DETECTING DISTRIBUTED TERMINATION WHEN PROCESSORS CAN FAIL*

C. W. Lermen
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:

A1: 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 A1 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, [MB76, 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.

1.1 The Problem: Detecting Distributed Termination.

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

$$S: \textbf{cobegin } P_1 \;//\; P_2 \;//\; \cdots P_r \textbf{ coend}$$

denotes the concurrent execution of processes $P_1, P_2, \cdots P_r$. S terminates when $P_1, P_2, \cdots P_r$ 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) $\bigwedge_{i=1}^{r} (\text{failed(processor i)} \vee R_i) \Rightarrow R$

Executing 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, FR80, Le80]. None of these algorithms handle processor failures, nor were there 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 [FR80] and [Le80] 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.

1.4 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:

proj(mq , s) ≡ For mq a message queue, and s a set of message types, proj(mq , s) is the message queue obtained from mq by deleting those messages with types not contained in s.

last(m , mq) ≡ true if m is the last message on mq.

length(mq) ≡ For mq a message queue, length(mq) 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

I4: Whenever process P_i becomes active it broadcasts a(i) before sending any other messages.

I5: Whenever process P_i becomes quiet it broadcasts q(i).

I6: 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.

$$DT_i: T_i \wedge \text{invariant}(T_i)$$

where

$$T_i \equiv [\forall j : 1 \le j \le r : \text{last}(f(j), mq_i(j)) \vee \text{last}(q(j), \text{proj}(mq_i(j), \{a(j), q(j)\}))]$$

1 Based on to A2, processor failures can be detected by a "time-out" 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.

invariant(T_i) $\equiv$ "T_i will not subsequently change value".

As will be seen below, the projection in the case 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., $T_i \wedge$ 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 I1, I2 (and I4). 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:

I7a: When T_i becomes true, process P_i broadcasts an es($a\#_i$,i,i) message, where $a\#_i = \sum_{j=1}^{r} \text{length}(\text{proj}(mq_i(j), \{a(j)\}))$.

I7b: Upon receipt of an es(u,j,j) message, process P_i broadcasts an es(u,j,i) message.

$a\#_i$ distinguishes between subsequent "flushes" initiated by the same process. I7a ensures that a message is transmitted between P_i and every other process; I7b causes the remaining messages to be transmitted.

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:

$$UBM_k(u,i) : [\forall j : 1 \le j \le r : last(es(u,i,j), proj(mq_k(j), \{es(u,i,j)\})) \vee last(f(j), mq_k(j))]$$

$UBM_k(u,i)$ evaluates to true at P_k only if process P_i has executed I7a 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 I7b. Clearly, once $UBM_k(u,i)$ 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:

I8: When $(\exists i,u : 1 \le i \le r : UBM_k(u,i))$ becomes true, then send an ee(u,k) message to process P_i.

Then, P_i can evaluate the following:

$$INV_i(a\#_i): [\forall j : 1 \le j \le r \wedge j \ne i : last(ee(a\#_i,j), proj(mq_i(j), \{ee(a\#_i,j)\})) \vee last(f(j), mq_i(j))]$$

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

$$INV_i(a\#_i) \Rightarrow [\forall k : 1 \le k \le r : (UBM_k(a\#_i,i) \vee last(f(j), mq_i(j))],$$

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 cc($a\#_i$,j). Since T_i remained true during the whole "flush", it must remain true forever. Hence,

$$INV_i(a\#_i) \Rightarrow (T_i \wedge \text{invariant}(T_i)).$$

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

$$a\#_i = \sum_{j=1}^{r} \text{length}(\text{proj}(mq_i(j), (a(j)))).$$

Step 2: Broadcast an rn($a\#_i$,1,i) message. (17a)

Step 3: If $INV_i(a\#_i)$ 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 es(u,j,j), broadcast es(u,j,i). (I7b)
- If $(\exists u,j : UBM_i(u,j))$ evaluates to true, send ee(u,i) to process P_j. (I8)
- Upon receipt of a st(j) message, terminate. (I9b)

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:

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.
anymsg$_t$ **from** k	a boolean expression that evaluates to true if a subsequent **receive**$_t$ from k would not cause delay.
send m **to** k	message m is sent to process k.
broadcast m	message m is broadcast to all processes.

