DETECTING DISTRIBUTED TERMINATION
WHEN PROCESSES CAN FAIL*

C. W. Lerman
F. B. Schneider

TR 80-449
December 1980

Department of Computer Science
Cornell University
Ithaca, New York  14853

*This research was supported in part by the National Science Foundation under grant MCS-76-22360.
Abstract

A collection of protocols to facilitate detection of the termination of a computation on a distributed system are developed. Communication is assumed to be accomplished by use of asynchronous broadcasting. It is argued that this is a reasonable assumption for a distributed system in light of advances in local networking.

The protocols presented are all robust with respect to processor failures. They differ in their requirements — some make heavy use of the communications network at the end of a computation, while others spread the communications cost out through the computation.

Problems of restarting failed processors are also addressed.
Chapter 1
Motivating the Problem

1.1 Introduction

Computing systems that contain more than one processor can be programmed to exhibit some degree of fault tolerance. Two classes of configurations are possible: multiprocessors and distributed systems. A multiprocessor is characterized by the existence of shared memory, which can be used for communications and synchronization of processes, while in a distributed system this communication and synchronization is accomplished by message passing. It is probably simpler to write a concurrent program for a multiprocessor because of the availability of shared memory. For one thing, it allows processes to obtain consistent views of the system state. This considerably simplifies solutions to synchronization problems. However, the existence of shared memory can be an impediment to constructing fault tolerant systems. In addition, multiport shared memories are expensive, can become performance bottlenecks and can prevent the geographic separation of processors. Thus, use of distributed systems seems appropriate if fault tolerant operation is to be achieved. Even then, careful attention must be paid to software structure so that the resulting system -- hardware and software -- is fault tolerant. One such structuring technique motivates the problem addressed in this thesis.

1.2 Assumptions About the Hardware

A distributed system consists of some processors and a communications network. Below, some assumptions we make about this hardware are described.
In order to cope with processor failures, they must be detected. Thus, we assume:

AI: All processor failures are detected.

Of course, with a finite amount of hardware, it is possible that some failures will go undetected. But, it is always possible to construct a system in which AI is approximated (to any degree desired).

When a processor fails, any processes running on that processor are considered to have failed, too. We restrict attention here to passive failure modes. That is, when failure is detected, the offending processor is shut down. Consequently:

A2: After a process has failed, it no longer sends messages.

A process failure can then be detected by other processes in the system. This is accomplished by sending a message to the process believed to have failed, and waiting for a sufficient time interval to elapse so the sender can be certain that no response will be forthcoming. By A2, lack of a response indicates failure.

Some assumptions about the network must be made, as well. For the most part, construction of a network satisfying them is well within the state of art. First,

A3: Fully reliable communication is provided. Consequently:

(a) No messages are received that have not been sent.

(b) All messages that are sent will be received, provided the destination processor is not failed.

One way to accomplish this is for a sender to retransmit a message until either it receives an acknowledgment from the destination or it receives a notification that the destination has failed. Secondly, we assume
A4: The relative order of messages is preserved between pairs of processes.

That is, process P receives messages from process Q in the same order they were sent. This can be implemented by including sequence numbers on messages. Thirdly, we assume

A5: A facility exists to broadcast messages. Such broadcasts are reliable -- they are received either by every running process or by no running process [SS80].

Certain network topologies are better suited than others for providing cheap reliable broadcasts. Most local network topologies -- rings and contention networks -- support broadcasting. And, use of a local network when constructing a distributed system seems most appropriate. The salient characteristics of these local network are described in the following.

In a ring network (for example, see [FA73]), processors are connected to interface units. These, in turn, are connected to form a unidirectional ring. Messages are passed from interface to interface, so that every processor can receive all messages. Messages contain status bits that are used to indicate the processors that received the message. Since eventually every message sent is returned to the sender, a decision to retransmit the message, because it was missed by one or more processors, can be made by checking these status bits.

In a contention network (see, for example, [HS76, AB70]) processors are connected through interface units to a shared medium -- a coaxial cable, a radio channel, or a fiber-optic link. The medium can carry at most one message at a time. A collision occurs when more than one message is broadcast on the medium at a time. Therefore, a protocol is employed to
govern access to the shared medium, so that collisions are minimized. There are two general approaches to handling collisions. In both, an interface unit defers to a transmission in progress. In a slotted protocol, a unique time slot is assigned to every process, and a process can transmit only during this interval [Ab70]. Thus, as long as synchronization is maintained, no collisions will occur. In the second approach, collisions are allowed to occur. When a collision is detected, both senders stop and then delay (different) random time intervals. Retransmission is then attempted. The randomized delay helps to minimize the chance of subsequent collisions of the same messages. This is the scheme employed in Ethernet [MB76].

Ring and contention networks do not require complicated routing algorithms. They also support relatively cheap implementations of broadcasting. In fact, performing a broadcast is often no more expensive than sending a single message. This is not necessarily the case in "point-to-point" computer communications networks, such as the ARPA-net [RW70]. Even there, however, fast reliable broadcast can be provided [SS80], as long as processor failures leave the network connected.

1.3 The Problem: Detecting Distributed Termination.

A concurrent computation can be specified using cobegin statements. The statement

S: cobegin P_1 // P_2 // ... P_r coend

denotes the concurrent execution of processes P_1, P_2, ..., P_r. S terminates when P_1, P_2, ..., P_r have terminated.
One implementation of this statement on a distributed system involves executing the components of $S$ on different processors. Then, some means must be found to detect when each $P_i$ has terminated. This is complicated by the fact that each process $P_i$ may execute send and receive statements, to exchange information -- called basic messages -- with other components of the cobegin statement.

Assume we are given a protocol to detect when each process $P_i$ can progress no further -- i.e., $P_i$ might have terminated, might be waiting to receive a message that will never arrive, or might have been assigned to a processor that has failed. Then, a fault tolerant program to solve a specific problem can constructed as follows. First, the problem is decomposed in a top-down fashion into (smaller) subproblems, until each subproblem is defined as a fairly simple relation between two predicates. Then, each of these state transformations is implemented by using a cobegin statement. Consequently, the resulting program consists of a sequence of cobegins. Thus, let $S$ above be a cobegin statement that is supposed to transform the states satisfying assertion $(Q)$ into states satisfying assertion $(R)$. $S$ will implement this state transformation in a fault tolerant manner on a distributed system with $r$ processors provided:

1. Each $P_i$ is assigned to processor $i$.
2. $(\forall i : 1 \leq i \leq r : [Q] P_i [R_i])$
3. $(\exists i : 1 \leq i \leq r : \neg \text{failed(processor } i) )$
4. $\bigwedge_{i=1}^{r} (\text{failed(processor } i) \lor R_i) \Rightarrow R$

Structuring a program so that condition (4) is satisfied is often non-trivial. The key idea is that after a processor has failed, somehow a process executing on another processor must establish the necessary result
assertion for the failed component (process) -- as well as its own result assertion. In [Sc79] an approach involving broadcasts and state replication for structuring programs satisfying (3) is explored; in [La78b], a second approach using finite state machines is described.

A number of protocols for detecting distributed termination have appeared in the literature [DS79, Fi80, Lo80]. None of these algorithms handle processor failures, nor were they intended to. Detection of distributed termination has become a classical problem in the distributed programming literature because it is difficult even when failures cannot occur. So, most of the work to date has concentrated on the synchronization aspects of the problem, while ignoring considerations of fault tolerance. This work is therefore concerned with the design of fault tolerant protocols for detecting distributed termination.

