FAST RELIABLE BROADCASTS*

Fred B. Schneider
David Gries
Richard D. Schlichting+

TR 82-519
September 1982

Department of Computer Science
405 Upson Hall
Cornell University
Ithaca, New York 14853

*This work is supported by NSF Grants MCS 76-22360 and MCS 81-03605.

+Department of Computer Science, University of Arizona, Tucson, Arizona 85721.

Fast Reliable Broadcasts*

Fred B. Schneider
David Gries
Richard D. Schlichting+

Department of Computer Science
Cornell University
Ithaca, New York 14853

August 31, 1980
Revisions: September 3, 1982

ABSTRACT

A distributed program is presented that ensures delivery of a message to the functioning processors in a computer network, despite the fact that processors may fail at any time. All processor failures are assumed to be detected and to result in halting the offending processor. A reliable communications network is also assumed. Broadcast strategies that minimize the time to completion for a broadcast are defined.

---

*This work is supported by NSF Grants MCS 76-22360 and MCS 81-03605.

+ Department of Computer Science, University of Arizona, Tucson, Arizona 85721.

1. Introduction

A *reliable broadcast protocol* is a distributed program that ensures delivery of a message to the functioning processors in a computer network, despite the fact that processors may fail at any time. Fast reliable broadcast protocols have application in a wide variety of distributed programming problems [S80] [S82].

Broadcast networks –e.g. contention networks such as Ethernet [M76] and ring networks like DCS [F73]– would appear to implement reliable broadcast protocols directly in hardware, but do not [LL79]. In these networks, each processor is connected to a *network interface unit*. This unit monitors the network and copies messages identified with its address code into a buffer memory, which can be accessed by a connected processor. Unfortunately, there is no guarantee that a processor will receive every message addressed to it. For example,

(1) the buffer memory might be full when a message is received by the interface unit,

(2) the interface unit might not be monitoring the network at the time the message is delivered, or

(3) in a contention network, an undetected collision that affects only certain network interface units could cause them to miss a message.

Thus, while broadcast networks allow messages to be broadcast, they do not directly support reliable broadcasts. This is not to say that hardware that directly implements a reliable broadcast protocol cannot be constructed; merely that it has not been.

In point-to-point networks, in which a message sent can be received by only one processor, there are other impediments to implementing reliable broadcast protocols. If each processor sends at most one message per broadcast, then time linear in the number of processors is required, often an unacceptable delay for the completion of a broadcast. If each processor sends more than one message per broadcast, broadcasting is not an atomic action with respect to failures. Consequently, such protocols require a scheme in which processor failure causes another processor to assume its duties. Such a scheme, which can be subtle, is one of the contributions of this paper.

The paper is organized as follows. In Section 2, assumptions about the communications network and processor failures are discussed and the notion of a broadcast strategy is formalized. In

Section 3, a reliable broadcast protocol that will work with any broadcast strategy is derived. In Section 4, a way to generate broadcast strategies that minimize the length of time to complete a broadcast is presented. Section 5 discusses some implications of our work.

2. The Environment

2.1. Communications

Consider a network containing N processors, named 1, ..., N. We assume the

Reliable Communications Property: Each functioning processor can always communicate, directly or indirectly, with every other functioning processor.

Clearly, to withstand up to k failures, there must be k independent paths between any two processors. These paths may be direct or may involve relaying messages through other processors. Although achieving the Reliable Communications Property is likely to be expensive, it is impossible to distribute a message to every functioning processor if there is no way to communicate with some of them. Implementations that approximate the Reliable Communications Property do exist: the ARPANET and many local-area networks [N79] exhibit it most of the time.

Assuming the Reliable Communications Property does not trivialize the problem of designing a reliable broadcast protocol; it is still necessary to design the protocol so that failure of a processor will result in some other processor assuming its message-forwarding duties.

Processors communicate by exchanging *messages* and *acknowledgements*. Each message m contains the following information:

$m.sender \equiv$ the name of the processor that sent m.

$m.info \equiv$ the information being broadcast.

$m.seqno \equiv$ a sequence number assigned to the message by the processor b that initiates the broadcast. The first message broadcast has sequence number 1.

Execution of

$p!!msg(expr, s)$

by processor q sends a message m to processor p with $m.sender = q$, $m.info = expr$ and

$m.seqno = s$. If m is a message, execution of $p!!msg(m)$ by q sends m to p with the same $info$ and $seqno$ fields but with $sender$ being q. In either case, execution of !! does not delay q.

Execution of

$??msg(m)$

by a processor delays that processor until a message is delivered; then that message is stored in variable m.

Execution of $p!!ack(m)$ and $??ack(m)$ are used to send and receive acknowledgements. Their operation is similar to that of $p!!msg(m)$ and $??msg(m)$, the only difference being the identifying ack instead of msg.

This notation is inspired by the input and output commands of CSP [H78]. Two shrieks (!!) and queries ($??$) are used, instead of one, to indicate that messages are buffered and therefore no synchronization takes place. Also, in contrast to CSP, the sender names the receiver but the receiver does not name the sender.

2.2. Processor Failures

We assume a restricted type of processor failure.

Processor Failure: A processor has *failed* if it has stopped executing. Messages delivered to it may be lost.

The case where a processor continues executing, although not in a manner defined by its program, is not considered here. However, as long as the erroneous operation does not cause transmission of acknowledgments, the integrity of the protocol developed in Section 3 will not be affected –execution will merely be slowed until the failure is detected, at which time the protocol will continue.

