message from S likely to go through $C(j)$?

(f) The time it takes j to respond to a polling request.

(g) Message transit times and routing information. This information is usually collected by the routing mechanisms, and may be available to us without additional message exchanges with j. One way to use routing information is as follows: if $routing[i,S] = j$ or a server close to j, then j may be a good candidate father for i. (This is the basic idea of the Reverse Path Forwarding broadcast algorithm [Dala78].)

(5) For adjusting the UB tree, a child node should use filtering to avoid instability. That is, when node i evaluates information about node j, it should not always decide according to the very "latest" information. For example, if node i has received up to message 5 from its current father in the UB tree, it should not select as its new father a node that has just received message 6 (its own father also may have just received it and is in the process of forwarding it to i). However, if this node received message 6 a "significant" amount of time earlier, and i has still not received message 6 form its own father, then i should consider changing fathers. Similar types of filtering should be used with the other types of information discussed in (4) above.

(6) The amount of information exchanged for UB tree adjustments must be controlled. For example, if polling is used, the frequency of polling and the number of sites polled must be limited, to avoid congesting the network. There are two cases to consider here:

(a) Suppose that data messages are arriving at node i with some regularity. In this case, adjustments to the tree are not absolutely necessary; they are only done to try to improve performance. Thus, even though we could actually poll (or get information from) every node in the system, it is probably best to restrict ourselves to nodes that are likely to be good fathers or that are cheap to poll.

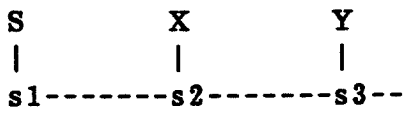
(b) Suppose that the father of i (or an ancestor in the tree) has failed, so that i is not receiving data messages. In this case, node i is forced to check with every node until it finds one that can give it more data messages. However, if there are other nodes that i can reach and that are in its same situation (i.e., disconnected form source), they should all try to cooperate to avoid having everyone polling everyone. (See Transitivity assumption in Section 2.) For example, the nodes can elect a coordinator. The coordinator can continue to probe all nodes, and can in the meantime send "I am alive" messages to its subordinates to tell them that the search is actively continuing. Similarly, the nodes can configure themselves into a tree, where the root is the only probing node. This tree can be formed with the remains of the UB tree after the failure, and once a connection is established to the source, can become the new UB tree. The coordinator or root may also probe nodes with different frequencies, checking with the nodes that are more likely to be connected to the source more often.

(7) For gap filling (GF), again, we do not want to have every node polling everyone. As in section (6b) above, we would prefer that nodes cooperate with each other to fill their gaps. The same ideas could be used here: a node could delegate the job of probing the rest of the system for the missing data messages to his father in a tree, or to its coordinator. (If a tree is used, it could be the same as the UB tree.)

6. THE EXAMPLES

The following examples illustrate some of the points made in the previous sections. They were also the main driving force behind the algorithms we will outline later in this report. That is, the algorithms were initially designed to try to make broadcasts efficient in these cases, and every new idea we developed was always tested against these examples. Thus, it is probably useful to present them here for reference.

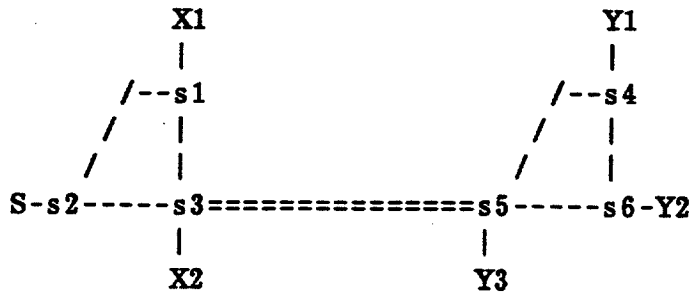
The Simple Example.



(S is the source, X and Y are hosts, and s1, s2, s3 are the servers/switches.) This example illustrates that without a multicast network facility, or without server programmability, the reliable broadcast algorithm will not be efficient. Specifically, there should only be one copy of each broadcast message sent from s1 to s2. Then s2 should make two copies, one for X and one for s3.

If s2 cannot handle a multi-destination message, then the hosts must attempt to do an efficient broadcast job, but "their hands are tied up" because s2 in not under their control. If S sends a single message to X and then asks X to forward it to Y, we are using the s1-s2 line efficiently, but the message to Y is being delayed at X. If, on the other hand, S sends two copies (one for X, one for Y) over the s1-s2 link, this may lead to congestion on this link. Neither solution is good, and furthermore, selecting the "less bad" one is difficult: the choice is a function of the capacity of the s1-s2 link, the load at X, and the non-broadcast message load currently in the network.

The Mountain Example, Part I.



(This is really a more complicated version of the simple example.) The source S and hosts X1, X2 form a cluster, and hosts Y1, Y2, Y3 form another cluster. All lines are cheap, except the s3-s5 line.

As in the previous example, it is best to have a multicast algorithm that sends a single copy of each broadcast message on line s3-s5. If this is not possible, the reliable broadcast algorithm should send one copy to Y3, and then have it forward the message to Y1, Y2. In other words, the UB tree would have Y3 as child of X2 (say), and Y1, Y2 as children of Y3.

If failures occur, the tree should adapt. For example, suppose that server s5 fails. When hosts Y1, Y2 notice that their father is not active, they should try to get in touch with some other node that can provide the missing data messages. As discussed earlier, it would be preferable to have only one host, say Y1, attempting this, while the second host, Y2, waits for information from Y1. (At the same time, host Y3 will also be trying to contact other nodes.)

Suppose that now server s5 is repaired, but that its connection to host Y3 is still down. At this point, Y1 will communicate with a host in the other cluster, say X2, and make it its new father. (As in the simple example, it is difficult to tell whether S or X2 would be the best father for Y1.) Host Y2 should then become a child of Y1 (not of X2), to avoid two message transmissions over the expensive s3-s5 link.