We also use

$[]_{j \in S}\ B \rightarrow C$ where S={a,b,...,n} to denote the guarded command set:

$$B_a^j \rightarrow C_a^j \ [] \ B_b^j \rightarrow C_b^j \ [] \ \ldots \ [] \ B_n^j \rightarrow C_n^j$$

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 = 1..i-1, i+1..r;

var $state_i$: array[1..r] of (active, quiet, failed) init (active);

$link_i$: array[op_i, 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 u 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:

$$state_i[j] = \begin{cases} quiet & \Leftrightarrow last(q(j), proj(mq_i(j), \{q(j), f(j), a(j)\})) \quad (E1) \\ failed & \Leftrightarrow last(f(j), mq_i(j)) \quad (E2) \\ active & \Leftrightarrow otherwise \quad (E3) \end{cases}$$

$$link_i[j,k] = u \Leftrightarrow [(i=k \wedge last(ce(u,j), proj(mq_i(j), \{ce(u,j)\}))) \vee \quad (E4)$$

$$(i \neq k \wedge last(es(u,k,j), proj(mq_i(j), \{es(u,k,j)\})))] \quad (E4)$$

$$a\#_i = \sum_{j=1}^{r} length(proj(mq_i(j), \{a(j)\})) \quad (E6)$$

Note that by definition, at P_i:

$$(\forall k : 1 \leq k \leq r : state_i[k] \neq active) \equiv T_i \quad (E7)$$

$$(\forall k : k \in op_i : link_i[k,j] = link_i[j,j] \vee state_i[k] = failed)$$

$$\equiv UBH_i(link_i[j,j], j) \quad (E8)$$

$$(\forall k : k \in op_i : link_i[k,i] = a\#_i \vee state_i[k] = failed) \equiv INV_i(a\#_i) \quad (E9)$$

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

P_i: do B1 → C1 ▯ B2 → C2 ▯ ··· ▯ Bn → Cn od,

then addition of the guards shown in figure 1 allows detection of distributed termination.

```
[] anymsg_q from P_j  → receive_q q(j) from P_j;                  /* I5  */
j∈op_i                  state_i[j] := quiet;                      /* E1  */
                        if T_i → broadcast es(a#_i,i,i);          /* I7a */
                         [] otherwise → skip;
                        fi;

[] anymsg_f from P_j  → receive_f f(j) from P_j;                  /* I6  */
j∈op_i                  state_i[j] := failed;                     /* E2  */
                        if T_i → broadcast es(a#_i,i,i);          /* I7a */
                         [] otherwise → skip;
                        fi;
                        for k : 1 ≤ k ≤ r ∧ UBM_i(link_i[k,k],k)) do /* E8  */
                              → send ee(link_i[k,k],k) to P_k;    /* I8  */

[] anymsg_a from P_j  → receive_a a(j) from P_j;                  /* I4  */
j∈op_i                  state_i[j] := active;                     /* E3  */
                        a#_i := a#_i + 1;                         /* E6  */

[] anymsg_es from P_j → receive_es es(u,k,j) from P_j;            /* I7a */
j∈op_i                  if k = j → link_i[j,j] := u;              /* E5  */
                                   broadcast es(u,j,i);           /* I7b */
                         [] k ≠ j → link_i[j,k] := u;             /* E5  */
                         [] otherwise → skip;
                        fi;
                        if UBM_i(link_i[k,k],k) →                 /* I8  */
                                   send ee(link_i[k,k],k) to P_k; /* I8  */
                         [] otherwise → skip;
                        fi;

[] anymsg_ee from P_j → receive_ee ee(u,j) from P_j;              /* I8  */
j∈op_i                  link_i[j,i] := u;                         /* E4  */

[] INV_i(a#_i)        → broadcast st(i);                          /* I9a */
                        halt;

[] anymsg_st from P_j → receive_st st(j) from P_j;                /* I9a */
j∈op_i                  halt;                                     /* I9b */
```