Some mechanism for detecting processor failures is assumed, which is abstracted for use as a predicate $failed(p)$:

$failed(p)$ $\equiv$ "processor p has failed"

This mechanism can be implemented using time-outs.

2.3. Broadcast Strategies

A *broadcast strategy* describes how a message being broadcast is to be disseminated to the processors in the network. We represent a broadcast strategy by a rooted, ordered tree in which the root corresponds to the processor originating the broadcast, other nodes correspond to the other processors, and there is an edge from p to q if processor p should forward to processor q the message being broadcast.[1] When a node has more than one successor in the tree, the message is forwarded to each of the successors in a predefined order, also specified by the broadcast strategy.

Given a broadcast strategy represented by graph (V, E), we define the relation

$$p \ SUCC \ q \equiv pq \in E$$

$SUCC^+$ and $SUCC^*$ are the conventional transitive closure and reflexive transitive closure of relation $SUCC$. We use the name of a relation to denote a set: $SUCC(P)$ is the set of successors of the elements of the set P, and similarly for $SUCC^+$ and $SUCC^*$.

A broadcast strategy describes a *preferred* method of broadcasting: as long as no processors fail, messages are disseminated as prescribed by the broadcast strategy. However, processor failure may require deviation from the strategy.

The broadcast strategy to employ in a given situation depends on what is to be optimized. For example, in Section 4 a broadcast strategy is given that minimizes the length of time it takes until all processors receive the message. Use of broadcast strategies that can be represented by a subgraph of the processor interconnection graph seems reasonable, since it minimizes message relaying, but it is not required.

Two common broadcast strategies are the "bush" of Figure 2.1a and the "chain" of Figure 2.1b. In some sense, these are the limiting cases of the continuum of broadcast strategies. A more complex broadcast strategy is shown in Figure 2.1c.

---

[1] Restriction to trees is not a limitation when considering broadcast strategies that ensure minimum time to completion. A broadcast strategy that cannot be represented as a tree must include a processor that receives the same message more than once.

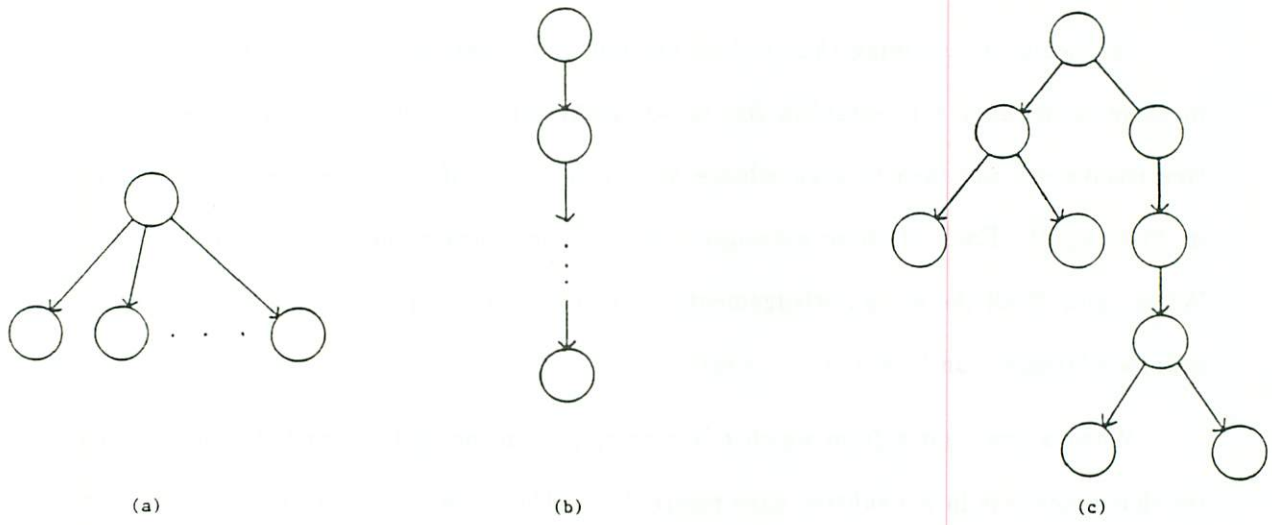

Figure 2.1 – Some Broadcast Strategies

3. Reliable Broadcasts with Unreliable Processors

We now present a reliable broadcast protocol for any broadcast strategy represented by an ordered tree with root b. A copy of the protocol runs at each processor, the copy for processor b being slightly different because broadcasts are initiated there.

The broadcast of message m originating at b is completed when every functioning processor has received a copy of m. Thus, the distributed program establishes $B(b,m)$ where:

$B(j,m) \equiv (\forall\ p\colon p \in SUCC^*(\{j\})\colon failed(p) \vee rec(p,m))$

and $rec(p,m) \equiv$ "processor p has received message m"

However, $B(b,m)$ may not remain true once it has been established, if failed processors restart. To avoid this problem, we postulate temporarily that once a processor has failed it remains in

that state forever. We return to this problem in Section 3.3, where we devise a processor restart protocol.

3.1. Establishing B(i,m) at Processor i

We begin by assuming that b does not fail, but others may. When processor i receives a message m, its duty is to establish $B(i,m)$ –to make sure that all functioning members of its subtree receive m– and then to acknowledge m. Upon receipt of m, i relays m to every processor p in $SUCC(\{i\})$. Each of these establishes $B(p,m)$ and then returns an acknowledgement to i. When, (and if) all these acknowledgements are received by i, $B(i,m)$ has been established and an acknowledgement can be sent to $m.sender$.

