Fault-Tolerant Broadcasts*

Fred B. Schneider
David Gries
Richard D. Schlichting+

Department of Computer Science
Cornell University
Ithaca, New York 14853

August 31, 1980
Revision: September 3, 1982
Revision: August 2, 1983

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 assumed.

---

*This work is supported by NSF Grants MCS 78-22360 and MCS 81-03605.

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

1. Introduction

A *fault-tolerant 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. Fault-tolerant broadcast protocols have application in a wide variety of distributed programming problems [S80] [S82].

Broadcast networks—contention networks such as Ethernet [M76] and ring networks like DCS [F73]—would appear to implement fault-tolerant 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,

- the buffer memory might be full when a message is received by the interface unit,
- the interface unit might not be monitoring the network at the time the message is delivered, or
- in a contention network, an undetected collision that affects only certain network interface units could cause them to miss a message.

Thus, while current broadcast networks allow messages to be broadcast, they do not directly support fault-tolerant broadcasts.

In point-to-point networks, in which a message sent can be received by only one processor, there are other impediments to implementing fault-tolerant 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 presented in 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 fault-tolerant broadcast protocol that will work with any broadcast strategy is presented and proved correct. Section 4 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 send messages to every other functioning processor.
>
> Messages between every pair of functioning processors are delivered uncorrupted and in the order sent.

Clearly, to withstand up to k failures, there must be $k+1$ independent paths between any two processors. These paths may be direct or may involve relaying messages through other processors. Thus, we are assuming the existence of an underlying routing protocol, such as the one in [MS79]. To ensure that messages are delivered uncorrupted and in the order sent, we assume the existence of protocols to append sequence numbers and checksums to messages, and, if necessary, to retransmit garbled or out-of-order messages. Although achieving the Reliable Communications Property is likely to be expensive, it is impossible to distribute a message to a processor if there is no way to communicate with it.

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

$m.sender$	the name of the processor that sent m.
$m.info$	the information being broadcast.
$m.seqno$	a sequence number assigned to the message by the processor b that initiates the broadcast. The first message broadcast has sequence number 1.

Let m be a message. Execution of

$$p!!msg(m)$$

by processor q sends a message m' to p with $m'.sender = q$, $m'.info = m.info$ and $m'.seqno = m.seqno$. Execution 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*. Thus, a message sent using $p!!msg(...)$ can be received only using $??msg(...)$, and a message sent using $p!!ack(...)$ can be received only using $??ack(...)$.

This notation is inspired by the input and output commands of CSP [H78]. As in CSP, we allow receive commands ($??$) to appear in the guards of guarded commands. Such a guard is never false; it is true only if execution of the receive would not cause a delay. In our notation, two shrieks (!!) and queries ($??$) are used, instead of one, to indicate that messages are buffered and therefore a sender is never delayed. 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 that has *failed* stops executing.

Thus, we do not consider the case where a malfunctioning processor continues executing, although not in a manner defined by its program. The validity of our processor failure assumption is arguable because processor failures can result in arbitrary behavior. However, most processor failures will cause the offending processor to cease sending messages, so to the other processors in a computer network such a malfunctioning processor will appear to have stopped. Processor failures that cause a malfunctioning processor to generate arbitrary messages are also easily handled—the messages are ignored and thus the failed processor has effectively stopped. Discussions of the implementation of true "fail-stop processors" and their cost appear in [S83] and [SS83].

In our proofs, we use the predicate $failed(p)$:

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

Each processor is assumed to have a local variable $FAILED$, which is a set of names of the processors that it has recognized as having failed. Failure of a processor p automatically causes $FAILED := FAILED \cup \{p\}$ to be executed in each functioning processor after some finite, but undetermined, amount of time. This can take place at different times for the different processors. Local variable $FAILED$ satisfies the following:

$p \in FAILED \Rightarrow failed(p)$
$failed(p) \Rightarrow$ eventually p will be added to $FAILED$

$FAILED$ models an idealized failure-detection mechanism. The proof of our broadcast protocol assumes that the implementation of $FAILED$ is consistent with the properties and meaning of *failed* stated above. Thus, an implementation that only approximates $FAILED$ will yield a broadcast protocol that runs correctly only as long as the implementation behaves as stipulated above. One way to approximate $FAILED$ is to use time-outs with synchronized clocks. Then, provided the time-out period is sufficiently long and network delays are not significant, $FAILED$ will behave as required.