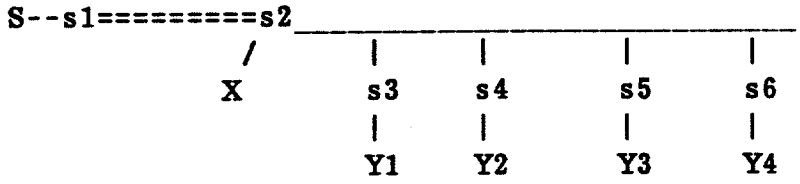
When the s5-Y3 link comes up again, host Y3 should select Y1 as its father, to avoid an extra message over the expensive link. However, the resulting tree is different form the one that existed before the failure, and may not be as good. Specifically, notice that now each data message traverses the s5-s4 link twice:

once in going from X2 to Y1, and once going from Y1 to Y3. If this is undesirable, we may want to have an algorithm that recognizes this, and makes the UB tree evolve to its original state.

The Mountain Example, Part II (or "The Other Side of the Mountain, Part II"). The diagram is the same as in part I, except that the s3-s5 link is now an expensive broadcast link that reaches all servers, s1 through s6. (All servers have packet radio transmitters/receivers, but servers in different clusters can only communicate if an airplane with a repeater flies overhead.)

The best strategy in this case is for the source to broadcast a single message over the broadcast link that is heard by everyone. However, the reliable broadcast algorithm must recognize the special properties of broadcast links in order to achieve this. (And of course, the servers must be able to handle multi-address messages from their host, assumptions 5a and 6b.) For example, suppose that after some failures the UB tree has Y3 as child of S, and Y1, Y2 as children of Y3. From our discussion in the previous example, Y2 would not want to become a direct descendant of S, for this would place an additional load on the expensive link. However, in this case, there is no extra cost in having S send to Y1 and Y2: the same message to Y3 can be received by these nodes. Thus, the UB tree should really have Y1, Y2, Y3 all as children of S.

The Local Net Example.



Servers s2 through s6 are on a local area network, of the ethernet type. The only expensive link is s1-s2. This example is useful in studying how hosts should cooperate when they are cut off from the source. For example, if link s1-s2 fails, the hosts should elect a coordinator or root to perform the probing of the rest of the network. This coordinator should probably be host X. It could send a single "I am alive" message to Y1, Y2, Y3, Y4 while it is probing. Also, during the election of X, the protocol could take advantage of the broadcast facilities.

7. THE SCENARIOS

Given the number of optional assumptions we have presented in Section 3, there are clearly many cases to consider, and each one can in principle have a different reliable broadcast algorithm. Instead of trying to study all cases, we have selected three basic scenarios (plus a few minor variations) which we think are the most likely to arise in practice:

I. *Programmable Servers.* For this scenario we make assumptions (1c), (2a), (3b), (4b), (5b), (6c), (7a), (8b, 8e), (9f). The servers and switches are programmable and they have non-volatile storage. Thus, to take advantage of these facilities it is best to program the reliable broadcast protocol within each server, and to have the servers themselves store data messages received in the past. So in essence, we can forget about hosts in this scenario, and assume that the source and recipient processes are also running in the servers.

This scenario arises when the computers that run the applications are linked directly together, without a communications sub-network like the ARPANET. It is also the scenario used by Awerbuch [Awer84], except that we have different failure assumptions.

In this case, no dynamic information and multicast facilities are available; the only static information available is the identity of the nodes directly connected to each server. However, since the protocol is running at the servers and switches, it is possible for it to collect some of this information, if necessary (e.g., if each server adds its identification to each message it forwards, then the path traversed will be known to the recipient process). Similarly, if desired, an unreliable multicast algorithm can be implemented.

For simplicity, when we present our protocol for this scenario, we will first assume that the communication links are all point to point (assumption 1a), and then we allow the addition of broadcast links (assumption 1c).

II. *Non-Programmable Servers, with No Multicast Facility.* For this scenario, we make assumptions (1c), (2b), (3a), (5a), (6b). This case arises when there is a communications sub-network like the ARPANET that links together the hosts, but it has no multicast facilities (except if source and destination

servers are on same broadcast line).

We consider three sub-cases:

(i) The network is completely unknown, and we have no dynamic information about it. That is, we make assumptions (4b), (7a), (8f), (9f).

(ii) The routing tables at all servers and switches are available. That is, we make assumptions (4b), (7a), (8a, 8e), (9a, 9c).

(iii) We have link cost and clustering information. Specifically, we make assumptions (4a), (7b), (8a, 8c), (9f).

In some of these sub-cases, we first assume that the network is point to point only (1a), and then we generalize to the case that includes broadcast links (1c).

III. *Non-Programmable Servers, with Multicast Facility.* For this scenario, we make assumptions (1c), (2b), (3a), (4a), (5a), (6a), (7a), (8a, 8c), (9f). Although many networks currently do not have adequate multicast facilities (e.g., the ARPANET), they may have them in the future, and this makes this case interesting. Furthermore, the multicast facility lets us solve the unreliable broadcast protocol easily and efficiently (with the routing decisions made at the server/switch level where they should be made), leaving the reliability problems to the hosts that usually have the non-volatile storage.

In the following sections, we discuss reliable broadcast algorithms for these scenarios.

8. PROGRAMMABLE SERVERS

This is the simplest case to consider. We consider the case where the network servers are directly programmable, and there is no underlying routing protocol. (We are going to do our own routing.) This case is a lot like Awerbuch's [Awer84], except that: (a) messages are permitted to get lost, and (b) the network might not be entirely point-to-point, but might be partly composed of broadcast channels. We also allow use of synchronized clocks and timeouts where they are needed.

Data messages are assumed to be sequence numbered. At any point in time, a node has a set INFO which records the set of indices of data messages so far received by that node. It is useful to have a partial ordering on sets of indices of data messages; we say that A < B provided that the largest element of A is strictly less than the largest element of B. We use this notation in the rest of the report.

8.1 The Basic Idea

The basic idea is very much like Awerbuch's. The servers configure themselves into a UB tree. For this, each node except the source attempts to "attach" to a parent, who will be responsible for transmission of data messages in the normal course of events. We refer to the resulting structure as the *parent graph*, not the UB tree, to stress that a tree is obtained only when there are no partitions and when failures are handled properly by the algorithm.

In order to avoid cycles in the parent graph, and in order to obtain up-to-date information, a node will only attach to a parent when it knows that the parent's INFO set is larger than its own INFO set. (That is, the parent's largest numbered message has a greater number than its own largest message. This is heuristic 4b of Section 5.) Moreover, a node will never accept any message larger than its own largest message from anyone other than its parent. These rules preserve the invariants that no node ever has a message larger than the largest message of any of its ancestors, and that no cycle is ever formed.