Thus, while [Fi80] and [Lo80] use single-destination synchronous message passing, our protocols are in terms of multiple destination (broadcast) asynchronous message passing. Programs written in terms of synchronous message passing are often easier to understand and reason about than programs using asynchronous message passing. Unfortunately, the tight coupling of processes resulting from the use of synchronous message passing makes the construction of fault tolerant systems more difficult. This is because fault tolerance is best achieved when the performance of processes and the communication network have minimal effects on each other. Consequently, asynchronous message passing is preferred. However, this makes process synchronization a difficult task. Processes have direct access only to local information, and any state information a process receives in messages reflects a past state of of the sender, due to delivery delays. Therefore, no single process can have complete knowledge
of the entire state of the system. Use of broadcast mitigates this difficulty.

Lastly, it is interesting to note that we have been able to derive the protocols in [DS79], [Le80], [Fr80] from ours by assuming processes do not fail, and performing optimizations on the communications used.

1. A Outline of the Thesis

In chapter 2, a simple protocol is developed that allows detection of distributed termination. There, whenever a process suspects distributed termination, it "flushes" all messages from the network to confirm this suspicion. Since this "flush" can be expensive, a second protocol with a more efficient "flush" is then given. A third protocol that uses a different approach is presented in chapter 3. There, all communications are known system-wide. By restricting the distributed termination problem by making additional assumptions, in chapter 4 the method described in [DS79] is obtained from our protocols. In chapter 5, the difficulties of restarting processes that have failed are addressed. Finally, chapter 6 summarizes the results.
Chapter 2
Two Protocols Using "Flushes"

2.1 A Simple Protocol and its Derivation

For the purpose of detecting distributed termination, a process \( P_i \) can be in one of three states. \( P_i \) is failed if it was being executed by a processor that failed, otherwise \( P_i \) is active or quiet as follows:

I1: A quiet process becomes active only upon receipt of a basic message.

I2: No quiet process sends basic messages.

I3: An active process can become quiet at any time.

Detecting distributed termination is equivalent to detecting that no process is active and no process will subsequently become active as a result of messages as yet undelivered. As will be seen, use of broadcasting considerably simplifies detecting this condition.

Since every process has direct access only to its local storage, message passing is used by a process to communicate its state to other processes. Such messages are called state messages. Messages - like variables - are typed. Every process \( P_i \) stores the messages received from every other process \( P_j \) in the sequence they were received (hence, by A4 this is the sequence they were sent) in a queue \( mq_i(j) \). Furthermore, \( P_i \) stores any messages it sends in the message queue \( mq_i(i) \). Initially, all message queues are empty. We define:

\[
\text{proj}(mq,s) = \text{For } mq \text{ a message queue, and } s \text{ a set of message types,}
\]

\[
\text{proj}(mq,s) \text{ is the message queue obtained from } mq \text{ by deleting those messages with types not contained in } s.
\]
last(m, mq) \equiv \text{true if } m \text{ is the last message on } mq.

length(mq) \equiv \text{For } mq \text{ a message queue, } length(mq) \text{ gives the number of messages in that queue.}

In order to detect distributed termination, whenever a process changes states, it will broadcast a state message signifying that fact.

Three message types are required for this:

- a(i) to signify that process \( P_i \) has become active.
- q(i) to signify that process \( P_i \) has become quiet.
- f(i) to indicate that the processor on which process \( P_i \) was running, has failed.

Then, we require

14: Whenever process \( P_i \) becomes active it broadcasts a(i) before sending any other messages.

15: Whenever process \( P_i \) becomes quiet it broadcasts q(i).

16: If process \( P_i \) fails, then f(i) is broadcast so as to arrive after all messages from that process.\(^1\)

Distributed termination can be detected at process \( P_i \) if the following predicate could be evaluated.

\[ DT_i \equiv T_i \land \text{invariant}(T_i) \]

where

\[ T_i \equiv (\forall j \colon 1 \leq j \leq r : \text{last}(f(j), mq_j(j)) \lor \text{last}(q(j), \text{proj}(mq_j(j), \{a(j), q(j)\}))) \]

\(^1\) Based on to A2, processor failures can be detected by a "tie-cut" mechanism. Presumably, the last message that was broadcast by a malfunctioning processor is received before the failure is announced, so that a proper sequence number -- to preserve A4 -- can be assigned to the f(i) message.
\textit{Invariant}(T_i) \equiv "T_i \text{ will not subsequently change value}".

As will be seen below, the projection of the \(q(j)\) messages is necessary, because a process might receive messages of other types after receipt of a \(q(j)\) message. Determining the value of \(T_i\) is fairly simple, since it involves only local storage at process \(P_i\). Ascertaining that \(T_i\) will remain true, once it is true (i.e., \(T_i \land \text{\textit{Invariant}}(T_i)\)) is somewhat more difficult -- it is an assertion about the future. Note, however, that the following is sufficient to ensure the invariance of \(T_i\):

\(m1\): There are presently no "\(a(j)\)" messages (from any \(P_j\)) in the communications network that are destined to process \(P_i\).

\(m2\): There are presently no basic communications destined to any process.

Condition \(m1\) arises from the use of asynchronous message passing primitives; condition \(m2\) follows from 11, 12 (and 14). \(P_i\) knows nothing about basic communication between other processes. Therefore, \(T_i\) could evaluate to true at \(P_i\) even though there is still an undelivered basic message sent by \(P_j\) prior to the last \(q(j)\) message that \(P_i\) received.

Assumption A4 -- preservation of the ordering of messages between processes -- can be exploited to "flush" messages from the communications network so that \(m1\) and \(m2\) are made true. This involves sending a message in both directions between every pair of processes. The protocol to accomplish this is:

17a: When \(T_i\) becomes true, process \(P_i\) broadcasts an \(es(a_{i,i},i,i)\) message.

where \(a_{i,i} = \sum_{j=1}^r \text{length}(\text{proj}(mq_i(j),\{a(j)\}))\).

17b: Upon receipt of an \(es(u,j,j)\) message, process \(P_i\) broadcasts an \(es(u,j,i)\) message.
A process $P_k$ can determine that there are no undelivered communications that were sent prior to the broadcast of the $es(u,i,i)$ message destined for it by using the following predicate:

$$\text{UBM}_k(u,i) : (\forall j : 1 \leq j \leq r : \text{last}(es(u,i,j), \text{proj}(mq_k(j), \{es(u,i,j)\}))$$

$$\vee \text{last}(f(j), mq_k(j)))$$

$\text{UBM}_k(u,i)$ evaluates to true at $P_k$ only if process $P_i$ has executed $17a$ and $P_k$ has received the $es(u,i,i)$ message, and every other process $P_j$ has also received the $es(u,i,i)$ message and broadcast an $es(u,i,j)$ message, as required by $17b$. Clearly, once $\text{UBM}_k(u,i)$ is true at every process $P_k$, then there are no undelivered basic messages that were sent prior to the broadcast of $es(u,i,i)$, due to $A4$. In order to notify $P_i$ of that, and to flush any messages destined to $P_i$, process $P_k$ executes:

$\text{IB}$: When $(\exists i,u : 1 \leq i \leq r : \text{UBM}_k(u,i))$ becomes true, then send an $ee(u,k)$ message to process $P_i$.

Then, $P_i$ can evaluate the following:

$$\text{INV}_i(a^{\#}_i) : (\forall j : 1 \leq j \leq r : \text{last}(ee(a^{\#}_i,j), \text{proj}(mq_i(j), \{ee(a^{\#}_i,j)\}))$$

$$\vee \text{last}(f(j), mq_i(j)))$$

If $\text{INV}_i(a^{\#}_i)$ evaluates to true at $P_i$ then the "flush" has terminated and $T_i$ did not change value during the flush. Since

$$\text{INV}_i(a^{\#}_i) \Rightarrow (\forall k : 1 \leq k \leq r : (\text{UBM}_k(a^{\#}_i,i) \vee \text{last}(f(j), mq_i(j))))$$

it follows that all basic messages that were sent prior to the $es(a^{\#}_i,i,i)$ -- before the "flush" -- have been delivered. This also means that any
a(j) message generated as a consequence of such a basic message will reach
P_i prior to the ee(a\#_i,j). Since T_i remained true during the whole
"flush", it must remain true forever. Hence,

INV_i(a\#_i) \Rightarrow (T_i \land \text{invariant}(T_i)),

and distributed termination is detected.