When a processor p from which i is expecting an acknowledgement fails, there is no guarantee that processors in p's subtree have received m. Therefore, upon detecting that p has failed, i sends m to all processors in $SUCC(\{p\})$ and waits for acknowledgements from these processors instead of from p.

The protocol executed at processor i to establish $B(i,m)$ uses two set-valued variables:

$sendto$ $\equiv$ the set of processors to which m must be sent;
$ackfrom$ $\equiv$ the set of processors from which acknowledgements for m are awaited.

$B(i,m)$ is weakened to obtain the following invariant for a loop that will execute at processor i:

$$I \equiv (\forall\, p: p \in SUCC^{+}(\{i\}): p \in SUCC^{*}(sendto \cup ackfrom) \vee B(p,m) \vee failed(p))$$

Note that if i has not failed then

$$rec(i,m) \wedge I \wedge sendto = \Phi \wedge ackfrom = \Phi \;\Rightarrow\; B(i,m)$$

Thus, a loop of the following form may suffice to establish $B(i,m)$:

$??m;\ \{rec(i,m)\}$
$sendto,\ ackfrom := SUCC(\{i\}), \Phi;$
$\{rec(i,m) \wedge I\}$
do $sendto \neq \Phi \;\rightarrow\; \ldots \{rec(i,m) \wedge I\}$
 $[]\ ackfrom \neq \Phi \;\rightarrow\; \ldots \{rec(i,m) \wedge I\}$
od
$\{B(i,m)\}$

Filling in the bodies of the guarded commands is fairly simple. If $sendto \neq \Phi$, m is sent to a processor $dest$ in $sendto$, $dest$ is deleted from $sendto$, and $dest$ is added to $ackfrom$ to reestablish I.

If $ackfrom \neq \Phi$, at least one processor p has not yet returned an acknowledgement for m. When such an acknowledgement is delivered to i it can be received and p deleted from $ackfrom$. In order to do this and leave I invariant, it is sufficient for p to establish $B(p,m)$ prior to sending such an acknowledgement, since

$$I \wedge B(p,m) \Rightarrow wp(\text{"}ackfrom := ackfrom - \{p\}\text{"}, I).$$

Thus, the following proof obligation must be satisfied by the protocol:

> *Acknowledgment Precondition Obligation*: The precondition for processor p to send an acknowledgement for message m is $B(p,m)$.

Finally, since there is no point in waiting for an acknowledgement from a failed processor, such a processor can be removed from $ackfrom$. However, in order to reestablish I, its successors in the tree are added to $sendto$.

This results in the loop of Figure 3.1, where $FAILED$ is a set of processors with characteristic function $failed$[2] and operation $choose(sendto, dest)$ stores an arbitrary element of set $sendto$ into $dest$.[3]

3.2. Establishing B(b,m) When b May Fail

The protocol just described works correctly if processor b does not fail. We now investigate the complications that arise when b may fail. We also discuss termination and absence of deadlock, which were omitted in the simpler case.

Upon receiving a message m, processor i operates as above, and, provided b does not fail, $B(b,m)$ will be established by b. If b fails then some other processor that received m must estab-

---

[2]This use of set notation is for conciseness; an actual implementation would be in terms of *failed*.

[3]The selection of the element depends on the ordering on the selection of the successor of a given node. This is defined by the broadcast strategy being used.

```
??msg(m);
{rec(i,m)}
sendto, ackfrom:= SUCC({i}), Φ;   {I}

do sendto ≠ Φ   → choose(sendto,dest);
                  dest!!msg(m);
                  sendto:= sendto − {dest};
                  ackfrom:= ackfrom ∪ {dest}

 [] ackfrom ≠ Φ → if ??ack(a)  → ackfrom:= ackfrom − {a.sender};
                   [] ackfrom ∩ FAILED ≠ Φ → t:= ackfrom ∩ FAILED;
                                             sendto:= sendto ∪ SUCC(t);
                                             ackfrom:= ackfrom − t
                                             {I}

                   [] otherwise → skip
                   fi
od
{I ∧ sendto = Φ ∧ ackto = Φ ∧ rec(i,m)}
{Hence: B(i,m)}
```

Figure 3.1 – Protocol if b does not fail

lish $B(b,m)$. (If every processor that received m has failed, then $B(b,m)$ is trivially true.) Since no harm is done if $B(b,m)$ is established by more than one processor, we allow more than one to attempt to establish it. However, this means that i may receive more than one copy of m, each corresponding to a request for i to establish $B(i,m)$ and respond with an acknowledgement. In order to be able to send these acknowledgements, processor i maintains a set *ackto* of processors to which acknowledgements must be sent. Thus, three set-valued variables are used:

sendto ≡ the set of processors to which m must be sent;
ackfrom ≡ the set of processors from which acknowledgements for m are awaited;
ackto ≡ the set of processors that sent m to i for which acknowledgements must be returned.

After receiving m, process i must monitor b until *failed*(b) becomes true or $B(b,m)$ is known to be established. Therefore, some means must be found to notify processes that $B(b,m)$ has been established. But performing this notification is equivalent to performing a reliable broadcast! The way out of this dilemma is to use the sequence number *m.seqno* in each message m and require the

Broadcast Sequencing Restriction: Processor b does not initiate a broadcast until its previous broadcast has been completed.