8.2 Filling Gaps

In Awerbuch's case, all messages were received by all nodes in numerical order. This was a result of his assumptions about FIFO behavior of links. (This result could also be achieved by delaying the transmission of some messages until messages earlier in the sequence were known to have been delivered.) For our application, it is not important that messages arrive in the proper order. Rather, it seems preferable to transmit all messages possible, and fill in the gaps by some separate procedure.

We assume that each node keeps some "map" information containing the latest set of data messages it knows that each other node has already received.

This information is kept in an array MAP of sets of message indices, where the array is indexed by the set of other nodes. The information in MAP must be updated periodically; this can be done according to many different policies. (We discuss one below.) A node which sees that it has a gap in its own received data message sequence where some other node has a data message can send a request to that node for that particular data message. Certainly, any time a node receives a gap-filling data message from any other node, no matter for what reason, the node can use that data message to fill its own gap.

If it is considered very important to fill gaps quickly and we are not too concerned about a little extra message traffic, the node might send out several requests to different nodes for the same data message. It is important that such requests only be sent out for data messages that are actually gaps, i.e., less than the requester's largest index message. This is because a node is only supposed to accept data messages larger than its maximum from its parent.

Gaps could be filled in via the parent graph, but there is no particular reason to restrict them to be filled in this way. It would be inconvenient to have everyone trying to fill everyone else's gaps, thereby flooding the network, so some discipline should be imposed on gap-filling. However, there are many options.

Probably the most natural option is for each node to maintain MAP information for each of its network neighbors. (Network neighbors include not only point-to-point neighbors but also broadcast neighbors.) Nodes could update MAP by operating cyclically, sending their own INFO set to all network neighbors every so often (e.g., every half minute).

It seems quite convenient to use the network neighbors for filling gaps, even though the parent graph neighbors suffice for ordinary message transmission. This is because gaps might be likely to be filled faster if more neighbors got polled. Also, if the source happened to get disconnected, in the parent graph, from each of two subtrees, each of which had the same maximum index data message, but with incomparable sets of gaps, there is no way that the ordinary parent graph would cause the gaps to become filled. The roots of the two subtrees would not find parents to attach to. Polling network neighbors would cause the two subtrees to fill in each other's gaps, however.

8.3 Attachment

Attaching to a new parent involves some polling and an attachment protocol. Nodes will periodically inform their network neighbors of their highest data message index, just in case the neighbors might want to attach to them. (It is not necessary for a node to send its highest index to a neighbor j for which its MAP[j] already contains something at least as big as this highest index. This situation means that j would not find this node an appropriate parent anyhow, since its highest message is too small.)

Whenever a node i discovers that a neighbor's INFO is larger than i's, i may choose to try to attach to this node as a new parent. In practice, one might want to "damp" the effects of slight extra information, or extra information that persists for a short time. If i decides to try to attach to a new parent j, i first detaches itself from its old parent, if it has one (by setting its PARENT pointer to j and sending a DETACH message to the old parent). Then i sends a DECLARE message to j, with i's highest data message index. Then i waits for an ADOPTION acknowledgement from the j; j, if it acknowledges, will include all the "latest" data messages it has — those messages with indices higher than i's highest. This protocol is very similar to Awerbuch's.

There are a few technical points worth mentioning, about the attachment protocol. First, in the interval between when i sets its PARENT pointer to j and an ADOPTION message is received, it is permissible for i to accept new messages, even messages bigger than i's largest, from j. This will not violate the invariants. Second, if an ADOPTION message fails to arrive in a reasonable amount of time, i should terminate the protocol unilaterally and begin to search for a new candidate parent; while doing so, it can keep its PARENT pointer pointing to j, and continue to accept arbitrary messages from j.

Third, there is no guarantee that the DETACH message will get through. If it does not get through, the old parent will continue to send data messages to i. For point-to-point networks, this is no problem. Node i can detect this situation, if it later receives a spurious data message, i.e., one larger than i's largest from someone other than the node indicated by i's PARENT pointer, and can send another DETACH message. (Node i should probably wait some small interval of time to

make sure it has given the old parent time to receive the first DETACH message.) Things are a little more complicated in the broadcast subnetwork case, as we will describe below.

8.4 Propagating Data Messages

Except in the case where nodes poll their network neighbors for data messages to fill gaps, data messages get propagated along the parent graph only. There are two cases in which such propagation occurs. The first is the ordinary broadcast case. The source originates data messages which get sent downward in the parent graph. The second is the propagation of gap-filling data messages. When a node fills a gap by obtaining a data message from a network neighbor, it propagates this data message in all directions in the parent graph. The same general data-message-processing protocol is used in both cases. Namely, when a node i receives a new data message, it examines its MAP entries for all of its parent graph neighbors. It sends the message to all neighbors that do not already have the message index recorded in the appropriate MAP entries. We have already said that the MAP entries get updated when nodes inform their network neighbors about their INFO sets. We will also allow i's MAP[j] to get updated whenever i receives a new data message from j. Then the ordinary broadcast case will proceed correctly as a special case of the general data-message-processing protocol described above. (A node knows not to reflect a new message back to a parent who has just sent it because the node will have recorded that the parent already has it.)

8.5 Broadcast Subnetworks

We have not yet taken advantage of the broadcast subnetworks. The general way that the broadcast subnetworks are used is as follows. Whenever a node wants to send the same message to several network neighbors with which it shares a common broadcast channel, the message is broadcast. There are a couple of particular places in the algorithm where this broadcast will come in handy. First, a node might broadcast data messages to several parent graph neighbors at once. Second, a node might broadcast its INFO to several network neighbors at once.

A message might actually be received by some nodes for which it was not intended (i.e., be "overheard"). In cases where this matters, the message can contain the set of intended addresses, but there are some cases in which the nodes will be able to react appropriately without such tagging. Consider the two cases mentioned above.