However, if process P_i detects distributed termination, it cannot
simply terminate; another process might need P_i to complete its "flush".
Notice that once DT_i becomes true for P_i, it must become true for all other
processes. This follows from the use of reliable broadcasts for all state
communication. Rather than allowing each process to independently detect
that distributed termination has occurred, the first process to detect
distributed termination must notify all the other processes. Otherwise,
"flushes" initiated by other processes could never be completed. allowed
to terminate because other processes might not have finished their
"flushes" yet. To do this, we define a message type st(i) to signal that
P_i has detected distributed termination.

19a: When DT_i becomes true, process P_i broadcasts a st(i) message and
terminates.

19b: P_i terminates upon receipt of a st(j) message from any process P_j.

Briefly summarized, the protocol at process P_i is as follows:
Step 1: Wait for T_i to become true and compute

n\#_i = \sum_{j=1}^{i} \text{length}(\text{proj}(\text{sq}_i(j), (a(j)))).

Step 2: Broadcast an \text{ee}(n\#_i, i, i) message. (17a)

Step 3: If \text{INV}_i(n\#_i) ever becomes true -- distributed termination has
been detected -- broadcast an st(i) message and terminate. (19a)
And concurrently, the following is executed:
- Upon receipt of \( \text{es}(u_j, j) \), broadcast \( \text{es}(u_j, i) \). (17b)
- If \( \exists i, j: \text{UNM}_k(u_j) \) evaluates to true, send \( \text{ee}(u_i, l) \) to process \( P_j \). (18)
- Upon receipt of a \( \text{st}(j) \) message, terminate. (19b)

2.2 An Implementation of Protocol 1

In this section we describe an implementation of this protocol. It is programmed using a programming notation that contains the data structures of PASCAL [JW74] and the control structures of Dijkstra's guarded commands [Di76]. In addition, to facilitate message passing, the following commands are added:

- \( \text{receive}_t m \) from \( k \) causes the invoker to be delayed until a message of type \( t \) from process \( k \) is received. The message is copied into \( m \).
- \( \text{anymsg}_t \) from \( k \) a boolean expression that evaluates to true if a subsequent \( \text{receive}_t \) from \( k \) would not cause delay.
- \( \text{send} m \) to \( k \) message \( m \) is sent to process \( k \).
- \( \text{broadcast} m \) message \( m \) is broadcast to all processes.

We also use

\[
\prod_{j \in S} B \rightarrow C \quad \text{where } S = \{a, b, \ldots, n\} \text{ to denote the guarded command set:}
\]

\[
\prod_{j \in S} B \rightarrow C \quad \text{where } S = \{a, b, \ldots, n\} \text{ to denote the guarded command set:}
\]

Although the protocol has been developed in terms of message queues of unbounded length, the relevant information can be encoded in a bounded size data structure. This is done by defining two arrays at each process \( P_i \).
type \( op_i \cdot 1 \ldots l \cdot 1 \ldots r \);

var state_i : array[1..r] of (active, quiet, failed) init (active);
link_i : array[op_i \cdot 1..r] of integer init (0);

The state of process \( P_j \) as perceived by \( P_i \) is stored in \( state_i[j] \);
link_i[j,k] is used to store the parameter of the last relevant message to
traverse the channel from \( P_j \) to \( P_i \) that was part of a "flush" initiated by \( P_k \).

A message queue \( mq_i(j) \) is encoded as follows:

\[
\begin{align*}
\text{state}_i[j] &= \begin{cases} 
\text{quiet} & \iff \text{last}(q(j) \cdot \text{proj}(mq_i(j) \cdot (q(j) \cdot f(j) \cdot a(j)))) \\
\text{failed} & \iff \text{last}(f(j) \cdot mq_i(j)) \\
\text{active} & \iff \text{otherwise}
\end{cases} \\
\text{link}_i[j,k] &= u \iff ((i=k \land \text{last}(ce(u,j) \cdot \text{proj}(mq_i(j) \cdot (ce(u,j)))) \lor \\
(i=k \land \text{last}(es(u,k,j) \cdot \text{proj}(mq_i(j) \cdot (es(u,k,j))))))
\end{align*}
\]

\[
\hat{a}_i^j = \sum_{j=1}^{r} \text{length}(\text{proj}(mq_i(j), a(j)))
\]

Note that by definition, at \( P_i \):

\[
(\forall k \land k \leq r : \text{state}_i[k] \& \text{active}) \equiv T_i
\]

\[
(\forall k : k \geq op_i : \text{link}_i[k,j] = \text{link}_i[j,j] \lor \text{state}_i[k] = \text{failed}) \\
\equiv \text{INV}_i(\text{link}_i[j,j],j)
\]

\[
(\forall k : k \geq op_i : \text{link}_i[k,i] = \hat{a}_i^j \lor \text{state}_i[k] = \text{failed}) \equiv \text{INV}_i(\hat{a}_i^j)
\]

So, if a process \( P_i \) is given in the following form:

\[
P_i : \mathcal{E}a_E1 \rightarrow C1 [E2 \rightarrow C2 [\ldots [E_n \rightarrow C_n a_d]
\]

then addition of the guards shown in figure 1 allows detection of
distributed termination.
\[ \text{Figure 0 -- Implementation of Protocol 1} \]
2.3 A More Sophisticated "Flush"

While the protocol just developed is fairly simple to understand, the "flush" operation can make heavy demands on the communications network. In this section, a somewhat less costly procedure is developed. Recall that

$$a_{i}^{\#} = \min_{j=1}^{\infty} \text{length} (\text{proj}(mq_{i}(j), \{a(j)\})).$$

Notice that A5 and A3 imply that upon termination, every process will eventually receive the same number of $a(j)$ messages. Therefore,

$$DT_{i} \Rightarrow (\forall j : 1 \leq j \leq r : a_{j}^{\#} = a_{i}^{\#}).$$

This can be exploited by modifying the previous protocol as follows. Rules II - I4, and I6 are unaltered; I7 - I9 are deleted, and I5 is replaced by:

I5a: Whenever $P_{i}$ becomes quiet, it broadcasts $q(a_{i}^{\#}, i)$.

I5b: Whenever the value of $a_{i}^{\#}$ changes while $P_{i}$ is quiet, $P_{i}$ broadcasts $q(a_{i}^{\#}, i)$.

Note that as a result of this rule a quiet process might broadcast a sequence of $q(a_{i}^{\#}, i)$ messages. But, the first parameter of each those messages differ -- they are strictly increasing.

$P_{i}$ eventually receives $q(a_{i}^{\#}, j)$ messages from every process $P_{j}$, if distributed termination occurs. Therefore, define

$$m_{i} = \max_{j} \text{ullast} (q(u_{j}), \text{proj}(mq_{i}(j), \{q(u_{j})\})).$$

so that at any time, $P_{i}$ can infer that it will subsequently receive additional $a(j)$ messages, if $m_{i} - a_{i}^{\#} > 0$. Then, the actual number of broadcasts made as a result of I5b can be reduced without adverse effect.
I5b: Whenever the value of \( a_{\mathcal{F}_1} \) changes while \( P_i \) is quiet, if \( a_{\mathcal{F}_1} \neq n_i \), then \( P_i \) broadcasts \( q(a_{\mathcal{F}_1},i) \).