Thus, receipt of a message m' with $m'.seqno > m.seqno$ means that the broadcast of m is completed. Consequently, b can notify processes of completion of a broadcast simply by initiating the next one.

Upon receipt of a message m, processor i establishes $B(i,m)$ and acknowledges m. Thereafter, i monitors b and, if b fails, i attempts to establish $B(b,m)$. Variable r (for root) contains either i or b, depending on whether i is attempting to establish $B(i,m)$ or $B(b,m)$.

With this initial discussion, we can now describe the invariant of the loop that makes up the protocol, which is given in Figure 3.2. This invariant will be used to argue the partial correctness of the protocol, that progress is made during execution of the protocol, and that no deadlock occurs. As each part is given, the reader should verify that it is indeed an invariant, using the accompanying discussion and the fact that previously discussed parts are already known to be invariantly true. Note that, when necessary, a subscript on a variable is used to denote the processor to which it belongs. For example, $ackto_p$ is the instance of *ackto* on processor p.

Initially, each processor sets m to an empty message with sequence number 0. We pretend that this empty message has been sent by b so that *P1*, given below, is initially true at each processor i. Once true, $rec(i,m)$ cannot be falsified, for receiving a new message m would keep it true. Hence, *P1* is invariantly true:

P1: $rec(i,m)$

Next, consider the sets *sendto* and *ackfrom*. These are always subsets of the set of nodes of the subtree rooted at r for which processor i is attempting to establish $B(r,m)$. Initially $r = i$, but after i establishes $B(i,m)$ it may set r to b and attempt to establish $B(b,m)$, if it detects that b has failed. *P2* is initially true because *sendto* and *ackfrom* are empty.

P2: $((r = i \wedge sendto \cup ackfrom \subseteq SUCC^+(\{i\})) \vee$
$(r = b \wedge sendto \cup ackfrom \subseteq SUCC^*(\{b\}) - SUCC^+(\{i\}))$

Invariant *P3* describes even more about *sendto* and *ackfrom*: if a node p is in *sendto* or *ackfrom*, then none of its descendants are. This follows from the nature of a tree and the operations performed on the two sets.

P3: no descendant or ancestor of a node in $sendto \cup ackfrom$ is in $sendto \cup ackfrom$

P4 is initially true because no message has a sequence number less than 0. It remains true because of the Broadcast Sequencing Restriction. Later, in describing the protocol at b, we must be sure that $B(b,m)$ is true before b broadcasts another message.

P4: $(\forall\, m' : m'$ broadcast by $b \wedge m'.seqno < m.seqno : B(b,m'))$

P5 indicates that any successor p of node i is in one of four categories: m should be sent to p by i, or m has been sent to p by i but it has not returned an acknowledgement, or it has received and acknowledged m and $B(p,m)$ is true, or it has failed. *P5* is initially true because we assume every processor has received the empty message with sequence number 0. Verifying that each guarded command leaves *P5* true is fairly easy, except for the command with guard *??ack*(a). Here, the sender of the acknowledgement is deleted from *ackfrom*. In order to maintain the truth of *P5*, we require the previously given Acknowledgment Precondition Obligation: $B(p,m)$ must be a precondition for p to send an acknowledgement for m. Note that in Figure 3.2 $B(i,m)$ is explicitly given as the precondition of each acknowledgement that i sends.

P5: $r=b \vee (\forall p : p \in SUCC^+(\{i\}) : p \in SUCC^*(sendto \cup ackfrom) \vee B(p,m) \vee failed(p))$

P6 is true initially because $r = i$. It may be falsified by changing r (to b), but this is done only when $ackfrom = \Phi \wedge sendto = \Phi$, which together with *P5* and the fact that i has not failed implies $B(i,m)$. It can be also falsified by falsifying $B(i,m)$, but this is done only by setting m to a new message, and when this is done r is changed to i.

P6: $r = i \vee (B(i,m) \wedge failed(b))$

Whenever $r \neq i$, processor i must attempt to establish $B(b,m)$. To do so, i ensures that *every* processor either (1) is in $SUCC^*(sendto)$, (2) is in $SUCC^*(ackfrom)$, (3) has established $B(p,m)$, or (4) has failed.

$$P7\text{: } r = i \;\vee\; (\forall p : p \in SUCC^*(\{b\}) : p \in SUCC^*(sendto \cup ackfrom) \vee B(p,m) \vee failed(p))$$

$P7$ is initially true because, by convention, all processes have received the empty message with sequence number 0. The one tricky case concerns the first guarded command of the loop, where i is deleted from *sendto*. This does not falsify $P7$, for if $r \neq i$ then by $P6$ $B(i,m)$ holds, and thus $B(p,m)$ holds for all p in the subtree rooted by i.

Finally, we describe the set *ackfrom* a bit more precisely.

$P8$: $q \in ackfrom_p$ = m is in transit from p to q $\vee$
$p \in ackto_q$ $\vee$
an acknowledgement for m is in transit from q to p

Note that *ackto* for processor b is always empty, because no processor ever sends a message to b.