First, if a node *i* receives an overheard data message, it can use it without penalty if the message fills a gap; it can discard it if it is larger than *i*'s largest message. In either case, it can update MAP appropriately. So far, there seems to be no need for *i* to know whether *i* was included in the set of intended addresses. However, in the description of the attachment protocol above, it was suggested that receipt of spurious data messages might be used to trigger sending of DETACH messages. However, if the spurious data message is one that is overheard, it is not desirable to send such a DETACH message. Thus, it seems necessary for a node to be able to distinguish those data messages which are overheard from those which might be arriving from an old parent. The best way to do this seems to be to include the set of intended addresses in the broadcast message.

Second, if node *i* receives an overheard INFO message, it just uses it as it would use any INFO message, to update its MAP. It is not necessary to include addresses here.

It is very desirable to try to have broadcast subnetwork neighbors close in the parent graph (assuming the broadcast subnetwork is functioning). It seems that the rules we are already using would tend to do a pretty good job of arranging this. In particular, the ability to "overhear" broadcast INFO messages and use them to update MAP, together with the parent-finding rule we are using, seems to tend to cause nodes to attach to parent-graph ancestors which are on the same broadcast network.

We could augment this rule by an explicit test, however: we can allow any node *n* which is on a broadcast subnetwork to look for any ancestor *a* which has a child *c* such that *n*, *a* and *c* are all on the same broadcast subnetwork. Then *n* should attempt to attach to *a* directly as a parent.

Searching for such a node could be done in either of two ways: (1) by periodic polling of broadcast subnetwork neighbors, or (2) by following PARENT pointers. There are engineering decisions to be made here, about the frequency of searching, and in case (1), about whether all subnetwork neighbors or only some subset will be examined.

Attachment to node a still requires checking that the inequality on largest messages holds. (If the search for a is carried out by following PARENT pointers, then by the time a is found, it might no longer actually be an ancestor of n, so acyclicity cannot be guaranteed simply by the fact that n is reattaching to an ancestor.) In order to get the inequality to hold, it might be useful for n to wait, perhaps discarding some incoming messages from its former parent.

Note that this test involves a static test — whether the three nodes are physically on the same broadcast net. It is possible that the broadcast net might be down, in which case the effect of this protocol might just be to "collapse" part of a path in the parent graph. No harm would be done in this case — in fact, message delay would probably improve, but the savings in messages due to use of the broadcast would not be realized.

8.6 Optimizations

There are some easy optimizations in space possible. For example, a node can throw away data messages when its MAP tells it that all its network neighbors already have them. Also, a node i need not send all of INFO to a neighbor j, just those indices not in i's MAP[j].

8.7 Code

The code has not been written out completely. The algorithm is to be written as an interrupt-driven algorithm, where the interrupts are both the arrival of messages and the expiration of timers.

9. NON-PROGRAMMABLE SERVERS; NO MULTICAST

In this section we will assume that the servers are not programmable and that there is no multicast facility. The network thus consists of "hosts" which are programmable, and "servers" which are not. Each host is connected to a server, and the servers are connected in some network of the sort described in Section 2.

For this scenario we consider several cases, based on the server network configuration, the communication primitives provided to the hosts, and the static and dynamic information available for decision-making.

9.1 Unknown Server Configuration

First, let us consider the extreme case, where the server network is completely unknown. It does not seem possible to get optimal performance in this case. For example, consider once again the Simple Example of Section 6, and assume that the s1-s2 link in the network is of high cost. The source can send messages to X and Y separately, but this means that two copies will traverse the expensive s1-s2 link. Less message traffic flows over this link if the source sends messages only to X, and X thereafter sends the messages to Y.

It seems impossible to detect such a situation without some information about the server network. We considered doing "experiments" involving trying alternative routes to see if there was any observable improvement, but it seems that the observable improvement might be too small. This is especially true if the application we are programming is assumed to be sharing the server network with many other applications. Thus, we will not attempt to detect such situations in the absence of any information about the server network.

In this situation, we can still use some of our ideas, however. An algorithm similar to the one described in the preceding subsection can be run on hosts rather than servers, to configure a parent graph. Some differences are as follows.

First, nodes need to know whom to poll in trying to find a new parent. The potential parents are now all the nodes of the network. Similarly, nodes need to know whom to tell their INFO to. Again, potential neighbors for gap-filling are all the nodes of the network. Each node seems to require a set C of candidates to try (or to favor) for both of these tasks.

Second, direct use cannot be made of the broadcast subnetworks. We assume that a server, when presented with a request from a host to send the same message *m* to several nodes, will know to take advantage of network broadcast where possible. However, the hosts do not have enough information to configure the parent graph in order to make hosts whose servers can communicate via broadcast channels adjacent. (Hosts do not "overhear" broadcasts, and the explicit rule cannot be used.)

Although this algorithm will miss some optimizations, it still serves to implement reliable broadcast at what is probably a reasonable cost.

In the remainder of Section 9, we assume that, although the servers are not programmable, we have partial information about the server network.

9.2 Point-to-Point Server Networks: Routing Tables

For simplicity, in this section assume that the server network is all point-to-point. One of the first kinds of partial information we considered was the server routing tables. For example, one might consider using "extended reverse-path forwarding" (ERPF) [Dala78] to construct a tree to use for broadcast.

In order to use the routing tables, the information must be sent somehow to the hosts. Since not all the servers need correspond to hosts, it might be necessary that some hosts have information about servers other than their own. Assuming that this information can get to the appropriate places, an ERPF tree can then be set up as usual. All the hosts will appear at leaves of the tree. The ERPF tree is not used directly for implementing the broadcast; rather, the hosts have to derive some kind of parent graph of hosts from the ERPF tree. The host parent graph should be configured as much as possible like the ERPF tree. The (dynamically changing) host parent graph will be used for normal broadcast, but will only provide unreliable broadcast, both because of lost messages and because of messages which get missed during changes in the host parent graph. Thus, a gap-filling procedure is still needed.

Gap-filling can be done much as before; we must specify which hosts should try to keep which other hosts informed about their INFO. One possibility is to stick to parent graph neighbors, but this has the same problem as described

earlier: if the source dies, disconnected subgraphs will never bring each other up to date. Another idea is for each node to keep a set C of "close neighbor" hosts in the network. (Exactly how this information is known is unspecified here, but presumably some network topology information can be provided along with the routing tables.) C is then the preferred set of nodes to use for gap-filling; the nodes in C are kept informed more frequently than other nodes.