As before, distributed termination has occurred if the last message received from every process is \( q(a_{\mathcal{F}_1},j) \), and no subsequent message deliveries will disturb that (ml and m2 of 2.1). The following predicate evaluates to true if the last message received from every process indicated that the process is quiet.

\[
SN_i(a_{\mathcal{F}_1}) = \left[ \forall j : 1 \leq j \leq r : \text{last}(q(a_{\mathcal{F}_1},j),\text{proj}(mq_i(j),\{q(a_{\mathcal{F}_1},j)\})) \right. \\
\left. \lor \text{last}(f(j),mq_i(j)) \right]
\]

If

\[
SN_i(a_{\mathcal{F}_1}) \land \text{invariant}(SN_i(a_{\mathcal{F}_1}))
\]
evaluates to true, then distributed termination has occurred. Thus, in order to detect distributed termination, a process \( P_i \) must wait until that assertion is true. Then, the following protocol can be used to determine if \( \text{invariant}(SN_i(a_{\mathcal{F}_1})) \) is true:

I7: Whenever \( SN_i(a_{\mathcal{F}_1}) \) evaluates to true, \( P_i \) broadcasts \( c(a_{\mathcal{F}_1},i) \).

Then, a process waits until the following predicate evaluates to true.

\[
CN_i(a_{\mathcal{F}_1}) = \left[ \forall j : 1 \leq j \leq r : \text{last}(c(a_{\mathcal{F}_1},j),\text{proj}(mq_i(j),\{c(a_{\mathcal{F}_1},i)\})) \right. \\
\left. \lor \text{last}(f(j),mq_i(j)) \right]
\]

A proof that the protocol is correct follows.

Processes that have not failed, always alternate between two states: active and quiet. Each time a process becomes active we will say it begins a new burst. Therefore, execution of a process involves one or more bursts, and strictly increasing burst numbers can be assigned to each burst within a process; the first time a process becomes active corresponds
to burst number one. Let $b^j_i$ denote the $j$-th burst of process $P_i$.

Burst $b^n_i$ of process $P_i$ is called the ancestor of burst $b^m_j$ of process $P_j$ if while $P_i$ executes in its $n$-th burst, it sends the basic message to $P_j$ that caused $P_j$ to start its $m$-th burst. For each burst, except for the first one of any process $P_i$ ($b^1_i$), there is a unique ancestor. A process can never be its own ancestor, due to I2.

Now, consider the graph of the ancestor relation. Any directed path in this graph can contain more than one burst from a given process, but only with strictly decreasing burst numbers. This is because in the execution of each process there is a positive time interval between the receipt of a message and the transmission of any subsequent message. And, for each burst the chain of ancestors goes back to a burst with number one -- back to the initialization.

To see that the protocol described above is sufficient for detecting distributed termination, assume that $CN_i(a^f_i)$ evaluates to true at process $P_i$. Suppose $P_i$ subsequently receives an $a(j)$ message from some process $P_j$ because $P_j$ started a new burst $b^n_j$. Now consider the directed path from $b^n_j$ to burst $b^1_k$ for some $P_k$. For any two adjacent bursts on this path -- say $b^a_s$ of process $P_s$ and $b^d_q$ of process $P_q$ -- such that $b^a_s$ is an ancestor of $b^d_q$ -- we will first prove:

$P_i$ received the $a(q)$ message signalling the start of $b^d_q$ or $P_i$ did not receive either the $a(s)$ message signalling the start of $b^a_s$ or the $a(q)$ message signalling the start of $b^d_q$.

Suppose $P_i$ did not receive the $a(q)$ message corresponding to the start of $b^d_q$. $P_i$ started burst $b^d_q$ upon receipt of a basic message $m$ from $P_s$ executing in $b^a_s$, due to the definition of ancestor.
Three cases must be considered:

Case 1: $P_s$ sent $m$ to $P_q$ before broadcasting $q(a^{\#_1}_s)$. Then, $P_q$ would have received $m$ before $q(a^{\#_1}_s)$, due to A4. Hence, $P_q$ would have become active and could not have broadcast the $c(a^{\#_1}_s,q)$ message, or else $P_i$ would have already received the $a(q)$ message corresponding to burst $b^d_q$. Since such a message would precede the $q(a^{\#_1}_i,q)$, this is a contradiction.

Case 2: $P_s$ sent the basic message to $P_q$ between the $q(a^{\#_1}_i,m)$ and the $c(a^{\#_1}_i,s)$ messages.

This is not possible, due to I7.

Case 3: $P_s$ sent the basic message to $P_q$ after the $c(a^{\#_1}_i,s)$ message.

This implies that $P_s$ became active after the broadcast of $c(a^{\#_1}_i,s)$. Thus, $P_i$ did not yet receive the $a(s)$ message corresponding to that event.

Since $P_i$ did not receive the $a(j)$ message corresponding to burst $b^n_j$ of process $P_j$, $P_i$ could not have received any $a(v)$ messages corresponding to bursts of processes $P_v$ that appear on the path from $b^n_j$ to $b^1_k$. This means that $P_i$ did not receive any messages from $P_k$, which contradicts the fact that $CN_i(a^{\#_1}_i)$ is true.

2.4 Implementation of Protocol 2

Define the following data structures for process $P_i$:

```plaintext
type op_i = (1..i-1, i+1..r);
var nam_i : array[1..r] of integer init (0);
fd_i : array[op_i] of boolean init (false);
\#_i : integer init (1);
```

\[ mn_i \rightarrow \text{integer init (0)}; \]
\[ quiet_i \rightarrow \text{boolean init (false)}; \]

A message queue \(mq_i(j)\) is encoded as follows:

\[
u \leftarrow \text{last}(q(u,j), \text{proj}(mq_i(j), \{q(u,j), c(u,j)\}))
\]
\[\text{nam}_i[j] = \begin{cases} u & \text{last}(c(u,j), \text{proj}(mq_i(j), \{q(u,j), c(u,j)\})) \\ 0 & \text{otherwise} \end{cases}
\]

\[
f_{d_i}[j] = \text{last}(f(j),mq_i(j))
\]
\[a_{\#_i} = \max_{j=1}^{r} \text{length}(\text{proj}(mq_i(j), \{a(j)\}))
\]
\[mn_i = \max \{\text{abs(nam}_i[j]) \mid j \}
\]
\[quiet_i \iff "P_i is quiet"
\]

Observe the following equivalences:

\[
SN_i(a_{\#_i}) \equiv (\forall j : 1 \leq j \lt r : \text{nam}_i[j] = a_{\#_i} \lor f_{d_i}[j])
\]
\[
CN_i(a_{\#_i}) \equiv (\forall j : 1 \leq j \lt r : \text{nam}_i[j] = -a_{\#_i} \lor f_{d_i}[j])
\]

Let \(\text{MAX}_i(x)\) denote the following piece of program:

\[
\text{if } x > \text{mn}_i \rightarrow \text{mn}_i := x;
\]
\[\square \text{otherwise } \rightarrow \text{skip}; \]
\[fi; \]

Everytime \(P_i\) becomes quiet, it executes:

\[quiet_i := \text{true}; \text{broadcast } q(a_{\#_i}, i); \quad /* E7, 15a */ \]
\[\text{nam}_i[i] := a_{\#_i}; \text{MAX}_i(a_{\#_i}); \quad /* E1, E6 */ \]

And, when \(P_i\) becomes active, it performs:

\[a_{\#_i} := a_{\#_i} + 1; quiet_i := \text{false}; \quad /* E5, E7 */ \]