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

$$a\#_i = \sum_{j=1}^{r} \text{length}(\text{proj}(mq_i(j), \{a(j)\})).$$

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

$$DT_i \Rightarrow (\forall j : 1 \le j \le r : a\#_j = a\#_i).$$

This can be exploited by modifying the previous protocol as follows. Rules I1 - 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

$$mn_i = \max_j \{u \mid \text{last}(q(u, j), \text{proj}(mq_i(j), \{q(u, j)\}))\}$$

so that at any time, P_i can infer that it will subsequently receive additional a(j) messages, if $mn_i - a\#_i > 0$. Thus, the actual number of broadcasts made as a result of I5b can be reduced without adverse effect.

I5b: Whenever the value of $a\#_i$ changes while P_i is quiet, if $a\#_i \geq mn_i$ then P_i broadcasts $q(a\#_i, i)$.

As before, distributed termination has occurred if the last message received from every process is $q(a\#_i, j)$, and no subsequent message deliveries will disturb that (m1 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.

$$SN_i(a\#_i) = [\forall j : 1 \leq j \leq r : last(q(a\#_i, j), proj(mq_i(j), \{q(a\#_i, j)\})) \vee last(f(j), mq_i(j))]$$

If

$$SN_i(a\#_i) \wedge invariant(SN_i(a\#_i))$$

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 $invariant(SN_i(a\#_i))$ is true:

I7: Whenever $SN_i(a\#_i)$ evaluates to true, P_i'' broadcasts $c(a\#_i, i)$.

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

$$CN_i(a\#_i) = [\forall j : 1 \leq j \leq r : last(c(a\#_i, j), proj(mq_i(j), \{c(a\#_i, i)\})) \vee last(f(j), mq_i(j))]$$

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_i^j denote the j-th burst of process P_i.

Burst b_i^n of process P_i is called the ancestor of burst b_j^m 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_i^1), 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 $CN_i(a\#_i)$ 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_j^n. Now consider the directed path from b_j^n to burst b_k^1 for some P_k. For any two adjacent bursts on this path -- say b_s^a of process P_s and b_q^d of process P_q, such that b_s^a is an ancestor of b_q^d -- we will first prove:

> P_i received the a(q) message signalling the start of b_q^d or P_i did not receive either the a(s) message signalling the start of b_s^a or the a(q) message signalling the start of b_q^d.

Suppose P_i did not receive the the a(q) message corresponding to the start of b_q^d. P_q started burst b_q^d upon receipt of a basic message m from P_s executing in b_s^a, due to the definition of ancestor.

Three cases must be considered:

Case 1: P_s sent m to P_q before broadcasting $q(a\#_i,s)$.

Then, P_q would have received m before $q(a\#_i,s)$, due to A4. Hence, P_q would have become active and could not have broadcast the $c(a\#_i,q)$ message, or else P_i would have already received the a(q) message corresponding to burst b_q^d, since such a message would preceed the $q(a\#_i,q)$. This is a contradiction.

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

This is not possible, due to I7.

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

This implies that P_s became active after the broadcast of $c(a\#_i,s)$. 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_j^n 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_j^n to b_k^1. This means that P_i did not receive any messages from P_k, which contradicts the fact that $CN_i(a\#_i)$ is true.

2.4 Implementation of Protocol 2

Define the following data structures for process P_i:

```
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);
     a#_i   : integer init (1);
```

mn_i : **integer init (0);**

$quiet_i$: **boolean init (false);**

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

$$nam_i[j] = \begin{cases} u \Leftrightarrow last(q(u,j).proj(mq_i(j).\{q(u,j).c(u,j)\})) & \text{(E1)} \\ -u \Leftrightarrow last(c(u,j).proj(mq_i(j).\{q(u,j).c(u,j)\})) & \text{(E2)} \\ 0 \Leftrightarrow otherwise & \text{(E3)} \end{cases}$$