In this case, where the parent graph is being configured by a separate method, it does not seem to matter if the gap-filling procedure actually does more than just fill real gaps; it is allowable for the gap-filling procedure to also give messages to a node which are larger than the node's largest message. The policy of only accepting messages larger than the node's largest was required earlier in order to prevent formation of cycles in the parent structure; however, now the parent structure is constructed by independent methods, so the policy is not required.

It seems that the strategy of this section should do a good job of handling the difficulty described in the example of Section 9.1 (i.e., the Simple Example of Section 6). The ERPF tree would connect the servers in a line. Presumably, the host parent graph would be constructed to connect the hosts in a line also.

The strategy would also perform well for the Mountain Example (part I) of Section 6. ERPF would configure a tree of servers, with hosts all at leaves, where only a single edge would go over the expensive link s3-s5. When a host parent graph is constructed from this server graph, it should look as much like the server graph as possible; thus, it would tend to have few edges going across the expensive link.

The strategy of this section seems to work rather well, assuming that the host graph could be configured to look a lot like the ERPF graph. But how can this be done? We have not worked out a complete algorithm to configure and update the host parent graph from the information presented in the routing tables. A general strategy might use the following ideas. Each server needs an "owner" host, a "nearby" host who will maintain the server's routing information and act as the server's surrogate in the broadcast tree. The notions of "owner" and "nearby" are dynamic, however. They seem to depend most naturally on the ERPF tree itself.

In fact, the best policy seems to be to define the "owner" of a server to be the nearest host to that server in the ERPF tree. (Ties are broken arbitrarily.) The servers with a given owner are all contiguous in the ERPF tree, and are contiguous with the owner. Then the parent of a host i in the host parent graph is defined as follows. Parent pointers are followed from i just until a server is reached whose owner is some $j \leq i$. Then j is defined to be the parent of i.

This strategy has been described as if central information were available and the underlying server network and routing tables were unchanging. More work is required to see how to properly distribute the information. If this algorithm seems interesting, we will pursue this further.

The ideas in this section should generalize to other means of constructing the broadcast tree besides ERPF. Basically, what has been done here is to break the problem up into two pieces — construction of a host parent graph for ordinary broadcasting, and a gap-filling procedure. Assume that any method is provided which generates an appropriate host parent graph. Then this graph can be used for propagation of data messages as before, and the same gap-filling strategies described earlier in this section can be used to fill gaps (and perhaps add other messages that do not actually fill gaps).

9.3 Point-to-Point Server Networks: Clusters

In this section, we describe another algorithm, similar to the one in the first section, designed for a network for which certain static and dynamic information is provided. Again, assume for simplicity that the server network is point-to-point only. An important problem with which we must contend is the variable bandwidth of the links. As a simplification for a first cut, let us assume that the links in the server network are divided into "cheap links" (i.e., those of high bandwidth) and "expensive links", (i.e., those of low bandwidth). Later, a hierarchy or continuum of costs could be considered.

At any point in time, certain pairs of hosts can communicate via cheap links only. The ability to communicate via cheap links only, defines an equivalence relation on the hosts at any fixed time. The equivalence classes will be called "clusters". The organization of the hosts into clusters changes dynamically. We

assume that two types of information about the network are available to the hosts. First, each host has some static information — a set C of potential cluster neighbors. The set C for node *i* contains those hosts with which it is possible for *i* to communicate over cheap links, at least some of the time. The C sets could be thought of as describing physical network topology. Second, each node obtains some dynamic information. Every message that arrives from another host carries with it a single bit saying whether that message arrived via cheap links only, or whether it traversed at least one expensive link. Note that it is easy for a host to determine whether a particular potential cluster neighbor is an actual current cluster neighbor, just by exchanging messages with that node and seeing whether the messages arrive via cheap links only.

As before, the algorithm will involve constructing a host parent graph, augmented by a gap-filling procedure. The algorithm will be similar to our first algorithm, in that our algorithm explicitly establishes the parent attachments, in a way involving comparison of the information at the child and parent nodes. As before, we insure in this way that no cycles get formed and that no child has an INFO set that is greater than that of any of its ancestors. Therefore, as in the first section, the gap-filling procedure must be restricted so that it only fills actual gaps, and never adds a data message larger than the node's largest.

Gap-filling could be done just along the tree, since our construction will insure that connectivity gets established if possible, even if the source dies. On the other hand, it might be useful for nodes to attempt to fill gaps for all their potential cluster neighbors, or maybe for all their current cluster neighbors. (In all cases, the gap-filling procedure is essentially the same — whatever the chosen set, a node occasionally informs the nodes in the chosen set of its INFO set. Any node can ask anyone for a data message if it knows from its MAP that the node has the data message. What can vary is exactly what is selected as the chosen set.)

Now we describe the procedure used to establish the host parent graph. We would like hosts to use comparison of INFO sets as before, attempting to attach to a parent with greater INFO. However, we would also like hosts to attach to hosts in the same cluster if possible, even in preference to attaching to a host in

another cluster with a much greater INFO set.

First, we claim that our previous attachment rule that requires a parent to have strictly greater INFO is inadequate. Consider the situation where an entire cluster is equally up-to-date, but all are rather out-of-date. (Perhaps no one is getting any new information from the source since they were all attached to a common parent in their own cluster who has just died.) Then new attachments should form. We would like all of these nodes to form a tree and only one of them to go outside the cluster looking for a new parent. However, if we use the strict inequality constraint, no attachments can occur among these nodes. Thus, we should loosen the constraint on attachment to allow some cases where the parent does not have strictly greater information than the child. We must be careful when doing this, however, since we want to avoid cycles. We use the following.

Constraint on attachment: A node i can only attach to a node j provided that one of the following holds.

1. j's INFO is strictly greater than i's INFO.
2. j's INFO is equal to i's INFO and j is a root.
3. j is already an ancestor of i (and remains an ancestor of i during the reattachment procedure).