The following guarded commands are added to the original program of \(P_i\).
anymsg\ q from P\_j \rightarrow \text{receive}
q(u, j) from P\_j: /* 15 */
\begin{align*}
\text{nam\_i} & : = u; \quad /* \text{E1} */ \\
\text{MAX\_i} & : = (u); \quad /* \text{E6} */ \\
\text{if } \text{SN\_i}(a^\#) & \text{ broadcast } \text{c}(a^\#_i, i) \quad /* \text{E7} */ \\
\text{nam\_i}(i) & : = -a^\#_i; \quad /* \text{E2} */ \\
\end{align*}
\text{if otherwise \ then skip; } \text{fi;}

anymsg\ f from P\_j \rightarrow \text{receive}
f(j) from P\_j: /* 16 */
\begin{align*}
\text{fd\_i} & : = \text{true}; \quad /* \text{E4} */ \\
\text{if } \text{SN\_i}(a^\#_i) & \text{ broadcast } \text{c}(a^\#_i, i) \quad /* \text{E7} */ \\
\text{nam\_i}(i) & : = -a^\#_i; \quad /* \text{E3} */ \\
\text{if else } & \text{CH\_i}(a^\#_i) \text{ \ then } \text{halt; } \text{fi; } \\
\text{if otherwise \ then skip; } \text{fi;}
\end{align*}

anymsg\ a from P\_j \rightarrow \text{receive}
a(j) from P\_j: /* 14 */
\begin{align*}
a^{\#}_i & : = a^{\#}_i + 1; \quad /* \text{E5} */ \\
\text{if } a^{\#}_i & \geq \text{mq\_i} \wedge \text{quiet\_i} \quad /* \text{E15b} */ \\
\text{broadcast } & \text{q}(a^{\#}_i, i); \quad /* \text{E1} */ \\
\text{nam\_i}(i) & : = a^{\#}_i; \quad /* \text{E6} */ \\
\text{MAX\_i}(a^{\#}_i); \quad /* \text{E5} */ \\
\text{if otherwise \ then skip; } \text{fi;}
\end{align*}

anymsg\ c from P\_j \rightarrow \text{receive}
c(u, j) from P\_j: /* 17 */
\begin{align*}
\text{nam\_i}(j) & : = -u; \quad /* \text{E2} */ \\
\text{if } \text{CN\_i}(a^\#_i') & \text{ \ then } \text{halt; } \text{fi; } \\
\text{if otherwise \ then skip; } \text{fi;}
\end{align*}

Figure 2 -- Implementation of Protocol 2
Chapter 3
Avoiding "flushes"

3.1 Derivation of a Protocol without "flush"

A special "flush" was necessary in the previous protocols to check the invariance of $T_i$ once it became true. Recall, however, that the only event that can cause a quiet process to become active is the receipt of a basic message, since all $a(j)$ messages result from such communication. The "flush" was necessary to determine if there were undelivered basic messages in the network.

Another approach for deciding whether $T_i$ will remain invariant involves keeping track of all basic communication as it occurs. This can be accomplished by requiring that the sender and the receiver of a basic message announce their respective actions. Hence, two new message types are defined:

$s(i,j)$ to signify that process $P_i$ sent a basic message to $P_j$.
$r(i,j)$ to indicate that process $P_i$ received a basic message from $P_j$.

Now the protocol of the previous chapter is revised as follows. First, step 14 is replaced by:

I4a: Whenever process $P_i$ sends a basic message to $P_j$, $P_i$ broadcasts "$s(i,j)$".

I4b: Whenever process $P_i$ receives a basic message from $P_j$, $P_i$ broadcasts "$r(j,i)$".

17 through 19 are no longer required, nor are the $a(j)$, $es(u,i,j)$ or $ee(u,i)$ message types. Actually, an $a(i)$ message is just a special type of $r(j,i)$ message. In the protocol of the last chapter, $P_i$ sent an $a(i)$ message whenever it became active. Due to II, this could happen only after
a quiet process receives a basic message. Hence, an \( r(j,k) \) message that process \( P_i \) receives is equivalent to an \( a(k) \) message if and only if \( \text{last}(q(k), m_{q_i}(k)) \). And, as before we must devise a scheme to evaluate invariant\( (T_i) \), in order to detect distributed termination.

A process \( P_i \) can evaluate the following predicate to determine if there are undelivered basic communications, and which processes are active:

\[
\text{MA}_i \equiv [\forall k : \exists i > r : (\forall j : 1 \leq j \leq r : \text{length}(\text{proj}(m_{q_i}(j), \{s(j,k)\})) = \text{length}(\text{proj}(m_{q_i}(k), \{r(j,k)\})))]
\]

Since for every basic communication from process \( P_j \) to \( P_k \) there is a \( s(j,k) \) and an \( r(j,k) \) message,\(^1\) it follows that if \( \text{MA}_i \) is true, then \( P_i \) received all matching pairs. Hence, all basic messages that \( P_i \) knows have been sent, have been received. Due to A4, the \( i \)-th \( s(j,k) \) message and the \( i \)-th \( r(j,k) \) message correspond to the same basic message sent by \( P_j \) and received by \( P_k^* \).

We claim

\[ T_i \land \text{MA}_i \Rightarrow T_i \land \text{invariant}(T_i) \]

and so this suffices as a protocol for detecting distributed termination.

A proof follows.

Suppose \( T_i \) and \( \text{MA}_i \) evaluate to true at \( P_i^* \). This implies that \( P_i \) received at least one \( q(j) \) message from every process \( P_j^* \). Let \( n_j \) denote the number of \( q(j) \) messages that \( P_i \) received from \( P_j^* \). Since there is exactly one \( q(j) \) message for every burst\(^2\) of \( P_j^* \), \( n_j \) also denotes the number of bursts of \( P_j \) that \( P_i \) knows about. Note, that \( n_j \geq 1 \) for all processes.

---

1 Except when \( P_k^* \) failed prior to the receipt of the basic message.

2 See chapter 2 for the definition of bursts.
P_j. We will prove that if T_i and MA_i is true for P_i, then all processes are quiet or failed, and that no process will subsequently become active.

Assume there is a process P_k that does become active again. This means that P_k will start a new burst b^a_k such that a > n_k. Mark the nodes of the ancestor relation graph so that all nodes representing a burst b^2_j of a process P_j, such that z ≤ n_j, are colored blue, and that all other nodes are red. There is at least one node of each color in the graph -- the node representing b^a_k is red and the nodes representing b^1_j for all j are blue. Further, there is a directed path from b^a_k to b^l_f for some process P_f. Thus, on this path, which starts from a red node and ends in a blue one, there exists a pair of adjacent vertices b^m_r and b^t_b where b^m_r is red and b^t_b is blue. Now, let b^m_r be a burst in process P_r and b^t_b be a burst in P_b. Hence, during b^t_b, P_b sent a basic message to P_r, causing P_r to start its next burst b^m_r. From 14a, P_b must have also broadcast an s(b,r) message at that time. This must have been received by P_i, because P_i has already received n_b q(b) messages, and the n_b-th q(b) message was sent after the s(b,r) message in question, due to 15. But P_i could not have received the corresponding s(b,i) message, due to our choice of b^t_b and b^m_r. This contradicts the assumption that MA_i is true at P_i. Hence, there does not exist such a process P_k that will subsequently become active.

