CHAPTER 7 Termination detection

7.1 Introduction

In distributed processing systems, a problem is typically solved in a distributed manner with the cooperation of a number of processes. In such an environment, inferring if a distributed computation has ended is essential so that the results produced by the computation can be used. Also, in some applications, the problem to be solved is divided into many subproblems, and the execution of a subproblem cannot begin until the execution of the previous subproblem is complete. Hence, it is necessary to determine when the execution of a particular subproblem has ended so that the execution of the next subproblem may begin. Therefore, a fundamental problem in distributed systems is to determine if a distributed computation has terminated.

The detection of the termination of a distributed computation is non-trivial since no process has complete knowledge of the global state, and global time does not exist. A distributed computation is considered to be globally terminated if every process is locally terminated and there is no message in transit between any processes. A "locally terminated" state is a state in which a process has finished its computation and will not restart any action unless it receives a message. In the termination detection problem, a particular process (or all of the processes) must infer when the underlying computation has terminated.

When we are interested in inferring when the underlying computation has ended, a termination detection algorithm is used for this purpose. In such situations, there are two distributed computations taking place in the distributed system, namely, the *underlying computation* and the *termination detection algorithm*. Messages used in the underlying computation are called *basic* messages, and messages used for the purpose of termination detection (by a termination detection algorithm) are called *control* messages.

A termination detection (TD) algorithm must ensure the following:

1. Execution of a TD algorithm cannot indefinitely delay the underlying computation; that is, execution of the termination detection algorithm must not freeze the underlying computation.
2. The termination detection algorithm must not require addition of new communication channels between processes.

7.2 System model of a distributed computation

A distributed computation consists of a fixed set of processes that communicate solely by message passing. All messages are received correctly after an arbitrary but finite delay. Communication is *asynchronous*, i.e., a process never waits for the receiver to be ready before sending a message. Messages sent over the same communication channel may not obey the FIFO ordering.

A distributed computation has the following characteristics:

1. At any given time during execution of the distributed computation, a process can be in only one of the two states: *active*, where it is doing local computation and *idle*, where the process has (temporarily) finished the execution of its local computation and will be reactivated only on the receipt of a message from another process. The active and idle states are also called the *busy* and *passive* states, respectively.
2. An active process can become idle at any time. This corresponds to the situation where the process has completed its local computation and has processed all received messages.
3. An idle process can become active only on the receipt of a message from another process. Thus, an idle process cannot spontaneously become active (except when the distributed computation begins execution).
4. Only active processes can send messages. (Since we are not concerned with the initialization problem, we assume that all processes are initially idle and a message arrives from outside the system to start the computation.)
5. A message can be received by a process when the process is in either of the two states, i.e., *active* or *idle*. On the receipt of a message, an *idle* process becomes *active*.
6. The sending of a message and the receipt of a message occur as atomic actions.

We restrict our discussion to executions in which every process eventually becomes idle, although this property is in general undecidable. If a termination detection algorithm is applied to a distributed computation in which some processes remain in their active states forever, the TD algorithm itself will not terminate.

Definition of termination detection

Let $p_i(t)$ denote the state (active or idle) of process p_i at instant t and $c_{i,j}(t)$ denote the number of messages in transit in the channel at instant t from process p_i to process p_j. A distributed computation is said to be terminated at time instant t_0 iff:

$$(\forall i :: p_i(t_0) = idle) \wedge (\forall i, j :: c_{i,j}(t_0) = 0).$$

7.3 Termination detection using distributed snapshots

The algorithm uses the fact that a consistent snapshot of a distributed system captures stable properties. Termination of a distributed computation is a stable property. Thus, if a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation.

The algorithm assumes that there is a logical bidirectional communication channel between every pair of processes. Communication channels are reliable but non-FIFO. Message delay is arbitrary but finite.

7.3.1 Informal description

The main idea behind the algorithm is as follows: when a computation terminates, there must exist a unique process which became idle last. When a process goes from active to idle, it issues a request to all other processes to take a local snapshot, and also requests itself to take a local snapshot. When a process receives the request, if it agrees that the requester became idle before itself, it grants the request by taking a local snapshot for the request. A request is said to be *successful* if all processes have taken a local snapshot for it. The requester or any external agent may collect all the local snapshots of a request. If a request is successful, a global snapshot of the request can thus be obtained and the recorded state will indicate termination of the computation, viz., in the recorded snapshot, all the processes are idle and there is no message in transit to any of the processes.