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.

[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.

Generally speaking, the successors of a node in the broadcast strategy will be neighbors of the node in the network, but this is not necessary. The broadcast strategy defines how a message is to be broadcast; it is the duty of a lower-level protocol to ensure delivery of messages to their destinations, as postulated in the Reliable Communications Property.

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

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

$SUCC^+$ and $SUCC^*$ denote the conventional transitive closure and reflexive transitive closure of relation $SUCC$. We also use the name of a relation to denote a set: $SUCC(P)$ is the set of successors of the elements of 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. Processor failure may require deviation from the strategy. Clearly, the broadcast strategy to employ in a given situation depends on what is to be optimized. However, use of broadcast strategies that can be represented by a subgraph of the processor interconnection graph seems reasonable, since it minimizes message relaying.

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.

3. Fault-tolerant Broadcasts with Unreliable Processors

We now present a fault-tolerant 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 is slightly different because broadcasts are initiated there.

Throughout, $\overline{m}$ denotes the value of the message currently being broadcast by b. The broadcast of $\overline{m}$ is completed when the following holds.

> *Fault-Tolerant Broadcast*: If any functioning processor has a copy of $\overline{m}$ then every functioning processor has a copy of $\overline{m}$.

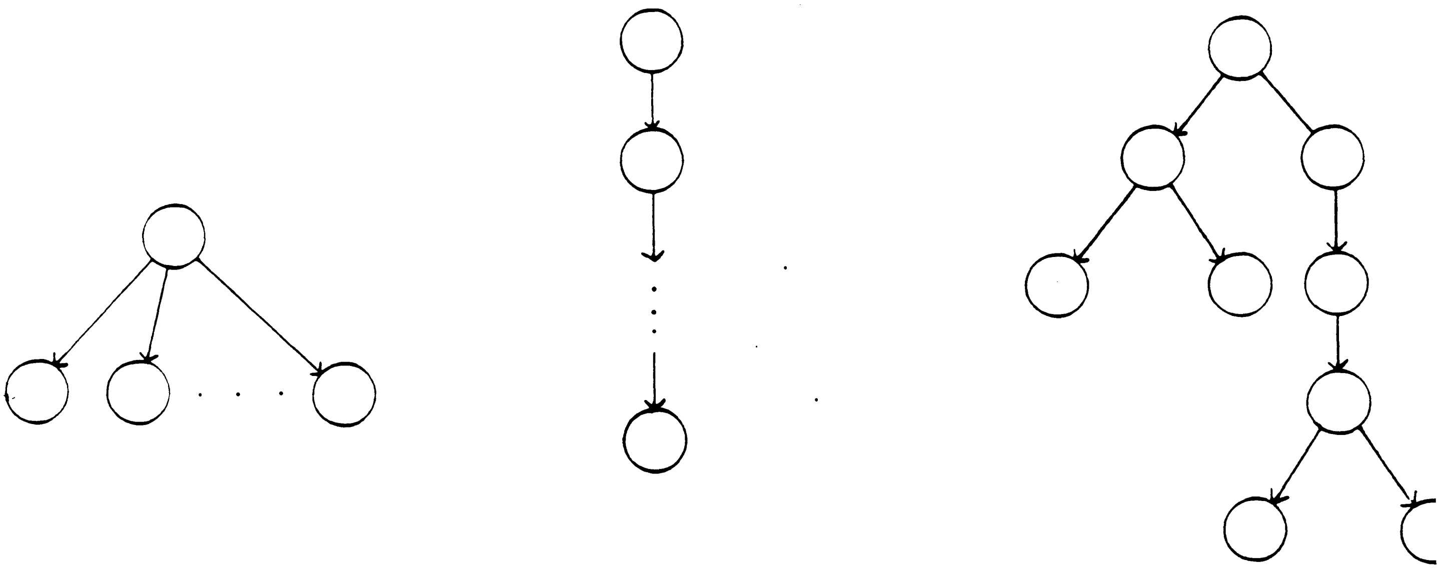

Figure 2.1 — Some Broadcast Strategies

This means that one way to complete a broadcast is for every processor that has received the message to fail.