1.2 Implementation of Protocol 3

Again, we describe an implementation using the notation introduced in section 2.2. The relevant information in the unbounded message queue is encoded as follows.

type op_i = (1..i-1, i+1..r);

var state_i : array[1..r] of (active, quiet, failed) init (active);
count\_i : array[1..r, 1..r] of integer init (0);

\[
\begin{cases}
\text{quiet} \iff \text{last}(q(j), proj(mq\_i(j), \{q(j), f(j)\})) \\
\text{failed} \iff \text{last}(f(j), mq\_i(j)) \\
\text{active} \iff \text{otherwise}
\end{cases}
\]

\[
\text{state}\_i[j] = \begin{cases}
\text{quiet} \iff \text{last}(q(j), proj(mq\_i(j), \{q(j), f(j)\})) \\
\text{failed} \iff \text{last}(f(j), mq\_i(j)) \\
\text{active} \iff \text{otherwise}
\end{cases}
\]

\[
\text{count}\_i[j,k] = \text{length}(proj(mq\_i(j), \{s(j,k)\})) - \\
\text{length}(proj(mq\_i(k), \{r(j,k)\}))
\]

(1.1)

(1.2)

(1.3)

Then, we have the obvious equivalences:

\[
T\_i \equiv (\forall j : 1 \leq j \leq r : \text{state}\_i[j] = \text{active})
\]

(1.5)

\[
M\_i \equiv (\forall k : 1 \leq k \leq r : (\forall j : 1 \leq j \leq r : \text{count}\_i[j,k] = 0) \lor \text{state}\_i[k] = \text{failed})
\]

(1.5)

\[
T\_i \equiv (\forall j : 1 \leq j \leq r : \text{state}\_i[j] = \text{active})
\]

(1.5)

\[
M\_i \equiv (\forall k : 1 \leq k \leq r : (\forall j : 1 \leq j \leq r : \text{count}\_i[j,k] = 0) \lor \text{state}\_i[k] = \text{failed})
\]

(1.5)

The original program of process P\_i is modified to comply with 11, 12, 13, 14a, 14b, 15 and 16. To accomplish this, the guarded commands shown in figure 3 are added.

\[\begin{array}{ll}
\text{[anymsg\_q from P\_j] + receive\_q q(j) from P\_j;} & /* 15 */ \\
\quad \text{state\_i[j] := quiet;} & /* 1.1 */ \\
\text{[anymsg\_f from P\_j] + receive\_f f(j) from P\_j;} & /* 16 */ \\
\quad \text{state\_i[j] := failed;} & /* 1.2 */ \\
\text{[anymsg\_o from P\_j] + receive\_o o(j,k) from P\_j;} & /* 14a */ \\
\quad \text{count\_i[j,k] := count\_i[j,k] + 1;} & /* 1.4 */ \\
\text{[anymsg\_r from P\_j] + receive\_r r(k,j) from P\_j;} & /* 14b */ \\
\quad \text{count\_i[k,j] := count\_i[k,j] - 1;} & /* 1.4 */ \\
\quad \text{state\_i[j] := active;} & /* 1.3 */ \\
\end{array}\]

\[\text{[T\_i ^ MA\_i] + halt;} & /* 1.6 */
\]

**Figure 3 -- Implementation of Protocol 3**
Chapter 4
"Diffusing Computations"

4.1 Introduction

In [DS79], Dijkstra and Scholten develop a solution to a simplified version of the problem we have been considering. In their problem formulation, an environment sends basic messages to some processes, thus activating them. These processes may then activate other processes, and so forth. This "spreading" of activity leads to a "diffusing computation". The simplified version of the problem is characterized by the following:

R1: No process can fail.

R2: Initially, one process -- the environment -- is active.

R3: Processes can send messages only to one process at a time (i.e., there are no broadcasts).

R4: (a) Every process returns an acknowledgment for each basic message it receives.

(b) A process becomes quiet if and only if it has received an acknowledgment for each basic message it has sent and it has acknowledged all basic messages it received.

Consequently, assumptions A1, A2 and A5 of chapter 1 are no longer relevant.

4.2 The Protocol

The protocol that a process must obey is as follows:

Il': A quiet process becomes active only upon receipt of a basic message.
I2': No quiet process sends basic messages.

This is no different than before. From R4, there are restrictions about when a process may become quiet — in contrast to our assumptions in chapter 2 and 3. Therefore, define variables \( \text{sts}_i \) (still to send by process \( P_i \)) and \( \text{ytr}_i \) (yet to receive by process \( P_i \)):

\[
\text{sts}_i = (\text{Nbr}_i - \text{Nas}_i)
\]

where \( \text{Nbr}_i \) is the number of basic messages \( P_i \) received, and \( \text{Nas}_i \) is the number acknowledgments \( P_i \) sent.

\[
\text{ytr}_i = (\text{Nbs}_i - \text{Nar}_i)
\]

where \( \text{Nbs}_i \) is the number of basic messages \( P_i \) sent, and \( \text{Nar}_i \) is the number of acknowledgments \( P_i \) received.

Then, R4 (b) is equivalent to:

I3': \( P_i \) becomes quiet if and only if \( \text{sts}_i = 0 \land \text{ytr}_i = 0 \).

Note, from R4 (a):

DS0: \( (\forall i: P_i \text{ is a process} : \text{sts}_i \geq 0 \land \text{ytr}_i \geq 0) \)

In [DS79] I1' through I3' are combined to form the following assertion, which is kept invariant:

DS1: \( (\forall i: P_i \text{ is a process other than the environment} : \text{sts}_i > 0 \lor \text{ytr}_i = 0). \)

R4 (a) is equivalent to

I4': Whenever process \( P_i \) receives a basic message from \( P_j \), \( P_i \) signals receipt by eventually sending an acknowledgment to \( P_j \).

A diffusing computation can be modelled using a finite directed graph. Such a graph will be called a computation graph. Each process \( P_i \) is represented by a vertex labelled \( P_i \). Initially, there are no edges in the graph. Hence, each of the processes constitutes its own connected
component. For each basic message sent by $P_i$ to $P_j$, a directed edge from $P_i$ to $P_j$ is added to the graph. This edge is then removed only after the receipt of the acknowledgment for the basic message. Thus, detecting distributed termination is equivalent to determining that there are no edges in the graph.

R2 states that the computation is started by one process — the environment. Once the outdegree of the node for the environment becomes zero, there are no active processes left in the system, and termination is detected, provided the following is invariant:

DS2: All nodes in the computation graph corresponding to active processes are reachable from the node representing the environment.

DS2 is a global invariant that depends on the state of the whole network. In order to keep it true, we must find a strategy that every process can independently follow.

In [DS79] the problem is solved in the following manner. A basic message that activates a quiet process is called an engagement message and the edge representing such a message is called engagement edge. Hence, a process has exactly one engagement edge at any time.

IS: A process $P_i$ can signal the receipt of an engagement message only if $(act_i = 1 \land ytr_i = 0)$.

This means that the acknowledgment corresponding to the receipt of an engagement message will always be the last one sent, and furthermore, acknowledging an engagement message is equivalent to what we have previously termed a q message. Note the following:
a) Each engagement edge connects two active processes, because the indegree of the destination and the outdegree of the source are greater than zero.

b) Engagement edges do not form cycles. The receipt of a basic message can only lead to an engagement edge if the receiving process is quiet and hence has outdegree zero.

c) Each active process has exactly one incoming engagement edge.

From this we can conclude that all active processes and the engagement edges connecting these processes form a tree with the environment as root.

Hence, there is a path from the environment to every active process.