7.3.2 Formal description

The algorithm needs logical time to order the requests. Each process i maintains an *logical clock* denoted by x, which is initialized to zero at the start of the computation. A process increments its x by one each time it becomes idle. A basic message sent by a process at its logical time x is of the form $B(x)$. A control message that requests processes to take local snapshot issued by process i at its logical time x is of the form $R(x, i)$. Each process synchronizes its logical clock x loosely with the logical clocks x's on other processes in such a way that it is the maximum of clock values ever received or sent in messages. Besides logical clock x, a process maintains a variable k such that when the process is idle, (x,k) is the maximum of the values (x, k) on all messages *R(x, k)* ever received or sent by the process. Logical time is compared as follows: $(x, k) > (x', k')$ iff $(x > x')$ or $((x = x')$ and $(k > k'))$, i.e., a tie between x and x' is broken by the process identification numbers k and k'.

The algorithm is defined by the following four rules [8]. We use guarded statements to express the conditions and actions. Each process i applies one of the rules whenever it is applicable.

```
R1: When process $i$ is active, it may send a basic message to process $j$ at
    any time by doing
                        send a $B(x)$ to $j$.

R2: Upon receiving a $B(x')$, process $i$ does
                    let $x := x' + 1$;
                    if($i$ is idle) $\rightarrow$ go active.

R3: When process $i$ goes idle, it does
            let $x := x + 1$;
            let $k := i$;
            send message $R(x, k)$ to all other processes;
            take a local snapshot for the request by $R(x, k)$.

R4: Upon receiving message $R(x', k')$, process $i$ does
      [$((x', k') > (x, k)) \wedge (i$ is idle$) \rightarrow$ let$(x, k) := (x', k')$;
                    take a local snapshot for the request by$R(x', k')$;
      $\Box$
      $((x', k') \leq (x, k)) \wedge (i$ is idle$) \rightarrow$ do nothing;
      $\Box$
      $(i$ is active$) \rightarrow$ let $x := max(x', x)$].
```

7.3.3 Discussion

As per rule R1, when a process sends a basic message to any other process, it sends its logical clock value in the message. From rule R2, when a process receives a basic message, it updates its logical clock based on the clock value contained in the message. Rule R3 states that when a process becomes idle, it updates its local clock, sends a request for snapshot $R(x, k)$ to every other process, and takes a local snapshot for this request.

Rule R4 is the most interesting. On the receipt of a message $R(x', k')$, the process takes a local snapshot if it is idle and $(x', k') > (x, k)$, i.e., timing in the message is later than the local time at the process, implying that the sender of $R(x', k')$ terminated after this process. In this case, it is likely that the sender is the last process to terminate and thus, the receiving process takes a snapshot for it. Because of this action, every process will eventually take a local snapshot for the last request when the computation has terminated, that is, the request by the latest process to terminate will become successful.

In the second case, $(x', k') \leq (x, k)$, implying that the sender of $R(x', k')$ terminated before this process. Hence, the sender of $R(x', k')$ cannot be the last process to terminate. Thus, the receiving process does not take a snapshot for it. In the third case, the receiving process has not even terminated. Hence, the sender of $R(x', k')$ cannot be the last process to terminate and no snapshot is taken.

The last process to terminate will have the largest clock value. Therefore, every process will take a snapshot for it; however, it will not take a snapshot for any other process.

7.4 Termination detection by weight throwing

In termination detection by weight throwing, a process called *controlling agent*[1] monitors the computation. A communication channel exists between each of the processes and the controlling agent and also between every pair of processes.

Basic idea

Initially, all processes are in the idle state. The weight at each process is zero and the weight at the controlling agent is 1. The computation starts when the controlling agent sends a basic message to one of the processes. The process becomes active and the computation starts. A non-zero weight W $(0 < W \leq 1)$ is assigned to each process in the active state and to each message in transit in the following manner: When a process sends a message, it sends a part of its weight in the message. When a process receives a message, it add the weight received in the message to its weight. Thus, the sum of weights on all the processes and on all the messages in trasit

[1] The controlling agent can be one of the processes in the computation.

is always 1. When a process becomes passive, it sends its weight to the controlling agent in a control message, which the controlling agent adds to its weight. The controlling agent concludes termination if its weight becomes 1.