Total Correctness

Suppose processor b sets its local variable m to a new message $\overline{m}$ to be broadcast and stores $SUCC(\{b\})$ in *sendto*. We want to argue that, after a finite amount of time, $B(b,\overline{m})$ holds. First, note that each processor p sends $\overline{m}$ to each other processor q at most once and receives at most one acknowledgement from each processor for it. This is due to invariant $P3$ and the way *sendto* and *ackfrom* are changed: $\overline{m}$ is sent to q only if $q \in sendto$, and upon sending $\overline{m}$ to q it is deleted from *sendto*, never to be placed there again. This places an upper bound of $2N(N-1)$ on the number of messages and acknowledgements sent to accomplish the broadcast of $\overline{m}$.

Define,

$Rmsg\#(m) \equiv$ *total number of times m has been received*;
$Sack\#(m) \equiv$ *total number of times an acknowledgement for m has been sent*;
$Rack\#(m) \equiv$ *total number of times an acknowledgement for m has been received*;

and the following 8-tuple, whose values are always non-negative and bounded from above:

```
m:= (sender: b, info: '', seqno: 0);
ackto, sendto, ackfrom:= Φ, Φ, Φ;
r:= i;

do sendto≠Φ → choose(sendto, dest);
               sendto:= sendto − {dest};
               if dest=i → skip
                [] dest≠i ∧ failed(dest) → sendto:= sendto ∪ SUCC({dest})
                [] dest≠i ∧ ¬failed(dest) → ackfrom:= ackfrom ∪ {dest};
                                            dest!!msg(m)
               fi

 [] ackfrom ∩ FAILED≠Φ → t:= ackfrom ∩ FAILED;
                         sendto:= sendto ∪ SUCC(t);
                         ackfrom:= ackfrom − t;

 [] ??ack(a) → if a.seqno=m.seqno → {B(a.sender,m)}
                                    ackfrom:= ackfrom − {a.sender}
                [] a.seqno<m.seqno → skip
               fi

 [] failed(b) ∧ r≠b ∧ ackfrom=Φ ∧ sendto=Φ → {B(i,m)}
                                             r, sendto:= b, {b};

 [] ??msg(new) → if new.seqno=m.seqno → ackto:= ackto ∪ {new.sender}
                  [] new.seqno<m.seqno → {B(i,new)} new.sender!!ack(new)
                  [] new.seqno>m.seqno → {B(b,m), hence B(i,m)}
                                         (∀p: p∈ackto: p!!ack(m));
                                         m, r:= new, i;
                                         ackto:= {m.sender};
                                         sendto:= SUCC({i});
                                         ackfrom:= Φ
                 fi

 [] ackto≠Φ ∧ (r=b ∨ (sendto=Φ ∧ ackfrom=Φ)) → {B(i,m)}
                                               (∀p: p∈ackto: p!!ack(m));
                                               ackto:= Φ
od
```

For processor b, the guarded command beginning with $failed(b)$ is replaced by the following guarded command:

```
ackfrom = Φ ∧ sendto = Φ → {B(b,m)}
            Delay until a new message is ready to be broadcast;
            Initiate a new broadcast:
                m:= (sender: b, seqno: next sequence number, info: new message);
                sendto:= SUCC({b})
```

Figure 3.2 – Reliable Broadcast Protocol

$<3N(N-1)(\overline{m}.seqno-1) - (\Sigma m:\ m.seqno < \overline{m}.seqno:\ Rmsg\#(m) + Sack\#(m) + Rack\#(m)),$
$(\mathbf{N}p:\ \neg failed(p)),$
$(\mathbf{N}p:\ \neg failed(p):\ \neg rec(p,\overline{m})),$
$(\mathbf{N}p:\ \neg failed(p):\ \neg rec(p,\ \overline{m}) \vee (rec(p,\ \overline{m}) \wedge r_p = p),$
$(\Sigma p:\ \neg failed(p) \wedge rec(p,\ \overline{m}):\ |\ SUCC^*(sendto_p) \cup SUCC^+(ackfrom_p)\ |),$
$(\Sigma p:\ \neg failed(p) \wedge rec(p,\ \overline{m}):\ |\ SUCC^*(ackfrom_p)\ |),$
$(\mathbf{N}m:\ m.seqno = \overline{m}.seqno:\ m$ a message in transit),
$(\Sigma p:\ \neg failed(p):\ |\ ackto_p\ |) >$

Consider the value of this 8-tuple just after b has set its local variables m and $sendto$ to $\overline{m}$ and $SUCC(\{b\})$, respectively. By inspection, one can see that, with one exception, each processor failure and each iteration of the loop by any processor lexicographically decreases the 8-tuple. For example, receipt of $\overline{m}$ by p leaves the first two components the same but decreases either the third or sixth component.

The one exception to decreasing the 8-tuple is initiation of a new broadcast by b, which will occur only when $B(b,\overline{m})$ is true. Assume that b performs no broadcast after $\overline{m}$. Then, since the 8-tuple is bounded below and decreases with each iteration, after a finite amount of time no further iteration can occur and each processor is delayed. We will show that this delay implies that $sendto = \Phi$ and $ackfrom = \Phi$ for each processor, which with *P1* implies that $B(i,\overline{m})$ holds for each processor i. Hence, all processors have received $\overline{m}$ and are waiting for another broadcast.