4.1 Implementation

The following data structures are defined for the implementation.

type nb_i = \{P_j | P_j is a process with which P_i can directly communicate\};

var ats_i: array[nb_i] of integer init (0);
ytr_i: integer init (0);
egt_i: integer init (0);
t_i: integer init (0);

Define:

egt_i = j \iff P_j sent the engagement message to P_i, and the engagement message is not yet acknowledged. \hfill (E1)

ats_i[j] = n \iff P_j sent n basic messages (except engagement messages) to P_i that are not yet acknowledged. \hfill (E2)

t_i = \sum_{j \in nb_i} ats_i[j] \hfill (L3)

ytr_i = (Nlb_i - Nar_i) \hfill (I-)
\[ st_i = \begin{cases} ts_i + 1 & \iff eg_i = 0 \\ 0 & \iff eg_i = 0. \end{cases} \]

The program for a process \( P_i \) is changed as follows:

Every receive statement for a basic message that was sent by any process \( P_j \) is followed by:

\[
\begin{align*}
\text{if } eg_i &= 0 \Rightarrow eg_i := j; \\
\text{if } eg_i > 0 &\Rightarrow atsi[j] := atsi[j] + 1; \\
&\quad ts_i := ts_i + 1; \\
\end{align*}
\]

fi

And, every send statement for a basic message is followed by:

\[
ytr_i := ytr_i + 1
\]

The guards shown in figure 4 are added to the program.

\[
\begin{align*}
\text{if } atsi[j] &> 0 \Rightarrow \text{send } \text{ack}(i) \text{ to } P_j; \\
&\quad jcnb_i \\
&\quad atsi[j] := atsi[j] - 1; \\
&\quad ts_i := ts_i - 1; \\
\end{align*}
\]

\[
\begin{align*}
\text{if } ts_i &= 0 \land ytr_i = 0 \land eg_i = j \Rightarrow \text{send } \text{ack}(i) \text{ to } P_j; \\
&\quad jcnb_i \\
&\quad eg_i := 0; \\
\end{align*}
\]

\[
\begin{align*}
\text{anymsg } &\text{ ack from } P_j \Rightarrow \text{receive } \text{ack from } P_j; \\
&\quad jcnb_i \\
&\quad ytr_i := ytr_i - 1; \\
\end{align*}
\]

Figure 4 -- Implementation of the Restricted Protocol
Chapter 5
Restarting Failed Processes

2.1 Introduction

Our previous protocols have been concerned with removing failed processes from the system without jeopardizing the ability to detect distributed termination. We now consider the dual problem -- returning failed processes that have been repaired to active duty. A processor undergoes a restart when it rejoins the computing system after a failure. When a processor restarts, all processes that were running on it are restarted. Here, we consider how a restarted process can be added to the system, while retaining the ability to detect distributed termination. The complimentary problem of how to move portions of a computation in progress from a running process to the newly restarted one is not addressed.

A first, simple, solution would be to return processors to active duty only when a new cobegin statement is begun. While in many cases this would suffice, situations in which execution of a cobegin takes a substantial amount of time would be ill-served. Instead, a protocol is desired that will allow a processor to rejoin a computation as soon as it is running again. In order to do this the notion of time and causality in a distributed system is introduced.

A clock is a mapping from events to integers. \( c(E) \) is the time event \( E \) happened according to clock \( c \). A clock is valid if the mapping imposes a total ordering on events and if this ordering is consistent with causality. This means that if event \( E \) might be responsible for causing event \( F \), \( c(E) < c(F) \). A method for implementing valid clocks in distributed system is described in [La78a]. In this implementation, each process \( P_i \) has a
local clock $c_i$, and a protocol is defined to keep all clocks in close
synchronization. For the remainder of the chapter, we assume:

A6: A valid clock is available.

This clock is used to generate timestamps that can be put on
messages. A timestamp is the time the message is sent according to the
valid clock.

5.2 The Protocol for Restarting Failed Processes

The new protocol is similar to the one of section 2.3. It contains
the following rules:

I1: A quiet process becomes active only upon receipt of a basic message.

I2: No quiet process sends basic messages.

I3: An active process can become quiet at any time.

I4: When $P_i$ becomes quiet, it broadcasts $q(t,i)$, where $t$ is the time
according to the valid clock.

I4b: If $P_i$ is quiet and \( \text{last}(q(t,i), \text{proj}(mq_j(j),(q(t,i)))) \) then, upon
receipt of a $q(t',j)$ message from $P_j$ where $t' > t$, $P_i$ broadcasts
$q(t',i)$.

We will say that process $P_i$ is engaged to time $t$, if it broadcasts a $q(t,i)$
message (see also [L80]). Let $mt_i$ denote the time to which $P_i$ is engaged.
Note that a quiet process is always engaged to the highest time of those it
received on any $q$ message (I4b), and the time it last became quiet itself
(14). An active process is engaged to time 0 by convention.

The following new message type is defined to handle restarts:
$\text{rs}(t,j)$ signifies that process $P_j$ is to be restarted.
Then, we require

15: Whenever a failed process $P_i$ is to be restarted, $P_i$ broadcasts $rs(t,i)$, where $t$ is the time of the broadcast, according to a valid clock.

Rule 16 about failure of processes remains unchanged:

16: If process $P_i$ fails, then $f(i)$ is broadcast so as to arrive after all messages from that process.

Process $P_i$ suspects distributed termination when the last message it received from every non-failed process $P_j$ was a $q(mt_i,j)$ message. Hence, $P_i$ evaluates the following predicate:

$$ST_i(mt_i) = (\forall j: 1 \leq j \leq r : \text{last}(q(mt_i,j), \text{proj}(cq_i(j), \{q(mt_i,j)\}))$$

$$\lor \text{last}(f(j), \text{proj}(cq_i(j), \{f(j), rs(t,j)\}))$$

$$\land mt_i > 0)$$

If $ST_i(mt_i)$ evaluates to true at $P_i$, there can be no undelivered basic messages destined to $P_i$ that were sent before $mt_i$ in the network. Therefore, if $ST_i(mt_i)$ is an invariant of $ST_i(mt_i)$

$$ST_i(mt_i) \land \text{invariant}(ST_i(mt_i))$$

evaluates to true, distributed termination has occurred. Once $ST_i(mt_i)$ is true the following can be used to determine the invariance of $ST_i(mt_j)$.

17: Whenever $ST_i(mt_i)$ evaluates to true, $P_i$ broadcasts $c(mt_i,i)$.

We say, $P_i$ confirms its engagement to $mt_i$ when it executes 17. Hence, after all processes have confirmed engagement to a particular time $mt_i$, there can be no basic messages left in the network that were sent before $mt_i$. Furthermore, all processes were quiet at time $mt_i$ and so distributed termination has been detected -- if no processes restart. The invariance of $ST_i(mt_i)$ also depends on the fact that no $rs$ messages are received by
$P_i$ once $ST_i(mti)$ is true. Unfortunately, a restart could happen in the midst of the termination of the running processes. This is avoided if the following is kept invariant:

**RS:** An $rs(t,j)$ message of process $P_j$ will be received by all processes that have not failed before they terminate, or by none.

In order to keep RS invariant, we define the following new message type (synchronization):

$sy(t,i)$ signifies that process $P_i$ received a $rs(t,j)$ message from a process $P_j$ that is to be restarted.

Then, we require:

**IS:** $P_i$ can receive an $rs(t,j)$ message only if it is not in a confirmed engagement, i.e., $\neg ST_i(mti)$.\(^\dagger\) Upon receipt of an $rs(t,j)$ message, $P_i$ sends a $sy(t,i)$ to $P_j$.

Note, that by definition of $ST_i(mti)$, the receipt of an $rs(t,i)$ message breaks an engagement of $P_i$ to $mt_i$, hence $P_i$ never confirms this engagement and therefore, no process will terminate that is engaged to $mt_i$.

The following, last, rule takes care of old messages destined for a restarting process $P_i$ that were sent prior to the failure of $P_i$. Recall that there might be still such messages in the network.

**I9:** After broadcasting an $rs(t,i)$ message, a restarting process $P_i$ ignores all messages from $P_j$ until it receives the $sy(t,j)$ message or a $f(j)$ message.