$$fd_i[j] \equiv last(f(j).mq_i(j)) \quad \text{(E4)}$$

$$a\#_i = \sum_{j=1}^{r} length(proj(mq_i(j).\{a(j)\})) \quad \text{(E5)}$$

$$mn_i = \max_j \{abs(nam_i[j])\} \quad \text{(E6)}$$

$$quiet_i \Leftrightarrow "P_i \text{ is quiet}" \quad \text{(E7)}$$

Observe the following equivalences:

$$SN_i(a\#_i) \equiv (\forall j : 1 \le j \le r : nam_i[j] = a\#_i \vee fd_i[j]) \quad \text{(E8)}$$

$$CN_i(a\#_i) \equiv (\forall j : 1 \le j \le r : nam_i[j] = -a\#_i \vee fd_i[j]) \quad \text{(E9)}$$

Let $MAX_i(x)$ denote the following piece of program:

```
if x > mn_i      → mn_i := x;
 [] otherwise → skip;
fi;
```

Everytime P_i becomes quiet, it executes:

```
quiet_i := true; broadcast q(a#_i.i);        /* E7, I5a */
nam_i[i] := a#_i; MAX_i(a#_i);               /* E1, E6  */
```

And, when P_i becomes active, it performs:

```
a#_i := a#_i + 1; quiet_i := false;          /* E5, E7  */
```

The following guarded commands are added to the original program of P_i.

```
[] anymsg_q from P_j → receive_q q(u,j) from P_j;                    /* I5 */
j∈op_i                 nam_i[j] := u;                                /* E1 */
                       MAX_i(u);                                     /* E6 */
                       if SN_i(a#_i) → broadcast c(a#_i,i)           /* I7 */
                                       nam_i[i] := -a#_i;            /* E2 */
                        [] otherwise → skip;
                       fi;

[] anymsg_f from P_j → receive_f f(j) from P_j;                      /* I6 */
j∈op_i                 fd_i[j] := true;                              /* E4 */
                       if SN_i(a#_i) → broadcast c(a#_i,i)           /* I7 */
                                       nam_i[i] := -a#_i;            /* E3 */
                        [] CN_i(a#_i) → halt;                        /* E9 */
                        [] otherwise → skip;
                       fi;

[] anymsg_a from P_j → receive_a a(j) from P_j;                      /* I4 */
j∈op_i                 a#_i := a#_i + 1;                             /* E5 */
                       if a#_i ≥ nn_i ∧ quiet_i →
                                       broadcast q(a#_i,i);          /* I5b */
                                       nam_i[i] := a#_i;             /* E1 */
                                       MAX_i(a#_i);                  /* E6 */
                        [] otherwise → skip;
                       fi;

[] anymsg_c from P_j → receive_c c(u,j) from P_j;                    /* I7 */
j∈op_i                 nam_i[j] := -u;                               /* E2 */
                       if CN_i(a#_i)' → halt;                        /* E9 */
                        [] otherwise → skip;
                       fi;
```

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 s(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 I4 is replaced by:

I4a: Whenever process P_i sends a basic messages 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)"

I7 through I9 are no longer required, nor are the s(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 I1, 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 s(k) message if and only if last(q(k),mq_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:

$$MA_i \equiv [\forall k : 1 \le k \le r : (\forall j : 1 \le j \le r : length(proj(mq_i(j),\{s(j,k)\})) = length(proj(mq_i(k),\{r(j,k)\}))) \vee last(f(k),mq_i(k)))].$$

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 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

$$T_i \wedge MA_i \Rightarrow T_i \wedge invariant(T_i)$$

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

Suppose T_i and 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 $n_j \ge 1$ 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_k^a such that $a > n_k$. Mark the nodes of the ancestor relation graph so that all nodes representing a burst b_j^z of a process P_j, such that $z \leq n_j$, 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_k^e is red and the nodes representing b_j^1 for all j are blue. Further, there is a directed path from b_k^a to b_f^1 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_r^m and b_b^t where b_r^m is red and b_b^t is blue. Now, let b_r^m be a burst in process P_r and b_b^t be a burst in P_b. Hence, during b_b^t, P_b sent a basic message to P_r, causing P_r to start its next burst b_r^m. From I4a, 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 I5. But P_i could not have received the corresponding s(b,r) message, due to our choice of b_b^t and b_r^m. 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.