Assuming that all messages in transit have reached their destination, that no further failure occurs, and that all processors are delayed, inspection of the guards of the loop of the protocol yields the following for each functioning processor:

(1) $sendto = \Phi$
(2) $ackfrom \cap FAILED = \Phi$
(3) no acknowledgement is in transit
(4) $\neg failed(b) \vee r = b \vee ackfrom \neq \Phi$
(5) no message is in transit
(6) $ackto = \Phi \vee (r \neq b \wedge ackfrom \neq \Phi)$

Suppose some processor p has $ackfrom \neq \Phi$, i.e. some q is in $ackfrom_p$. By *P8*, (3) and (5), $p \in ackto_q$. Since $ackto_q \neq \Phi$, this means that $q \neq b$, and from (6) we conclude that $r_q \neq b$ and $ackfrom_q \neq \Phi$. Further, by *P2* we conclude that

$r_q = q$ and $\Phi \neq ackfrom_q \subseteq SUCC^+(\{q\})$

Repeating this argument, we find that some descendent $q1$ of q also satisfies $\Phi \neq ackfrom_{q1} \subseteq SUCC^+(\{q\})$, some descendant $q2$ of $q1$ will satisfy the same property, and so forth, indefinitely. This leads to a contradiction because the broadcast strategy is a finite tree. Hence, all processors have empty *ackfrom* sets and $B(i,\overline{m})$ holds at every processor.

An Optimization

As it now stands, each processor monitors b. However, if b fails before a broadcast that it initiated has completed then either (1) no functioning processor has received the message or (2) some running processor has received the message from a processor that has failed. Thus,

$$failed(b) \Rightarrow B(b,m) \vee (\exists p: \neg failed(p) \wedge rec(p,m): failed(sender(m)))$$

This allows "$failed(b)$" to be replaced by "$failed(m.sender)$" in the protocol above. Now, every processor need not monitor b; each need only monitor a processor it is communicating with (e.g. its father). However, it is now possible that more than one processor will attempt to establish $B(b,m)$, even if b does not fail.

3.3. Processor Restarts

The restriction that a failed processor remains halted can now be relaxed. A processor is *restarted* after the cause of its failure has been identified and corrected. Once a processor i has been restarted, it executes a *restart protocol*, during which $B(b,m)$ is reestablished for each message m broadcast by b.

The restart protocol establishes

$$(\forall\ m:\ m\ broadcast\ by\ b:\ B(b,m))$$

which is equivalent to

$$(\forall\ m:\ m\ broadcast\ by\ b\ \wedge\ \neg\ rec(i,m):\ B(b,m)) \wedge (\forall\ m{:}m\ broadcast\ by\ b\ \wedge\ rec(i,m):\ B(b,m)).$$

This suggests that the protocol involve two steps:

(1) The first conjunct is established by having some functioning processor p send to i a copy of every message that was broadcast by b that p received. Naturally, this requires that these messages be stored someplace.

(2) Then, the second conjunct is established. To do this, processor i initiates a broadcast of each message m it has received that was broadcast by b and not forwarded to i during step (1) of the restart protocol. This is necessary because all the processors that had received m might have failed; if i is the first of these to be restarted, it must broadcast m.

Optimization of step (1) is possible if information about messages received is retained across processor failures.

4. Minimum Time Broadcasts

We now turn attention to development of a broadcast strategy that minimizes the time required to complete a broadcast provided no processor fails. When used in conjunction with the reliable broadcast protocol developed in the preceding section, fast reliable broadcasts are achieved, although if processors fail then broadcast completion time is not necessarily minimized.

The broadcast strategy we develop is based on the following assumptions:

(1) All processors execute at approximately the same speed.

(2) The delay involved in sending a message between every pair of processors is equal and constant.

(3) The communications network has sufficient buffer capacity.[4]

These assumptions are approximated by local computer networks made up of a homogeneous collection of processors.

In order to develop a minimum-time broadcast strategy, the relationship between processor execution speed and communications network delay must be quantified. We therefore define

D: the delay associated with delivery of a message between two processors, and

E: the time that must elapse after a message is sent by a given processor as part of the broadcast, before that processor can send another message as part of the same broadcast.

D is determined by the performance characteristics of the communications network; E is related to processor execution speed, the processing allocated for dealing with broadcasts, and the number of broadcasts in which the processor can participate at any given time. Without loss of general-

[4]The required buffer capacity is derived below.

ity, assume E and D are integers.

The following is a broadcast strategy that minimizes completion time:

> S: A processor relays message m immediately upon receiving it and every E seconds thereafter, as long as some processor has not been sent the message.

Depending on the values of D and E, this strategy can lead to different broadcast strategy trees. For given values of D and E, let $R_{DE}(t)$ be the number of processors that receive m at time t. Assuming that the broadcast is initiated at time 0 and (for the moment) that there is an infinite number of processors, then no processor will have received m prior to time 0; one processor (the root) receives the m at time 0; and the number of processors that receive m at time t depends on D, when other processors received the message, and how quickly they relay it according to S above. Thus,

$$(4.1) \quad R_{DE}(t) = \begin{cases} 0 & \text{if } t < 0, \\ 1 & \text{if } t = 0, \\ \sum_{j=0}^{ceil\lfloor (t-D)/E \rfloor} R_{DE}(t-D-jE) & \text{if } t > 0, \end{cases}$$

which simplifies to

$$R_{DE}(t) = R_{DE}(t-D) + R_{DE}(t-E) \qquad \text{for t} > \text{E}.$$

Let $T_{DE}(t)$ be the number of processors that receive the message as of time t. Clearly,

$$T_{DE}(t) = \sum_{i=0}^{t} R_{DE}(i) .$$

Given a network with N processors, t_B, the elapsed time to perform a broadcast, is the smallest integer such that $T_{DE}(t_B) \geq N$. (This means that t_B is $O(logN)$, where the base of the logarithm depends on D and E.) The required buffer capacity at any given time is determined by the number of messages in transit, i.e. the number of messages that will be delivered up to D time units in the future. Because $R_{DE}(t)$ is monotonic, the heaviest demands for buffering are always made towards the end of a broadcast. Consequently, the maximum buffer capacity C required for that broadcast is computed as follows.