Notation

- The weight on the controlling agent and a process is in general represented by W.
- $B(DW)$: A basic message B is sent as a part of the computation, where DW is the weight assigned to it.
- $C(DW)$: A control message C is sent from a process to the controlling agent where DW is the weight assigned to it.

7.4.1 Formal description

The algorithm is defined by the following four rules [9]:

Rule 1: The controlling agent or an active process may send a basic message to one of the processes, say P, by splitting its weight W into $W1$ and $W2$ such that $W1 + W2 = W$, $W1 > 0$ and $W2 > 0$. It then assigns its weight $W := W1$ and sends a basic message $B(DW := W2)$ to P.

Rule 2: On the receipt of the message $B(DW)$, process P adds DW to its weight W $(W := W + DW)$. If the receiving process is in the idle state, it becomes active.

Rule 3: A process switches from the active state to the idle state at any time by sending a control message $C(DW := W)$ to the controlling agent and making its weight $W := 0$.

Rule 4: On the receipt of a message $C(DW)$, the controlling agent adds DW to its weight $(W := W + DW)$. If W = 1, then it concludes that the computation has terminated.

7.4.2 Correctness of the algorithm

To prove the correctness of the algorithm, the following sets are defined:

A: set of weights on all active processes;
B: set of weights on all basic messages in transit;
C: set of weights on all control messages in transit;
W_c: weight on the controlling agent.

Two invariants I_1 and I_2 are defined for the algorithm:

I_1: $W_c + \sum\limits_{W \in (A \cup B \cup C)} W = 1$.

I_2: $\forall W \in (A \cup B \cup C)$, $W > 0$.

Invariant I_1 states that the sum of weights at the controlling process, at all active processes, on all basic messages in transit, and on all control messages in transit is always equal to 1. Invariant I_2 states that weight at each active process, on each basic message in transit, and on each control message in transit is non-zero.

Hence,

$$\begin{aligned} & W_c = 1 \\ & \implies \textstyle\sum_{W \in (A \cup B \cup C)} W = 0 \text{ (by } I_1) \\ & \implies (A \cup B \cup C) = \phi \text{ (by } I_2) \\ & \implies (A \cup B) = \phi. \end{aligned}$$

Note that $(A \cup B) = \phi$ implies that the computation has terminated. Therefore, the algorithm never detects a false termination.

Further,

$$\begin{aligned} & (A \cup B) = \phi \\ & \implies W_c + \textstyle\sum_{W \in C} W = 1 \text{ (by } I_1). \end{aligned}$$

Since the message delay is finite, after the computation has terminated, eventually $W_c = 1$. Thus, the algorithm detects a termination in finite time.

7.5 A spanning-tree-based termination detection algorithm

The algorithm assumes there are N processes P_i, $0 \le i \le N$, which are modeled as the nodes i, $0 \le i \le N$, of a fixed connected undirected graph. The edges of the graph represent the communication channels, through which a process sends messages to neighboring processes in the graph. The algorithm uses a fixed spanning tree of the graph with process P_0 at its root which is responsible for termination detection. Process P_0 communicates with other processes to determine their states and the messages used for this purpose are called signals. All leaf nodes report to their parents, if they have terminated. A parent node will similarly report to its parent when it has completed processing and all of its immediate children have terminated, and so on. The root concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated.

The termination detection algorithm generates two waves of signals moving inward and outward through the spanning tree. Initially, a contracting wave of signals, called *tokens*, moves inward from leaves to the root. If this token wave reaches the root without discovering that termination has occurred, the root initiates a second outward wave of *repeat* signals. As this repeat wave reaches leaves, the token wave gradually forms and starts moving inward again. This sequence of events is repeated until the termination is detected.