To determine $invariant(ST_i(mti))$, every process $P_i$ evaluates the following predicate:

\(^\dagger\) Processes therefore require the ability to "peek" at the contents of a message before actually receiving it.
\[ CT_i(\text{mt}_i) \supseteq \forall j : 1 \leq j \leq r : \text{last}(c(\text{mt}_i, j), \text{proj}(mq_i(j), \{c(\text{mt}_i, i)\})) \]
\[ \vee \text{last}(f(j), \text{proj}(mq_i(j), \{f(j), rs(t, j)\})) \]

and we claim

\[ CT_i(\text{mt}_i) \equiv DT_i. \]

To prove this, assume \( CT_i(\text{mt}_i) \) evaluates to true at process \( P_i \) and suppose there are processes that become active again after time \( \text{mt}_i \). We do not need to consider restarted processes, due to RS. Choose \( P_j \) so that it is the first process that becomes active after \( \text{mt}_i \). This can only happen if \( P_j \) received a basic message from some process \( P_k \). Since \( P_j \) broadcast a \( c(\text{mt}_i, j) \) message, the basic message must have been sent by \( P_k \) after the \( q(\text{mt}_i, k) \) was broadcast. But this means that \( P_k \) was active after \( \text{mt}_i \), due to IL. Furthermore \( P_k \) was active before \( P_j \), since it sent the basic message that caused the activation of \( P_j \). This contradicts our choice of \( P_j \).

5.1 Implementation of protocol \( \Delta \)

Define the following data structures:

\[ \text{type op}_i = \{1..i-1, i+1..r\}; \]

\[ \text{var engtime}_i : \text{array}[1..r] \text{ of integer init } 1; \]

\[ \text{mt}_i : \text{integer init } 0; \]

A message queue \( mq_i[j] \) is encoded as follows:

\[ \text{engtime}_i[j] = \begin{cases} 0 \iff \text{last}(f(j), \text{proj}(mq_i(j), \{f(j), rs(t, j)\})) & (E1) \\ t \iff \text{last}(q(t, j), \text{proj}(mq_i(j), \{q(t, j), c(t, j), f(j)\})) & (E2) \\ -t \iff \text{last}(c(t, j), \text{proj}(mq_i(j), \{q(t, j), c(t, j), f(j)\})) & (E3) \\ 1 \iff \text{otherwise} & (E4) \end{cases} \]
\[ mt_i = \begin{cases} 
    0 & \iff P_i \text{ is active} \\
    \text{abs}(\text{engtime}_i[i]) & \iff P_i \text{ is quiet} 
\end{cases} \]  \(E5\)

Note the following equivalences:

\[ ST_i(mt_i) \equiv [(\forall j: 1 \leq j \leq r : \text{engtime}_i[j] = \text{engtime}_i[i] \lor \text{engtime}_i[j] = 0) \land \text{engtime}_i[i] = mt_i] \]  \(E7\)

\[ CT_i(mt_i) \equiv [(\forall j: 1 \leq j \leq r : \text{engtime}_i[j] = \text{engtime}_i[i] \lor \text{engtime}_i[j] = 0) \land \text{engtime}_i[i] = -mt_i] \]  \(E8\)

Define the following predicate:

\[ RR_i \equiv [(\forall j: 1 \leq j \leq r : (\text{abs}(\text{engtime}_i[j]) = \text{abs}(\text{engtime}_i[i]) \lor \text{engtime}_i[j] = 0) \land \text{mt}_i > 0] \]  \(E9\)

If \( RR_i \) evaluates to true at \( P_i \), then \( P_i \) is in a confirmed engagement and hence, is not allowed to receive restart messages.

The guarded commands of figure 5 are added to the original program at \( P_i \).
\( \text{anymsg}_q \text{ from } P_j \rightarrow \text{receive}_q q(t, j) \text{ from } P_j; \) /* I4 */
\text{engine}_q[j] := t; /* E2 */
\text{if } mt_i \neq 0 \land t > mt_i \rightarrow \text{broadcast } q(t, i); /* I4b */
\text{engine}_q[i] := t; /* E2 */
\text{mt}_i := t; /* E6 */
\text{if otherwise } \rightarrow \text{skip};
\text{fi;}
\text{if } ST_i(mt_i) \rightarrow \text{broadcast } c(mt_i, i); /* I7 */
\text{engine}_i[i] := -mt_i; /* E3 */
\text{if otherwise } \rightarrow \text{skip};
\text{fi;}

\( \text{anymsg}_f \text{ from } P_j \rightarrow \text{receive}_f f(j) \text{ from } P_j; \) /* I6 */
\text{engine}_f[j] := 0; /* E1 */
\text{if } ST_i(mt_i) \rightarrow \text{broadcast } c(mt_i, i); /* I7 */
\text{engine}_i[i] := -mt_i; /* E3 */
\text{if CT}_i(mt_i) \rightarrow \text{halt}; /* I8 */
\text{if otherwise } \rightarrow \text{skip};
\text{fi;}

\( \text{anymsg}_c \text{ from } P_j \rightarrow \text{receive}_c c(t, i) \text{ from } P_j; \) /* I5 */
\text{engine}_c[i] := -t_i /* I3 */
\text{if CT}_i(mt_i) \rightarrow \text{halt}; /* E3 */
\text{if otherwise } \rightarrow \text{skip};
\text{fi;}

\( \text{anymsg}_r \text{ from } P_j \rightarrow \text{receive}_r r_u(t, j) \text{ from } P_j; \) /* I5 */
\text{engine}_r[j] := 1; /* E4 */
\text{send } sy(t, i) \text{ to } P_j; /* I5 */

Figure 5 -- Implementation of Protocol 4
Chapter 6
Evaluation and Summary

Four protocols have been developed to allow detection of distributed
termination. The cost associated with each of these protocols varies,
depending how frequently individual processes become quiet. Therefore,
none of these protocols is optimal for all possible computations.

The protocol in 2.1, was based on an expensive "flush" of the network
once a process suspected distributed termination. This involves at least
r+1 broadcasts (the es messages and the st message) and an additional r-1
message transmissions (the ee messages). Furthermore, every process might
concurrently initiate a "flush", resulting in $r^2 + r$ broadcasts plus $r^2 - r$
message transmissions. This is quite costly, especially since distributed
termination might not have occurred yet; actually, in the worst case, a
broadcast of a q message can trigger "flushes" at every other process.
But, it is a cost incurred only when processes suspect distributed
termination has occurred.

In Section 2.3, a more sophisticated method of guaranteeing the
invariance of a global assertion was presented. There, the activation of a
previously quiet process might cause other quiet processes to broadcast at
most one additional q message. If a process becomes quiet, other processes
might broadcast one e message. Hence, in the worst case there can be $2r - 1$
broadcasts corresponding to any burst of a process. One can probably
expect a much better behavior on the average.

In contrast, the protocol developed in chapter 3 does not involve any
additional message exchange when distributed termination is being checked.
Instead, for each basic message that is sent, two broadcasts are made, one
by the sender of the basic message and the other one by the receiver of the basic message. As a result, processes can independently determine that distributed termination has occurred based on local message queues.

The protocol in chapter 5 is very similar in its behavior to that of section 2.3, except that it allows restart of failed processes.

Thus, in order to choose which protocol to use in a given situation, it is important to know the ratio of the number of basic messages exchanged to the number of times distributed termination is suspected by a process. If this ratio is high, then the protocol of section 2.3 (or 5.2) seems better; if it is low, the protocol of chapter 3 is probably better.

Quantification of "high" and "low" requires further knowledge of the cost of performing a broadcast relative to the cost of sending a message to a single process -- a cost that is highly dependent on the nature of the network. For example, note that in a true broadcast network, the basic message sent from \( P_i \) to \( P_j \) can serve as a the \( s(i,j) \) message as well.

The use of broadcasting is very important for all our protocols. It allows all processes to receive the same information about states of other processes, and therefore to come to the same conclusions as far as termination is concerned.
Chapter 7

Bibliography


[Sc79] Schneider, F.B. "Synchronization in Distributed Programs", TR 79-341, Department of Computer Science, Cornell University, 1979 (to appear in TOPLAS).