It is easy to see that observing this constraint insures that the two invariants we want are maintained: that no cycles are formed, and that no child ever has INFO greater than that of any ancestor. (Sketch of proof: There is no way that a child could ever get INFO greater than an ancestor, since nodes only attach to parents with INFO at least as great as theirs, and data messages greater than a node's largest are only accepted from the node's parent. We argue that no cycles can occur. Assume that a cycle forms, and consider the step which inserts the last edge that completes the cycle. That last edge could not be inserted because of rule 1, because that would mean that a descendant had INFO greater than an ancestor. Likewise, that last edge could not be inserted because of rule 2, because the fact that a cycle is being completed implies that the new parent is not a root. Finally, the last edge could not be inserted because of rule 3, since if a new cycle is being formed, an old cycle would have already existed.)

Within this constraint, there is a lot of flexibility for deciding on appropriate parents. We give some specific rules.

It is useful for a node to know whether its parent is in the same cluster. This situation can be detected if parents continually send "I'm alive" messages to all children, when no data messages are being sent. It can be determined from the messages that they arrived over cheap links only, hence that the sender is in the same cluster as the receiver.

There is one general rule which seems useful to follow. Namely, if node i's parent and its grandparent are in the same cluster, then node i should reattach to its grandparent. This situation is detectable by node i's parent. (The reattachment will probably require some kind of 3-node "atomic transaction", in order to insure that constraint 3. above remains satisfied.)

There are several cases, in which different policies should be followed.

Case 1): Node i has a parent j which is in the same cluster as i.

In this case, the only way to improve i's situation is for i to find another node in the same cluster with greater information than i. (This is using Awerbuch's idea locally within the cluster.) Nodes should periodically inform their cluster neighbors of their INFO sets. (Actually, telling them the largest index is enough.) This is sufficient to allow i to determine appropriate candidates for a new parent.

Case 2): Node i has a parent j which is in a different cluster from i.

Then i should try several possibilities, in order of preference:

(a) First, i should look for a node in the same cluster as i but with more information than i. This can be done as in Case 1.

(b) Second, i should look for a node j in the same cluster as i with the same information as i, such that j is a root.

(c) If the first two simple tests fail to yield a parent from the same cluster, the third thing to try is the following. Node i should look for a node j in the same cluster as i, with as large INFO as possible (perhaps we should require that j's INFO be at least as great as i's INFO), such that j is not a "cheap descendant" of i in the parent graph.

Note: We say that j is a "cheap descendant" of i in the parent graph provided that there is a path directed upward from j to i in the parent graph which traverses only cluster neighbors.

Node i would like to attach to j, but since the constraint is not satisfied immediately, all i can do is to wait for a short while to see if the constraint becomes satisfied. Thus, i can buffer its input data messages for a short while to keep its own INFO from growing larger, and would check to see if j's INFO grows larger than i's INFO. If this happens in a short time, i will discard the buffered messages and attach to j. If not, i can process the buffered messages as usual.

The reason that i checks that j is not a cheap descendant is that it would be impossible for j's INFO to grow larger than i's during i's wait (above) if j were a descendant of i; it seems too expensive for i to actually test whether j is a descendant of i, so i makes a weaker test — whether j is a cheap descendant. This weaker test can be made economically, by searching the tree as long as only cheap descendants are encountered, searching for j.

Tests (a)-(c) are to be made frequently, since it is highly preferable for a node to attach to a node in the same cluster. However, if they fail, node i can still look (less frequently) for some further improvement among non-cluster neighbors. Thus, it makes the following tests.

(d) Node i looks for a non-cluster neighbor with greater INFO than i's.

Some discipline must be exercised in examining arbitrary non-cluster neighbors. It is possible, but possibly undesirable, to have non-cluster neighbors frequently informing each other of their INFO size. They could inform each other infrequently, with frequency perhaps depending on static distance. Or, they could inform upon request: node i could request INFO when this stage of this case is reached. Exactly whom i requests INFO from will depend on some information about the network.

Case 3): Node i has no parent.

This case can arise at initialization of the algorithm, or by timing out on a parent. (A node expects to receive some messages from its parent, either data messages or "I'm alive" reassurance messages.) Depending on the protocol used for

switching from an old parent to a new parent, this case could also arise after an old parent is detached, if the new attachment fails to complete.

Node i proceeds just as in case 2, with one addition. If all else fails, node i can settle for an arbitrary parent which has equal INFO to i and is a root.

This algorithm appears to have nice behavior; the given rules appear to cause gravitation to an acceptable parent graph if the algorithm is initialized with an arbitrary acyclic parent graph structure, and appear to cause proper adaptation to cluster partition and merge.

9.4 Combination Point-to-Point and Broadcast Server Networks: Clusters

It remains to be seen how to modify the preceding algorithm if the underlying server network consists of a combination of point-to-point links and broadcast links instead of just point-to-point links. We assume that the server network can take advantage of the broadcast subnetworks — when a host wants to broadcast a message to some set of hosts and the host's server knows that they are all on the same broadcast subnetwork, the server should make proper use of the broadcast subnetwork. Thus, if we just used the algorithm verbatim, partial benefit would be obtainable from the broadcast subnetworks.

In addition, it is desirable to try to have an explicit rule which attempts to configure the parent graph so that hosts which are part of a common broadcast subnetwork will be adjacent. Such a rule requires (static) information about which nodes are on a common broadcast subnetwork. Then a rule might be similar to one used in the first section: any node i which discovers an ancestor a with a child c such that i, a and c share a broadcast subnetwork, should reattach to a. Note that this reattachment should be done even if the connection between i and a is via an expensive link.

There are two problems with this rule. First, as before, it must be insured that either constraint (1) or constraint (3) above is satisfied, in order for i to reattach to a. It seems that the natural constraint to try to satisfy is (3), since a was discovered to be an ancestor of i. However, we need to insure that a is actually an ancestor of i at the moment of reattachment; this seems to require that the search for a and the reattachment be done as an atomic transaction.

Alternatively and perhaps more simply, we could allow i to wait as before for (1) to be satisfied.

Second, if we introduce this new rule, we get the possibility of "thrashing" of nodes reattaching to different parents. Specifically, assume i was previously attached to node p, where p and i are in the same cluster, and assume that i is to reattach to a in another cluster because of this new rule. Then immediately after reattaching, i might try to reattach once again to p, because of the rule in Case 2(c). (It is also conceivable, but unlikely, that i would want to reattach to p as a result of the rule in Case 2(a).) It seems that we might try to disable the rule in Case 2(c) in this case — i.e., where i has a parent a who has another child c for which i, a and c are on the same broadcast subnetwork, where i and c are in the same cluster, but a is in another cluster. However, that is not good enough, for there would be nothing to cause the pair, i and c, to reattach to a node in the same cluster if that turns out to be possible — each of i and c would be prevented from reattaching to a cluster neighbor by the fact that the other is also attached to a!