7.5.1 Definitions

1. Tokens: a contracting wave of signals that move inward from the leaves to the root.
2. Repeat signal: if a token wave fails to detect termination, node $P0$ initiates another round of termination detection by sending a signal called Repeat, to the leaves.
3. The nodes which have one or more tokens at any instant form a set S.
4. A node j is said to be outside of set S if j does not belong to S and the path (in the tree) from the root to j contains an element of S. Every path from the root to a leaf may not contain a node of S.
5. Note that all nodes outside S are idle. This is because, any node that terminates, transmits a token to its parent. When a node transmits the token, it goes out of the set S.

We first give a simple algorithm for termination detection and discuss a problem associated with it. Then we provide the correct algorithm.

7.5.2 A simple algorithm

Initially, each leaf process is given a token. Each leaf process, after it has terminated, sends its token to its parent. When a parent process terminates and after it has received a token from each of its children, it sends a token to its parent. This way, each process indicates to its parent process that the subtree below it has become idle. In a similar manner, the tokens get propagated to the root. The root of the tree concludes that termination has occurred, after it has become idle and has received a token from each of its children.

A problem with the algorithm

This simple algorithm fails under some circumstances. After a process has sent its token to its parent, it should remain idle. However, this is not the case. The problem arises when a process after it has sent a token to its parent, receives a message from some other process. Note that this message could cause the process (that has already sent a token to its parent) to again become active. Hence the simple algorithm fails since the process that indicated to its parent that it has become idle, is now active because of the message it

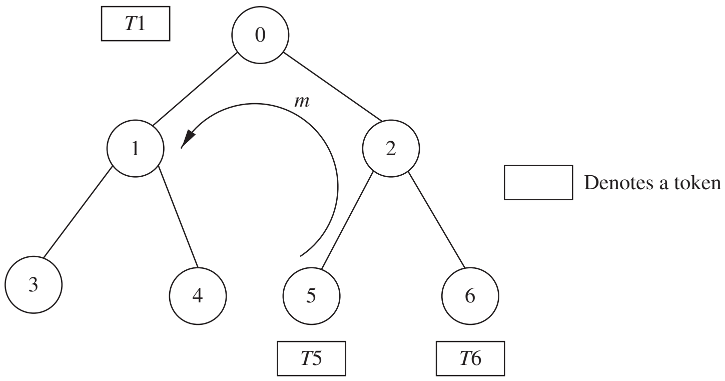

Figure 7.1 An example of the problem.

received from an active process. Hence, the root node just because it received a token from a child, can't conclude that all processes in the child's subtree have terminated. The algorithm has to be reworked to accommodate such message-passing scenarios.

The problem is explained with the example shown in Figure 7.1. Assume that process 1 has sent its token ($T1$) to its parent, namely, process 0. On receiving the token, process 0 concludes that process 1 and its children have terminated. Process 0 if it is idle, can conclude that termination has occurred, whenever it receives a token from process 2. But now assume that just before process 5 terminates, it sends a message m to process 1. On the reception of this message, process 1 becomes active again. Thus, the information that process 0 has about process 1 (that it is idle) becomes void. Therefore, this simple algorithm does not work.

7.5.3 The correct algorithm

We now present the correct algorithm that was developed by Topor [19] and it works even when messages such as the one if Figure 7.1 are present. The main idea is to color the processes and tokens and change the color when such messages are involved.

The basic idea

In order to enable the root node to know that a node in its children's subtree, that was assumed to be terminated, has become active due to a message, a coloring scheme for tokens and nodes is used. The root can determine that an idle process has been activated by a message, based on the color of the token it receives from its children. All tokens are initialized to white. If a process had sent a message to some other process, it sends a black token to its parent on termination; otherwise, it sends a white token on termination. Hence, the parent process on getting the black token knows that its child had sent a message to some other process. The parent, when sending its token (on terminating) to its parent, sends a black token only if it received a black token from one of its children. This way, the parent's parent knows that one of the processes in its child's subtree had sent a message to some other process. This gets propagated and finally the root node knows that message-passing was involved when it receives a black token from one of its children. In this case, the root asks all nodes in the system to restart the termination detection. For this, the root sends a repeat signal to all other process. After receiving the repeat signal, all leaves will restart the termination detection algorithm.