Let m_p be a local variable in process p that contains the last message delivered to it and let

$$eq(m1, m2) \equiv m1.seqno = m2.seqno \wedge m1.info = m2.info$$

A fault-tolerant broadcast protocol establishes the truth of $FTB(\overline{m})$, where

$$FTB(\overline{m}) \equiv (\exists p: p \in SUCC^*(\{b\}): \neg failed(p) \wedge eq(\overline{m}, m_p)) \Rightarrow B(b, \overline{m})$$

and

$$B(j,m) \equiv (\forall p: p \in SUCC^*(\{j\}): failed(p) \vee eq(m, m_p))$$

Restarting a failed processor can falsify $FTB(\overline{m})$. To avoid this problem, we postulate temporarily that once a processor has failed it remains failed. We return to this problem in Section 3.3, where we devise a processor-restart protocol.

3.1. Assuming b does not Fail

We begin by assuming that b does not fail, but other processors may. Thus, at least one functioning processor—b—has received $\overline{m}$, so in order to make $FTB(\overline{m})$ true, $B(b,\overline{m})$ must be established. To do this, when a processor i receives $\overline{m}$ and stores it in its local variable m_i, its duty is to establish $B(i,m_i)$—to make sure that all functioning members of its subtree receive m—and then to acknowledge it. Upon receipt of $\overline{m}$, i relays it to every processor p in $SUCC(\{i\})$. Each of these establishes $B(p,\overline{m})$ 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 $m_i.sender$.

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

3.2. Assuming b may Fail

We now investigate the complications that arise when b may fail. Upon receiving a message $\overline{m}$, processor i operates as described in Section 3.1 and, provided b does not fail, $B(b,\overline{m})$ will be established by b. If b fails and no other functioning processor has received $\overline{m}$, $FTB(\overline{m})$ is true (the antecedent is false), so the broadcast is completed. Otherwise, some functioning processor that received $\overline{m}$ must establish $B(b,\overline{m})$. Since no harm is done if $B(b,\overline{m})$ is established by more than one processor, we allow more than one to establish it. However, this means that i may receive more than one copy of $\overline{m}$, each corresponding to a request for i to establish $B(i,\overline{m})$ and respond with an acknowledgement. In order to send these acknowledgements, processor i maintains a set of processors to which acknowledgements must be sent. Thus, three set-valued variables are used by each processor:[2]

sendto	$\equiv$	the set of processors to which m_i must be sent;
ackfrom	$\equiv$	the set of processors from which acknowledgements for m_i are awaited;
ackto	$\equiv$	the set of processors that sent m_i to i for which acknowledgements must be returned.

After receiving $\overline{m}$, process i monitors b until it recognizes that b has failed or that $FTB(\overline{m})$ is true. Therefore, some means must be found to notify processes that $FTB(\overline{m})$ is true. Unfortunately, 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.[3]

Now, receipt by process i of a message m' with $m'.seqno > m_i.seqno$ means that the broadcast of m_i is completed. Thus, b can notify processes of completion of a broadcast simply by initiating the next one. Unfortunately, this means that the completion of the "last" broadcast carries with it some uncertainty.

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

With this initial discussion, we can now describe the invariant of the loop of the protocol for process i given in Fig. 3.1.[4] This invariant will be used to argue about the partial correctness of the protocol, that progress is made during execution of the protocol, and that no deadlock occurs. As each conjunct of the invariant is given, the reader should verify that it is indeed invariant, using the discussion following it and the previously established conjuncts of the invariant. 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.

---

[2]Again, local variable m_i of processor i contains the message $\overline{m}$ being broadcast.

[3]This is not really a restriction. A root b can have several identities and can concurrently run a separate instance of the protocol for each identity. This allows b to concurrently perform multiple broadcasts.

[4]There, operation *choose*(*sendto*, *dest*) stores an arbitrary element of set *sendto* into *dest*. 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

$P1$: $m_b.seqno \geq m_i.seqno$

Initially, each processor sets $m.seqno$ to 0, so that $P1$ is true. (By convention, we assume every process has received the empty initial message with sequence number 0). Since process b changes $m_b.seqno$ only to a higher number, execution of b cannot falsify $P1$. Process i changes m_i only when executing $m := new$, where new is a message received using $??msg(new)$. Since this message was sent using the statement $dest!!msg(m_p)$ by some other process p and p also maintains $P1$—i.e. $m_b.seqno \geq m_p.seqno$—this assignment cannot falsify $P1$.

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

$P2$ is initially true because $sendto$ and $ackfrom$ are empty. Sets $sendto$ and $ackfrom$ 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 $B(i,m)$, detecting the failure of b causes it to set r to b and to attempt to establish $B(b,m)$.