The idea that seems to work is that we can regard i and c as being temporarily "merged" into a single pseudo-node $\{i,c\}$. It is useless to reattach only one of this pair to something in its cluster, but it would be useful to reattach both of them to something in their own cluster at the same time. When one of them, say i, discovers a node p that it wants to attach to by one of rules 2(a)-(c), then before i does so, it enters into a protocol with c to insure that c also is able to attach to p; if this works out, then they both reattach. Note that it is not important that the entire reattachment protocol involving i, p and c be carried out atomically, since there is no terrible harm done if only one of i, c reattached.

10. NON-PROGRAMMABLE SERVERS WITH MULTICAST

In this case, the servers and switches are not programmable, but a good multicast facility is provided by the network. The unreliable multicast facility makes broadcasting the data messages simple and efficient; however, we must now ensure that any lost messages are eventually received by the hosts.

One of the first issues to resolve in this case is how hosts will detect missing data messages. Since the data messages are numbered by the source, a host can detect a gap, or missing message, as soon as it receives a message with a higher number. However, if the source is disconnected, or if it temporarily not broadcasting new messages, then the receiving host has no way of detecting the gap, i.e., of telling if the source is not broadcasting, or if the source is broadcasting and it has missed the data.

One way to solve the problem is to have each host periodically check with its "neighbors" to see what messages they have received. If indeed there are gaps, not only can they be detected, but they can also be filled in by the neighbor. This is the strategy used by the algorithms of the previous scenarios. Notice that every node must check, directly or indirectly, with every other node in the system for gaps. (If one node is missed, then that node could be the one with the information to detect and fill a gap.) In the algorithms of the previous sections, hosts do not check directly with every other host. Instead, hosts form a tree, and a host delegates the job of checking to its father.

A second way to solve the problem is to have the source broadcast a null data message if more than D seconds have gone by without a transmission. The null messages may or may not be sequenced, but for simplicity let us assume that they are numbered, just as if they were regular data messages. With this strategy, hosts can detect gaps on their own, without communicating with any neighbors. Specifically, a gap at data message n is recognized if any message with sequence number $n + 1$ or higher arrives, or if more than D seconds go by without the arrival of message $n + 1$.

With the second strategy, hosts do not have to send any messages out on a regular basis, only when a gap (i.e., a failure) occurs. Furthermore, the null messages use the efficient multicast facility. In the rest of this section we adopt this null message strategy.

In designing the reliable broadcast algorithm, it is important to note that there are really two different cases to consider:

(1) The first is filling in sporadic gaps, i.e., gaps that occur due to transient events in the system. In these cases, we expect both the number of missing

data messages and the number of hosts that missed them to be small. Also, occurrences of these gaps should be infrequent. Given these assumptions, the strategy we use for filling in sporadic gaps is not critical. We could simply have each node that detects a gap query every node in the system, of course, giving preference to nodes that are close or likely to have the desired information.

(2) The second case occurs when a group of hosts is disconnected from the source, i.e., when a network partition occurs. (Hosts in the partitioned group will timeout after D seconds.) In this case, it is not desirable to have every node trying to fill its gaps independently because this situation affects all nodes in the partition (in the same way), and the partition could last a long time. Furthermore, when the reconnection takes place, there may be a large number missed data messages to transmit, and this should not be done in a haphazardly fashion. Thus, in this case, it appears to be best to elect a "coordinator" (or root) for the group, and to have it probe the rest of the system for a reconnection. This coordinator should know the identity of all its member hosts. This way, when a connection is established, the list of hosts can be given to the host(s) that will supply the missed data so it can use the efficient multicast facility for dissemination the data.

Since case (2) seems to be more important, performance wise, let us look at it first. Should the nodes in a partition group simply elect a coordinator and communicate directly to it, or should they form a tree structure, as in our other algorithms? Studying the communication needs may help answer this question.

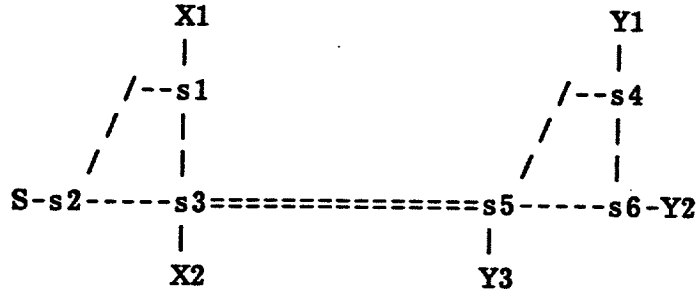
At the beginning of the partition, each node must inform the root or coordinator of its status (i.e., received data messages). Let us call this step 1. (One strategy for doing this will be discussed shortly.) Then the coordinator acknowledges this message, so that the host knows that the coordinator is indeed considering it a member of the group. A group id may be included in this acknowledgment to identify the group instance in the future. This is step 2. From then on, the coordinator sends out "I am alive messages" (with group id) to all members. This is step 3, and is repeated periodically.

Step 3 is executed frequently, and it can be performed efficiently if the coordinator uses the multicast facility to send the message to all members at once. Thus, from the point of view of efficiency of step 3, it seems to be best to have the coordinator have all members as direct children, i.e., a very bushy tree.

However, if everyone tries to communicate with the coordinator directly in step 1, especially immediately after a partition, the coordinator may become congested. Thus, for steps 1 and 2, it may be advisable to form a tree structure. This tree would only be used selecting the coordinator, but not for broadcasting the "I am alive" messages.

We now describe one possible way to perform steps 1 and 2. (We are still only considering partitions, not sporadic gap filling.) Each host is given a "priority list" that lists all hosts in order of increasing priority. For the time being assume that all hosts have the same list. For example, for the Mountain Example of Section 6 the list could be (lowest priority) Y2, Y1, Y3, X2, X1, S (highest priority). (For reference, the network for the Mountain Example is repeated below.) When a host X detects a disconnected source (timeout after D seconds), it tries to contact the node with the next highest priority after it. If it cannot be reached, it tries the next highest priority, and so on, until some node Y responds. Then X asks Y to carry on this selection process for it. If Y has the messages that were missed by X, then it gives them to X, and both X and Y return to normal operation. If Y does not have the messages, then it starts up the selection procedure by trying to reach nodes of higher priority. If Y was already doing this, then Y adds X to its list of "late comers" and continues the process.