The algorithm description

The algorithm works as follows:

1. Initially, each leaf process is provided with a token. The set S is used for book-keeping to know which processes have the token. Hence S will be the set of all leaves in the tree.
2. Initially, all processes and tokens are white. As explained above, coloring helps the root know if a message-passing was involved in one of the subtrees.
3. When a leaf node terminates, it sends the token it holds to its parent process.
4. A parent process will collect the token sent by each of its children. After it has received a token from all of its children and after it has terminated, the parent process sends a token to its parent.
5. A process turns black when it sends a message to some other process. This coloring scheme helps a process remember that it has sent a message. When a process terminates, if its is black, it sends a black token to its parent.
6. A black process turns back to white after it has sent a black token to its parent.
7. A parent process holding a black token (from one of its children), sends only a black token to its parent, to indicate that a message-passing was involved in its subtree.
8. Tokens are propagated to the root in this fashion. The root, upon receiving a black token, will know that a process in the tree had sent a message to some other process. Hence, it restarts the algorithm by sending a Repeat signal to all its children.
9. Each child of the root propagates the Repeat signal to each of its children and so on, until the signal reaches the leaves.
10. The leaf nodes restart the algorithm on receiving the Repeat signal.
11. The root concludes that termination has occurred, if:
    (a) it is white;
    (b) it is idle; and
    (c) it has received a white token from each of its children.

7.5.4 An example

We now present an example to illustrate the working of the algorithm.

1. Initially, all nodes 0 to 6 are white (Figure 7.2). Leaf nodes 3, 4, 5, and 6 are each given a token. Node 3 has token $T3$, node 4 has token T4, node 5 has token $T5$, and node 6 has token $T6$. Hence, S is {3, 4, 5, 6}.
2. When node 3 terminates, it transmits $T3$ to node 1. Now S changes to 1, 4, 5, 6. When node 4 terminates, it transmits $T4$ to node 1 (Figure 7.3). Hence, S changes to {1, 5, 6}.
3. Node 1 has received a token from each of its children and, when it terminates, it transmits a token $T1$ to its parent (Figure 7.4). S changes to {0, 5, 6}.
4. After this, suppose node 5 sends a message to node 1, causing node 1 to again become active (Figure 7.5). Since node 5 had already sent a token to its parent node 0 (thereby making node 0 assume that node 5 had terminated), the new message makes the system inconsistent as far as termination detection is concerned. To deal with this, the algorithm executes the following steps.
5. Node 5 is colored black, since it sent a message to node 1.

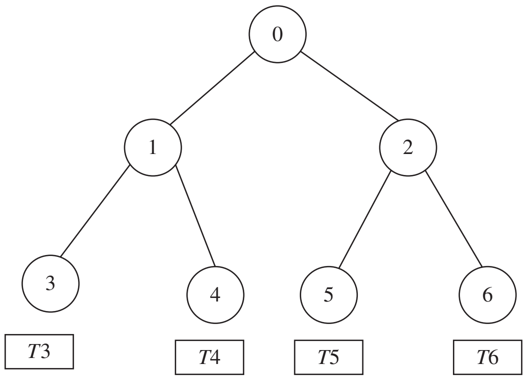

Figure 7.2 All leaf nodes have tokens. S = {3, 4, 5, 6}.

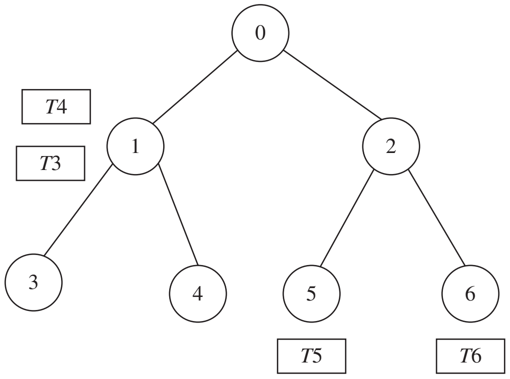

Figure 7.3 Nodes 3 and 4 become idle. S = {1, 5, 6}.