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

$P3$ follows from the nature of a tree and the operations performed on the two sets.

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

$P4$ 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.

$P5$ indicates that any successor p of node i is in one of four categories: i should send m to p, or i has sent m to p but has not yet received an acknowledgement, or i has sent m to p and received an acknowledgement, or p has failed.

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

$P5$ is initially true since, by convention, every processor has received the empty message with sequence number 0. Verifying that each guarded command leaves $P5$ true is fairly easy, except

being used.

for the command with guard $??ack(a)$. Here, the sender of the acknowledgement is deleted from *ackfrom*. In order to maintain *P5*, we require that $B(p,m)$ be a precondition for p to send an acknowledgement for m. In Figure 3.1, $B(i,m)$ is explicitly given as the precondition of each acknowledgement sent by i.

P6: $r = i \lor (B(i,m) \land failed(b))$

P6 is true initially because $r = i$. The first conjunct may be falsified by changing r to b, but this is done only when $ackfrom = \Phi \land sendto = \Phi$, which together with *P5* and the fact that i has not failed implies $B(i,m)$. The second conjunct can be 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.

Whenever $r \neq i$, processor i must attempt to establish $B(b,m)$. To do so, i ensures that *every* processor p either (1) is in $SUCC^*(sendto)$, (2) is in $SUCC^*(ackfrom)$, (3) has established $B(p,m)$, or (4) has failed. This information is given in *P7*:

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

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.

In *P8* we describe the set *ackfrom* a bit more precisely:

$$P8\colon\ q \in ackfrom_p \ \equiv\ \begin{array}{l} m \text{ is in transit from } p \text{ to } q \ \lor \\ p \in ackto_q \ \lor \\ \text{an acknowledgement for } m \text{ is in transit from } q \text{ to } p \end{array}$$

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, $FTB(\overline{m})$ holds.