3.2 Implementation of Protocol 3

Again, we describe an implementation using the notation introduced in section 2.2. The relevant information in the unbounded message queues 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);

$$state_i[j] = \begin{cases} quiet \Leftrightarrow last(q(j).proj(mq_i(j).\{q(j).f(j)\})) & (E1) \\ failed \Leftrightarrow last(f(j).mq_i(j)) & (E2) \\ active \Leftrightarrow otherwise & (E3) \end{cases}$$

$$count_i[j,k] = length(proj(mq_i(j).\{s(j,k)\})) - length(proj(mq_i(k).\{r(j,k)\})). \quad (E4)$$

Then, we have the obvious equivalences:

$$T_i \equiv (\forall j : 1 \le j \le r : state_i[j] \neq active) \quad (E5)$$

$$MA_i \equiv (\forall k : 1 \le k \le r : (\forall j : 1 \le j \le r : count_i[j,k] = 0) \vee state_i[k] = failed). \quad (E6)$$

The original program of process P_i is modified to comply with I1, I2, I3, I4a, I4b, I5 and I6. To accomplish this, the guarded commands shown in figure 3 are added.

```
[] anymsg_q from P_j  →  receive_q q(j) from P_j:          /* I5  */
jcop_i                   state_i[j] := quiet;              /* E1  */

[] anymsg_f from P_j  →  receive_f f(j) from P_j:          /* I6  */
jcop_i                   state_i[j] := failed;             /* E2  */

[] anymsg_s from P_j  →  receive_s s(j,k) from P_j:        /* I4a */
jcop_i                   count_i[j,k] := count_i[j,k] + 1; /* E4  */

[] anymsg_r from P_j  →  receive_r r(k,j) from P_j:        /* I4b */
jcop_i                   count_i[k,j] := count_i[k,j] - 1; /* E4  */
                         state_i[j] := active;             /* E3  */

[] T_i ∧ MA_i  →         halt;                             /* E6  */
```

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:

I1': 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 sts_i (still to send by process P_i) and ytr_i (yet to receive by proces P_i):

$sts_i \equiv (Nbr_i - Nas_i)$

where Nbr_i is the number of basic messages P_i received,

and Nas_i is the number acknowledgments P_i sent.

$ytr_i \equiv (Nbs_i - Nar_i)$

where Nbs_i is the number of basic messages P_i sent.

and 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 $(sts_i = 0 \wedge ytr_i = 0)$.

Note, from R4 (a):

DS0: $(\forall i : P_i \text{ is a process} : sts_i \geq 0 \wedge ytr_i \geq 0)$

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

DS1: $(\forall i : P_i \text{ is a process other than the environment} : sts_i > 0 \vee ytr_i = 0)$.

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.

I5': A process P_i can signal the receipt of an engagement message only if $(\text{nts}_i = 1 \wedge \text{ytr}_i = 0)$.

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.

4.3 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);
ts_i : integer init (0);

Define:

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

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

$$ts_i = \sum_{j \in nb_i} ats_i[j] \quad (E3)$$

$$ytr_i \equiv (Nin_i - Nor_i) \quad (E4)$$

Hence,

$$sts_i = \begin{cases} ts_i + 1 & \Leftrightarrow egt_i \neq 0 \\ 0 & \Leftrightarrow egt_i = 0. \end{cases}$$

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:

```
if egt_i = 0 → egt_i := j;                          /* E1 */
 □ egt_i > 0 → sts_i[j] := sts_i[j] + 1;            /* E2 */
               ts_i := ts_i + 1;                    /* E3 */
fi
```

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

```
ytr_i := ytr_i + 1                                  /* E4 */
```

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

```
□ sts_i[j] > 0 → send ack(i) to P_j;                /* I4' */
jenb_i           sts_i[j] := sts_i[j] - 1;          /* E2  */
                 ts_i := ts_i - 1;                  /* E3  */

□ ts_i = 0 ∧ ytr_i = 0 ∧ egt_i = j → send ack(i) to P_j;   /* I5' */
jenb_i                               egt_i := 0;          /* E1  */

□ anymsg_ack from P_j → receive_ack from P_j;       /* I4' */
jenb_i                 ytr_i := ytr_i - 1;          /* E4  */
```

Figure 4 -- Implementation of the Restricted Protocol

Chapter 5

Restarting Failed Processes

5.1 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.