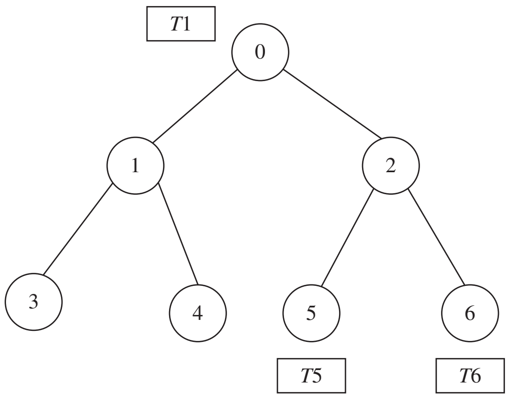

Figure 7.4 Node 1 becomes idle. $S = \{0, 5, 6\}$.

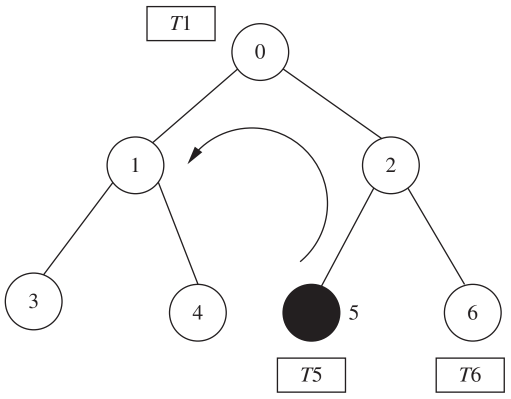

Figure 7.5 Node 5 sends a message to node 1.

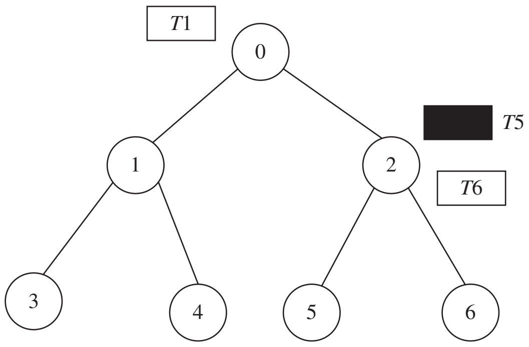

Figure 7.6 Nodes 5 and 6 become idle. $S = \{0, 2\}$.

6. When node 5 terminates, it sends a black token $T5$ to node 2. So, S changes to $\{0, 2, 6\}$. After node 5 sends its token, it turns white (Figure 7.6). When node 6 terminates, it sends the white token $T6$ to node 2. Hence, S changes to $\{0, 2\}$.
7. When node 2 terminates, it sends a black token $T2$ to node 0, since it holds a black token $T5$ from node 5 (Figure 7.7).

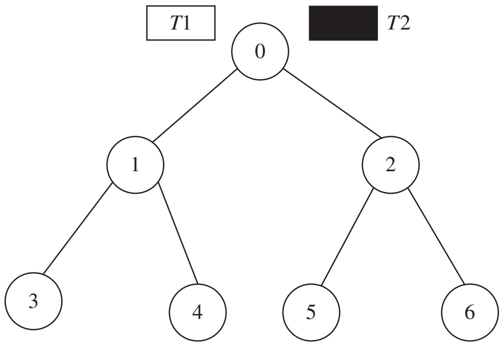

Figure 7.7 Node 2 becomes idle. $S = \{0\}$. Node 0 initiates a repeat signal.

8. Since node 0 has received a black token $T2$ from node 2, it knows that there was a message sent by one or more of its children in the tree and hence sends a repeat signal to each of its children.
9. The repeat signal is propagated to the leaf nodes and the algorithm is repeated. Node 0 concludes that termination has occurred if it is white, it is idle, and it has received a white token from each of its children.

7.5.5 Performance

The best case message complexity of the algorithm is $O(N)$, where N is the number of processes in the computation. The best case occurs when all nodes send all computation messages in the first round. Therefore, the algorithm executes only twice and the message complexity depends only on the number of nodes.

However, the worst case complexity of the algorithm is $O(N^*M)$, where M is the number of computation messages exchanged. The worst case occurs when only computation message is exchanged every time the algorithm is executed. This causes the root to restart termination detection as many times as there are no computation messages. Hence, the worst case complexity is $O(N^*M)$.