First, note that the loop of the protocol never terminates: because some of the guards delay until a message is received, they are never false. Secondly, note that each processor p sends $\overline{m}$ to

```
m:= (sender: b, info:nil, 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 → 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

 [] b∈FAILED ∧ r≠b ∧ ackfrom=Φ ∧ sendto=Φ → {B(i,m)}
                                            r, sendto:= b, SUCC({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 $b \in FAILED$ 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 3.1 – Reliable Broadcast Protocol

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.

Remark. The need for such history variables in order to complete the proof might be disturbing to some. If the algorithm terminates, then it does so because some function of the actual state keeps decreasing. However, the state of the system includes the contents of network buffers and the values of program counters in the various processors. Rather than reason about these—which could be quite messy—we have chosen to introduce history variables. □

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

$$
\begin{aligned}
<&3N(N-1)(\overline{m}.seqno-1) - (\Sigma m\colon m.seqno < \overline{m}.seqno\colon Rmsg(m) + Sack(m) + Rack(m)),\\
&(\mathbf{N}p\colon \neg failed(p)),\\
&(\mathbf{N}p\colon \neg failed(p)\colon \neg eq(\overline{m}, m_p)),\\
&(\mathbf{N}p\colon \neg failed(p)\colon \neg eq(\overline{m}, m_p) \vee (eq(\overline{m}, m_p) \wedge r_p = p)),\\
&(\Sigma p\colon \neg failed(p) \wedge eq(\overline{m}, m_p)\colon |SUCC^*(sendto_p) \cup SUCC^+(ackfrom_p)|),\\
&(\Sigma p\colon \neg failed(p) \wedge eq(\overline{m}, m_p)\colon |SUCC^*(ackfrom_p)|),\\
&(\mathbf{N}m\colon m.seqno = \overline{m}.seqno\colon m \text{ a message in transit}),\\
&(\Sigma p\colon \neg failed(p)\colon |ackto_p|) >
\end{aligned}
$$

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, 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 $B(b,\bar{m})$ is true. Assume that b performs no broadcast after $\bar{m}$. Then, since the 8-tuple is bounded below and decreases with each iteration, after a finite amount of time all messages will have been delivered and all processor failures will have been recognized. Thus, no further iteration can occur and each processor is delayed. By the lemma below, in this state $sendto = \Phi$ and $ackfrom = \Phi$ for each processor. If no functioning processor has $\bar{m}$ then $FTB(\bar{m})$ is true. We now show that if at least one functioning processor has $\bar{m}$, they all do. Suppose b has not failed. b has $\bar{m}$, so by *P5* all processors have received $\bar{m}$ or have failed. Suppose b has failed and another processor i has $\bar{m}$. By the lemma below, $r_i = b \neq i$. By *P7* and the lemma, all processors have failed or have received $\bar{m}$.

Lemma. Assume that all messages in transit have reached their destination, that no further failure occurs, that all failures have been recognized by all processors, and that all processors are delayed. Then $sendto = \Phi$ and $ackfrom = \Phi$ for all functioning processors. Further, for each processor i, $\neg failed(b) \vee r_i = b$ holds.

Proof. Inspection of the guards of the loop of the protocol in the state mentioned in the lemma 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

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

Repeating this argument, some descendent $q1$ of q satisfies $\Phi \neq ackfrom_{q1} \subseteq SUCC^{+}(\{q\})$, some descendant $q2$ of $q1$ satisfies the same property, and so forth, indefinitely. This leads to a contradiction because the broadcast strategy is a finite tree. Hence, all sets $ackfrom$ are empty. Appealing to (4) above yields $\neg failed(b) \vee r_i = b$ for all processors i. Q.E.D.

An Optimization

As it now stands, each processor monitors b. However, if b fails before $B(b,\bar{m})$ is established, then some functioning processor must have received the message from a processor that has failed. Thus,

$$b \in FAILED \Rightarrow FTB(\overline{m}) \vee (\exists p: \neg failed(p) \wedge eq(\overline{m}, m_p)): failed(m_p.sender))$$

This allows $b \in FAILED$ to be replaced by $m.sender \in FAILED$ in the above protocol. Thus, each processor need monitor only a processor with which it is communicating (e.g. its predecessor). However, now more than one processor may attempt to establish $B(b, \overline{m})$, 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

$$(\forall\ m\!: m \ broadcast\ by\ b\!: (\exists p\!: p \in SUCC^{*}(\{b\})\!: \neg failed(p) \wedge rec(p,m)) \Rightarrow rec(i,m))$$

is established, where:

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

We suggest a two-step restart protocol:

(1) Some functioning processor p relays to i a copy of every message p has received that i did/will not. Naturally, these messages must have been stored by processor p.

(2) Processor i initiates a broadcast of each message m it has received that was not forwarded to i during step (1). This is necessary because all the processors that originally received m might have faiied; if i is the first of these to be restarted, it must broadcast m.

4. Discussion

Chains and Bushes

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.

If $D > (N-1)E$ 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 fault-tolerant 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.

Related Work

Much of the work concerning the development of fault-tolerant 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 install 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. The transaction management facilities in Delta [LL81] employs a bush-like broadcast in conjunction with a two-phase commit protocol to implement fault-tolerant broadcasts even if the originating site fails in certain restricted ways.

Ellis develops a chain-like (Figure 2.1b) fault-tolerant 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.

In [SA83], Segall and Awerbuch describe a reliable broadcast protocol. Their work is based on a fault-tolerant protocol to compute spanning trees in a computer network [MS79]. A message is disseminated along the spanning tree in effect at the time its broadcast is initiated. New spanning trees are computed in response to events such as failure of a site or failure of a link. A protocol, which employs logical clocks in a manner similar to the "Restart Protocol" in [S82], is used to change the spanning tree in effect without affecting messages already in transit.

Byzantine Agreement 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, relative clock speeds, 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
Discussions with Gary Levin have been helpful. This paper was inspired by and results from a challenge made by Gerard LeLann. We are indebted to LeLann, Nissim Francez, Howard Sturgis, and Leslie Lamport for constructive criticisms of earlier drafts of this paper.

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.

[LL81] LeLann, G. A distributed system for real-time transaction processing. *IEEE Computer*, Feb. 1981, 43-48.

[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.

[MS79] Merlin, P.M. and A. Segall. A failsafe distributed routing protocol. *IEEE Trans. on Communications COM-27*, 9 (Sept. 1979), 1280-1287.

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

[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.

[S83] Schneider, F.B. Fail-stop processors. *Digest of Papers Spring Compcon '83*, IEEE Computer Society, March 1983, San Fransisco, CA, 66-70.

[SS83] Schlichting, R.D. and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. To appear *TOCS 1*, 3 (August 1983).

[SA83] Segall, A. and B. Awerbuch. A reliable broadcast protocol. *IEEE Trans. on Communications COM-31*, 7 (July 1983), 896-901.

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