$$C = \sum_{i=0}^{D-1} R_{DE}(t_B - i)$$

For given D and E and root b, it is possible to precompute the broadcast strategy tree for the strategy described by S. A more desirable approach, however, is for each processor to compute *as needed* the names of processors to which it must send messages. Then, it is unnecessary for a processor to store numerous broadcast strategy trees. Such a scheme is outline below.

Each processor is assumed to have a unique name. For each broadcast strategy in which a processor participates, a (possibly different) unique *relative name* is assigned to it, such that the relative name of b is 1. In addition, each processor is assigned a two-component *logical name* of the form $<t_r, id>$, which is computed based on that processor's role in the broadcast. The first component t_r partitions processors into equivalence classes based on when they received the message, hence it groups together processors that will transmit messages at the same time. The second components of the logical names form a consecutive numbering of all processors that have the same first component. In the following, computation of processor names for a single broadcast that originates at time 0 is described. Also, for the time being we assume that processors do not fail.

The logical name of a processor p is determined from the logical name of the processor that sent p the message and the time the message was received by p. Once a processor knows its logical name, it can determine the relative names of those processors to which it must relay the message. The following scheme is used to generate logical names.

The logical names of the processor that originates the broadcast is: $<0, 1>$

At time t, $t \geq t_r$, a processor with logical name $<t_r, id>$ constructs the logical name

$$<(t_r + D) \bmod E, A_{DE}((t_r + D) \bmod E, t + D - 1) + id>$$

for the processor with relative name $T_{DE}(t + D - 1) + id$ and then sends the message there.

$A_{DE}(v,w)$ is the number of processors as of time w with v as first component of its logical name. Such processors must have received the message at time t, where $t \bmod E = v$, and so:

$$A_{DE}(v,w) = \sum_{i=0}^{floor\lfloor (w-v)/E \rfloor} R_{DE}(v + iE)$$

The consecutive numbering within an equivalence class is preserved by adding *id* to the number of elements in the class when forming new logical names.

The possibility of processor failures complicates matters only to the extent that unanticipated delays might be incurred. Messages may not be received "in time" and therefore processors might be assigned to the wrong equivalence class (first component of logical name). This can be circumvented if, instead of using wall-clock time in the logical name computations, logical clocks are used [L78]. Processor failures would not be considered "events", so they would not affect the "time" (according to the logical clock) that messages are received.

4.1. Chains and Bushes

If the message-delivery delay D and processor execution speed E satisfy $D > (N-1)E$ then a bush broadcast strategy (Figure 2.1a) minimizes the length of time necessary to complete a broadcast. On the other hand, if $E > (N-1)D$ then the chain broadcast strategy (Figure 2.1b) is optimal. This corresponds to our intuition that in practice the bush strategy results in faster broadcasts –a processor is usually faster than the communications network, so $D > (N-1)E$ is a closer approximation to reality than $E > (N-1)D$.

Recall that in the optimized version of our reliable broadcast protocol, a processor failure can result in $B(b,m)$ being established by each processor that has directly received a message from a failed processor. If there are f of these processors, then $f-1$ of these attempts are unnecessary. It would seem, then, that to minimize the duplication of work resulting from a processor failure, the number of direct successors of each node in the broadcast strategy tree should be small. The chain broadcast strategy has just this property. But, surprisingly, if each processor has the same probability of failure, then the amount of duplication of work that could result from a processor failure is about the same in both the chain and bush broadcast strategies. This is because in a bush, the failure of only one processor –the root– could cause duplication of effort, while in the chain, failure of any of $N-2$ processors (the internal nodes of the chain) could result

in this undesirable duplication of effort. With knowledge of the probabilities of failure for each processor, it is possible to construct a tree that minimizes the amount of duplication of work resulting from processor failures.

4.2. Slow Networks/Fast Processors

In most networks, delivery of a message is a relatively slow operation when compared to execution of an instruction on a processor. Consider the above scheme when $D = Er$, for some integer r. The first component of all logical names will be 0 because

$$(t_r + D) \bmod E = (t_r + Er) \bmod E = t_r \bmod E.$$

Secondly, computation of the second component of a logical name will be simplified, because

$$A_{DE}(0,n) = T_{DE}(n) .$$

Therefore, a processor with logical name $<0,id>$ at time t constructs the logical name

$$<0,\ A_{DE}(0,t+D-1)+\ id> \ = \ <0,\ T_{DE}(t+D-1)+\ id>$$

for the processor with relative name $T_{DE}(t+D-1)+\ id$ and sends the message there. Note that the logical and relative names are the same. Note also that when $r = 2$, computation of $R_{DE}(t)$ is equivalent to computing Fibonacci numbers.