7.6 Message-optimal termination detection

Now we discuss a message optimal termination detection algorithm by Chandrasekaran and Venkatesan [2]. The network is represented by a graph $G = (V, E)$, where V is the set of nodes, and $E \subseteq V \times V$ is the set of edges or communication links. The communication links are bidirectional and exhibit FIFO property. The processors and communication links incur arbitrary but finite delays in executing their functions. The algorithm assumes the existence of a leader and a spanning tree in the network. If a leader is not available, the minimum spanning tree algorithm of Gallager *et al.* [7] can be used to elect a leader and find a spanning tree using $O(\mid E \mid + \mid V \mid \log \mid V \mid)$ messages.

7.6.1 The main idea

Let us reconsider the method for termination detection disussed in the previous section the root of the tree initiates one phase of termination detection by turning white. An interior node, on receiving a white token from its parent, turns white and transmits a white token to all of its children. Eventually each leaf receives a white token and turns white. When a leaf node becomes idle, it transmits a token to its parent and the token has the same color as that of the leaf node. An interior node waits for a token from each of its children. It also waits until it becomes idle. It then sends a white token to its parent if its color is white and it received a white token from each of its children. Finally, the root node infers the termination of the underlying computation if it receives a white token from each child, its color is white, and it is idle.

This simple algorithm is inefficient in terms of message complexity due to the following reasons. Consider the scenario shown in Figure 7.8, where node *p* sends a message *m* to node *q*. Before node *q* received the message *m*, it had sent a white token to its parent (because it was idle and it had received a white token from each of its children). In this situation, node *p* cannot send a white token to its parent until node *q* becomes idle. To insure this, in Topor's algorithm, node *p* changes its color to black and sends a black token to its parent so that termination detection is performed once again. Thus, every message of the underlying computation can potentially cause the execution of one more round of the termination detection algorithm, resulting in significant message traffic.

The main idea behind the message-optimal algorithm is as follows: when a node *p* sends a message *m* to node *q*, *p* should wait until *q* becomes idle and only after that, *p* should send a white token to its parent. This rule ensures that if an idle node *q* is restarted by a message *m* from from a node *p*, then the sender *p* waits till *q* terminates before *p* can send a white token to its parent. To achieve this, when node *q* terminates, it sends an acknowledgement (a control message) to node *p* informing node *p* that the set of actions triggered

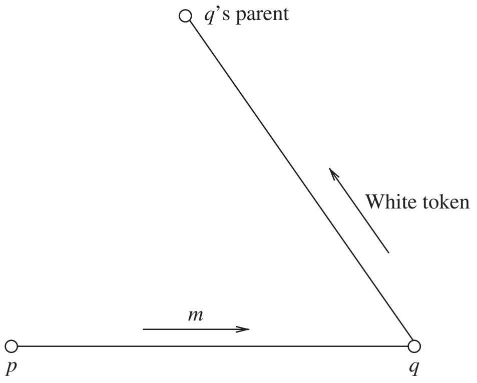

Figure 7.8 Node *p* sends a message *m* to node *q* that has already sent a white token to its parent [2].

by message *m* has been completed and that node *p* can send a white token to its parent. However, note that node *q*, after being woken up by message *m* from node *p*, may wake up another idle node *r*, which in turn may wake up other nodes. Therefore, node *q* should not send an acknowledgement to *p* until it receives acknowledgement messages for all of the messages it sent after it received message *m* from node *p*. This restriction also applies to node *r* and other nodes. Clearly, both the sender and the receiver keep track of each message, and a node will send a white token to its parent only after it has received an acknowledgement for every message it has sent and has received a white token from each of its children.

7.6.2 Formal description of the algorithm

Initially, all nodes in the network are in state NDT (not detecting termination) and all links are uncolored. For termination detection, the root node changes its state to DT (detecting termination) and sends a warning message on each of its outgoing edges. When a node *p* receives a warning message from its neighbor, say *q*, it colors[2] the incoming link (*q*, *p*) and if it is in state NTD, it changes its state to DT, colors each of its outgoing edges, and sends a warning message on each of its outgoing edges.

When a node *p* in state DT sends a basic message to its neighbor *q*, it keeps track of this information by pushing the entry *TO*(*q*) on its local stack.