The Mountain Example, Part I.



Eventually, the node in the partition group with the highest priority is unable to contact other nodes and declares itself coordinator. Then it broadcasts an acknowledgment message to all members (using multicast facility) giving id of all members, id of common gaps, and group id. Late comers can be added at any time after this, by having the late comer lists trickle up the tree, and new acknowledgements come down (either directly to the late comers, or in multicast mode).

Of course, the priority lists must be selected so that they produce good configurations. In the Mountain Example list, X2 follows Y3, since X2 is the best node for Y3 to contact first. If all nodes have the same list, the nodes form a linear structure. For example, in the Mountain Example, if s2 (i.e., S) fails, then Y2 would contact Y1, which in turn would contact Y3, and in turn would contact X2, and so on.

If we want both Y1 and Y2 to contact Y3 directly, we can "play games" with the priority lists. For example, let us make the priority lists as follows:

For Y2: Y2, Y3, Y1, X2, S, X1

For Y1: Y2, Y1, Y3, X2, S, X1

For Y3: Y2, Y1, Y3, X2, S, X1

For X2: Y2, Y1, Y3, X2, S, X1

for X1: Y2, Y1, Y3, X2, X1, S

As before, each node only searches for nodes of higher priority. For instance, node Y1 does not contact Y2; only nodes Y3, X2,..., in that order. With these lists, after S fails, both Y1 and Y2 become children of Y3 However, if Y3 is also unavailable, then Y2 becomes a child of Y1, and Y1 is the one that tries to reach X2.

Of course, we must ensure that these lists do not allow cycles. That is, the relationship "node x could become a child of y" should be acyclic. In the example, if we form a graph with arcs from Y2 to its potential coordinators (Y3, Y1, ..., X1), from Y1 to its potential coordinators (Y3, X2, S, X1), and so on, we see that the graph has no cycles.

Note that the priority lists we have suggested are *static*. Therefore the trees that are created from them may not be the best in all cases. If in the Mountain Example, suddenly a new high bandwidth link appears between hosts X1 and Y1, the polling structure will not take full advantage of it. However, the structures we are creating are used infrequently for failure recovery, not frequently for normal message broadcast. Thus, we believe the structures will be satisfactory. Observe that we avoided this type of static structure in previous sections where the trees were used for normal broadcasts.

Returning to the sporadic gap problem, we could also use the idea of priority lists in that case (although as we mentioned earlier, whether we do something fancy here or not is not as critical). Temporarily assume that the gap filling algorithm is independent from the partition recovery algorithm we have just described. Suppose that we have the same lists given above and that Y2 detects a gap at data message n. Y2 would first check with Y3 (Y3 follows Y2 in the Y2 list). If Y3 has n, then it would give it to Y2 and that would be it. If not, then Y3 takes up the job of finding message n.

What we have here is exactly the same procedure we had described earlier. Thus, both the gap filling and partition recovery algorithms can be combined into one. A coordinator is set up whenever a gap, of any type, is found. If the gap is sporadic, chances are the procedure will be terminated before the coordinator is selected because the gap is filled. If the gap is caused by a partition, the coordinator will be selected.

In all cases, remember that the coordinator must probe *all* nodes in the system, including nodes in its group. If the coordinator reaches a node with higher priority but without the desired data, the coordinator search process is resumed: the old coordinator asks this new node to become coordinator or to continue the search. If the coordinator reaches a node with lower priority but without the desired data, it does nothing. (The coordinator may request that this node inform it of any changes in its status. This way the coordinator will not have to probe it periodically.)

When a coordinator finds some or all of the missing data messages, it instructs the node that has them to broadcast the data to all of its members. In

some cases the coordinator could find two or more nodes with the data, and in this case it may decided to split the request. That is, each node with the data could be instructed to broadcast to a subset of the members, i.e., those members that it can communicate with easily. Once the data is transmitted, the coordinator can return to normal operation.

We have sketched one possible solution for the reliable broadcast problem when a multicast facility is available. Clearly, many details have been left out, and several other variations are possible. The most important observation to make is that the structures needed by the algorithm (i.e., the tree to form a group and select a coordinator) is not used for significant message traffic. Thus, the structure need not adapt to changing loads, only to failures. Thus, a simple fixed strategy like priority lists seems to be adequate.

REFERENCES.

[Awer84] Awerbuch, B., and Even, S. "Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network," *Proc. Symp. Principles of Distributed Computing*, 1984, 278-281.

[Blau83] Blaustein, B., Garcia-Molina, H., Ries, D., Chilenskas, M., and Kaufman, C. "Maintaining Replicated Databases Even in the Presence of Network Partitions," *Proc. IEEE Electronics and Aerospace Conference (EASCON 83)*, September 1983.

[Blau85] Blaustein, B.T., and Kaufman, C.W. "Updating Replicated Data During Communications Failures," *Proc. Eleventh Int. Conf. Very Large Data Bases*, August 1985, 49-58.

[Dala78] Dalal, Y.K., and Metcalfe, R.M. "Reverse Path Forwarding of Broadcast Packets," *Comm. ACM 21*, 12 (December 1978), 1040-1048.

[Garc83] Garcia-Molina, H., Allen, T., Blaustein, B., Chilenskas, M., Ries, D. "Data-patch: Integrating Inconsistent Copies of a Database after a Partition," *Proc. Third Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

[Rose80] Rosen, E.C. "The Updating Protocol of Arpanet's New Routing Algorithm," *Computer Networks 4*, 1 (February 1980), 11-19. (Also in Proc. Fourth Berkeley Workshop on Distributed Data Management and Computer Networks, August 1979.)

[Sari85] Sarin, S.K., Blaustein, B.T., and Kaufman, C.W. "System Architecture for Partition-Tolerant Distributed Databases," *IEEE Trans. Computers C-34*, 12 (December 1985), 1158-1163.