4.3. Synchronous Message Passing

Synchronous message passing primitives have been proposed for use in implementing distributed systems [H78]. When such primitives are used, a processor executing a send instruction is delayed until the message is received. This can be modeled by setting $D = E$. Then, a processor cannot send a message until the last one it sent has been received. For simplicity, assume $D = E = 1$. From equation (4.1) we get

$$R_{DE}(t) = \begin{cases} 0 & \text{if } t < 0, \\ 1 & \text{if } t = 0, \\ 2^{t-1} & \text{if } t > 0. \end{cases}$$

Therefore, $T_{DE}(t) = 2^t$. This clearly simplifies broadcast strategy computations. Computing

powers of 2 can be done easily by shifting.

5. Discussion

Much of the work concerning the development of reliable broadcast protocols has been done in connection with designing fault-tolerant distributed systems and computer networks. There, it is often necessary to communicate state information to all sites and to be certain that the states of these sites converge; i.e. either all functioning sites receive the new state information or none do. SAFETALK is an example of such a protocol [MPM80]. It employs a bush-like broadcast strategy (Figure 2.1a), but unlike our protocol, a broadcast may not complete if the originating site fails. This is sufficient for the applications for which the protocol was intended.

Ellis develops a chain-like (Figure 2.1b) reliable broadcast protocol and proves it correct using L-Systems [E77]. The protocol is intended for use in updating redundantly stored entities in a distributed database system. Unfortunately, the linear time delay of the protocol makes its use impractical in many situations. In [AD76] another chain-like protocol is proposed.

[PL79] describes "best-effort-to-deliver" and "guarantee-to-deliver" protocols. These protocols are based on broadcast strategies that do not allow minimum-time broadcasts; the strategies do not fully exploit parallelism inherent in a network.

Byzantine Generals Protocols [LSP80] and their variants (interactive consistency [PSL79], Crusaders Agreement [D82] and Weak Byzantine Generals [L81]) support broadcasts in networks in which no assumptions are made about processor failures or the communications network. The cost of broadcasting in such a harsh environment is very high: a total of $t+1$ rounds of message exchange are required to withstand up to t failures and the number of bits exchanged is bounded by a polynomial [DS81].

Broadcast protocols that are not robust with respect to processor failures are described in [DM78] and [W80]. They can be viewed as broadcast strategies and used in conjunction with the protocol developed in Section 3 to implement reliable broadcasts.

Acknowledgments

John Hopcroft provided invaluable assistance with some of the recurrence equations in Section 4. Discussions with Gary Levin have also been helpful. This paper was inspired by and results from a challenge made by Gerard LeLann.

References

[AD76] Alsberg, P.A., and J.D. Day. A principle for resilient sharing of distributed resources, *Proceedings of Second International Conference on Software Engineering*, San Fransisco, 1976, 562-570.

[DM78] Dalal, Y.K., and R. M. Metcalfe. Reverse path forwarding of broadcast packets, *CACM 21*, 12 (Dec. 1978), 1040-1048.

[D82] Dolev, D. The byzantine generals strike again, *Journal of Algorithms 3*, 1, (1982).

[DS81] Dolev, D. and H.R. Strong. Polynomial algorithms for multiple processor agreement, IBM Research Report RJ3342, 1981.

[E77] Ellis, C.A. Consistency and correctness of duplicate database systems, *Proceedings of the Sixth Symposium on Operating Systems Principles*, Purdue University, Nov. 1977, 57-84.

[F73] Farber, D.J., *et al.* The distributed computing system, *Proceedings of CompCon 73*, Feb. 1973.

[H78] Hoare, C.A.R. Communicating sequential processes, *CACM 21*, 8 (August 1978), 666-677.

[L78] Lamport, L. Time, clocks and the ordering of events in a distributed system, *CACM 21*, 7 (July 1978), 558-565.

[L81] Lamport, L. The weak byzantine generals problem, Opus 58, Computer Science Laboratory, SRI International, Sept. 1981.

[LSP80] Lamport, L., R. Shostak and M. Pease. The byzantine generals problem, *TOPLAS 4*, 3 (July 1982), pp. 382-401.

[LL79] LeLann, G. An analysis of different approaches to distributed computing, *Proceedings of the First International Conference on Distributed Computing Systems*, Alabama, Oct. 1979, 222-232.

[MPM80] Menasce, D.A., G.J. Popek and R.R. Muntz. A locking protocol for resource coordination in distributed databases, *TODS 5*, 2, 103-138.

[M76] Metcalf, R.M. ETHERNET: Distributed packet switching for local computer networks, *CACM 19*, 7 (July 1976), 395-403.

[N79] Needham, R.M. Systems aspects of the Cambridge Ring, *Proceedings of the Seventh Symposium on Operating Systems Principles*, Pacific Grove, California, Dec. 1979, 82-85.

[PL79] Pardo, R., and M.T. Liu. Multi-destination protocols for distributed systems, *Proceedings Computer Networking Symposium*, Gaithersburg, Maryland, Dec. 1979.

[PSL79] Pease, M., R. Shostak and L. Lamport. Reaching agreement in the presence of faults, *JACM 27*, 2 (April 1980).

[S80] Schneider, F.B. Broadcasts: A paradigm for distributed programs, Department of Computer Science, Cornell University, Technical Report TR 80-440, Oct. 1980.

[S82] Schneider, F.B. Synchronization in distributed programs, *TOPLAS 4*, 2 (April 1982), 125-148.

[W80] Wall, D.W. Mechanisms for broadcasts and selective broadcast, PhD Thesis, Computer Systems Laboratory, Stanford University, June 1980.