I4a: 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 $last(q(t,i), proj(mq_i(j), \{q(t,i)\}))$ 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 [Le80]). 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 (I4). An active process is engaged to time 0 by convention.

The following new message type is defined to handle restarts:

$rs(t,j)$ signifies that process P_j is to be restarted.

Then, we require

I5: 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 I6 about failure of processes remains unchanged:

I6: 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:

$$ST_i(mt_i) \equiv [\ (\forall j : 1 \le j \le r : last(q(mt_i,j),proj(mq_i(j),\{q(mt_i,i)\})) \vee last(f(j),proj(mq_i(j),\{f(j),rs(t,j)\})))$$
$$\wedge\ mt_i > 0]$$

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

$$ST_i(mt_i) \wedge invariant(ST_i(mt_i))$$

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

I7: Whenever $ST_i(mt_i)$ evaluates to true, P_i broadcasts $c(mt_i,i)$.

We say, P_i <u>confirms</u> its engagement to mt_i when it executes I7. 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 [illegible]

P_i, once $ST_i(mt_i)$ 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:

I8: P_i can receive an rs(t,j) message only if it is not in a confirmed engagement, i.e. $\neg ST_i(mt_i)$.[1] 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($ST_i(mt_i)$), every process P_i evaluates the following predicate:

---

1 Processes therefore require the ability to "peek" at the contents of a message before actually receiving it.

$$CT_i(mt_i) \equiv [\forall j : 1 \le j \le r : last(c(mt_i.j).proj(mq_i(j).\{c(mt_i.i)\})) \vee last(f(j).proj(mq_i(j).\{f(j).rs(t.j)\}))]$$

and we claim

$$CT_i(mt_i) \equiv DT_i.$$

To prove this, assume $CT_i(mt_i)$ evaluates to true at process P_i and suppose there are processes that become active again after time 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 mt_i. This can only happen if P_j received a basic message from some process P_k. Since P_j broadcast a $c(mt_i.j)$ message, the basic message must have sent by P_k after the $q(mt_i.k)$ was broadcast. But this means that P_k was active after mt_i, due to I1. 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.

5.3 Implementation of protocol 4

Define the following data structures:

```
type op_i = (1 .. i-1 , i+1 .. r);
var . engtime_i : array[1..r] of integer init (1);
      mt_i : integer init (0);
```

A message queue $mq_i[j]$ is encoded as follows:

$$engtime_i[j] = \begin{cases} 0 \Leftrightarrow last(f(j).proj(mq_i(j).\{f(j).rs(t'.j)\})) & (E1) \\ t \Leftrightarrow last(q(t.j).proj(mq_i(j).\{q(t.j).c(t.j).f(j)\})) & (E2) \\ -t \Leftrightarrow last(c(t.j).proj(mq_i(j).\{q(t.j).c(t.j).f(j)\})) & (E3) \\ 1 \Leftrightarrow otherwise & (E4) \end{cases}$$

$$mt_i = \begin{cases} 0 \Leftrightarrow P_i \text{ is active} & \text{(E5)} \\ abs(engtime_i[i]) \Leftrightarrow P_i \text{ is quiet} & \text{(E6)} \end{cases}$$

Note the following equivalences:

$$ST_i(mt_i) \equiv [(\forall j : 1 \le j \le r : engtime_i[j] = engtime_i[i] \vee engtime_i[j] = 0) \wedge engtime_i[i] = mt_i] \quad \text{(E7)}$$

$$CT_i(mt_i) \equiv [(\forall j : 1 \le j \le r : engtime_i[j] = engtime_i[i] \vee engtime_i[j] = 0) \wedge engtime_i[i] = -mt_i] \quad \text{(E8)}$$

Define the following predicate:

$$RR_i \equiv [\forall j : 1 \le j \le r : (abs(engtime_i[j]) = abs(engtime_i[i]) \vee engtime_i[j] = 0) \wedge mt_i > 0] \quad \text{(E9)}$$

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.

```
[] anymsg_q from P_j → receive_q q(t,j) from P_j;                    /* I4  */
j∈op_i                 engtime_i[j] := t;                            /* E2  */
                       if mt_i≠0 ∧ t>mt_i → broadcast q(t,i);        /* I4b */
                                            engtime_i[i] := t;       /* E2  */
                                            mt_i := t;               /* E6  */
                        [] otherwise → skip;
                       fi;
                       if ST_i(mt_i) → broadcast c(mt_i,i);          /* I7  */
                                       engtime_i[i] := -mt_i;        /* E3  */
                        [] otherwise → skip;
                       fi;

[] anymsg_f from P_j → receive_f f(j) from P_j;                      /* I6  */
j∈op_i                 engtime_i[j] := 0;                            /* E1  */
                       if ST_i(mt_i) → broadcast c(mt_i,i);          /* I7  */
                                       engtime_i[i] := -mt_i;        /* E3  */
                        [] CT_i(mt_i) → halt;                        /* E8  */
                        [] otherwise → skip;
                       fi;

[] anymsg_c from P_j → receive_c c(t,j) from P_j;                    /* I5  */
j∈op_i                 engtime_i[j] := -t;                           /* E3  */
                       if CT_i(mt_i) → halt;                         /* E8  */
                        [] otherwise → skip;
                       fi;

[] (¬RR_i) ∧ anymsg_rm from P_j → receive_rm rm(t,j) from P_j;       /* I5  */
                                  engtime_i[j] := 1;                 /* E4  */
                                  send sy(t,i) 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 c 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 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

[Ab70] Abramson, N. "The Aloha System", AFIPS Conference Proceedings Vol. 37, 1970 FJCC, pp. 281-285.

[Di76] Dijkstra, E.W. "A Discipline of Programming", Prentice-Hall, N.J. 1976.

[DS79] Dijkstra, E.W. and C.S. Scholten "Termination Detection for Diffusing Computations", EWD-687a, January 1979

[FA73] Farber, D., J. Feldman, F. Heinrich, M. Hopwood, K. Larson, D. Lorris and L.Rowe "The Distributed Computing System", Proceedings of CompCon 73, February 1973, pp 31-34.

[FR80] Francez, N. "Distributed Termination", TOPLAS 2, 1 January 1980, pp 42-55.

[Ho78] Hoare, C.A.R. "Communicating Sequential Processes", CACM 21, 8 August 1978, pp 666-677.

[JW74] Jensen, K. and N. Wirth "PASCAL User Manual and Report", 2nd edition, Springer Verlag 1974

[La78a] Lamport, L. "Time, Clocks and the Ordering of Events in a Distributed System", CACM 21, 7 July 1978, pp 558-565.

[La78b] Lamport, L., "The Implementation of Reliable Distributed Multiprocess Systems", Computer Networks 2, pp 95-114.

[Le80] Levin, G., "Proof Rules for Communicating Sequential Processes", Ph.D. Thesis, Department of Computer Science, Cornell University, June 1980.

[MB76] Metcalfe,R.M. and D.R. Boggs "Ethernet: Distributed Packet Switching for Local Computer Networks". CACM 19, 7 July 1976, pp 395-403.

[RW70] Roberts, L.G. and B.D. Wessler "Computer Network Development to Achieve Resource Sharing". AFIPS SJCC Proceedings, 1970, pp 543-549.

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

[SS80] Schneider, F.B. and R.D. Schlichting "Fast Reliable Broadcasts", Tech. Report, Department of Computer Science, Cornell University, October 1980.