When a node *x* receives a basic message from node *y* on the link (*y*, *x*) that is colored by *x*, node *x* knows that the sender node *y* will need an acknowledgement for this message from it. The receiver node *x* keeps track of this information by pushing the entry *FROM*(*y*) on its local stack. Procedure receive_message is given in Algorithm 7.1.

```
Procedure receive_message(y: neighbor);
(* performed when a node x receives a message from its neighbor y on the link
(y,x) that was colored by x *)
            begin
                receive message from y on the link (y,x)
                if (link (y,x) has been colored by x) then
                        push FROM(y) on the stack
            end;
```

Algorithm 7.1 Procedure *receive_message*.

[2] All links are uncolored or colored. The shade of the color does not matter.

Eventually, every node in the network will be in the state DT as the network is connected. Note that both sender and receiver keep track of every message in the system.

When a node p becomes idle, it calls procedure *stack_cleanup*, which is defined in Algorithm 7.2. Procedure *stack_cleanup* examines its stack from the top and, for every entry of the form $FROM(q)$, deletes the entry and sends the *remove_entry* message to node q. Node p repeats this until it encounters an entry of the form $TO(x)$ on the stack. The idea behind this step is to inform those nodes that sent a message to p that the actions triggered by their messages to p are complete.

```
Procedure stack_cleanup;
begin
        while (top entry on stack is not of the form "TO()") do
            begin
                pop the entry on the top of the stack;
                let the entry be FROM(q);
                send a remove_entry message to q
            end
  end;
```

Algorithm 7.2 Procedure *stack_cleanup*.

When a node x receives a *remove_entry* message from its neighbor y, node x infers that the operations triggered by its last message to y have been completed and hence it no longer needs to keep track of this information. Node x on receipt of the control message *remove_entry* from node y, examines its stack from the top and deletes the first entry of the form $TO(y)$ from the stack. If node x is idle, it also performs the *stack_cleanup* operation. The procedure *receive_remove_entry* is defined in Algorithm 7.3.

```
Procedure receive_remove_entry(y: neighbor);
(* performed when a node x receives a remove_entry message from its
neighbor y *)
  begin
          scan the stack and delete the first entry of the form TO(y);
          if idle then
                  stack_cleanup
  end;
```

Algorithm 7.3 Procedure *receive_remove_entry*.

A node sends a terminate message to its parent when it satisfies all the following conditions:

1. It is idle.
2. Each of its incoming links is colored (it has received a warning message on each of its incoming links).
3. Its stack is empty.
4. It has received a *terminate* message from each of its children (this rule does not apply to leaf nodes).

When the root node satisfies all of the above conditions, it concludes that the underlying computation has terminated.

7.6.3 Performance

We analyze the number of control messages used by the algorithm in the worst case. Each node in the network sends one warning message on each outgoing link. Thus, each link carries two warning messages, one in each direction. Since there are $\mid E \mid$ links, the total number of warning messages generated by the algorithm is $2^{*}\mid E \mid$. For every message generated by the underlying computation (after the start of the termination detection algorithm), exactly one *remove_message* is sent on the network. If M is the number of messages sent by the underlying computation, then at most M *remove_entry* messages are used. Finally, each node sends exactly one *terminate* message to its parent (on the tree edge) and since there are only $\mid V \mid$ nodes and $\mid V \mid -1$ tree edges, only $\mid V \mid - 1$ *terminate* messages are sent. Hence, the total number of messages generated by the algorithm is $2^{*} \mid E \mid + \mid V \mid -1 + M$. Thus, the message complexity of the algorithm is $O(\mid E \mid + M)$ as $\mid E \mid > \mid V \mid -1$ for any connected network. The algorithm is asymptotically optimal in the number of messages.

7.7 Termination detection in a very general distributed computing model

So far we assumed that the reception of a single message is enough to activate a passive process. Now we consider a general model of distributed computing where a passive process does not necessarily become active on the receipt of a message [1]. Instead, the condition of activation of a passive process is more general and a passive process requires a set of messages to become active. This requirement is expressed by an *activation condition* defined over the set DS_i of processes from which a passive process P_i is expecting messages. The set DS_i associated with a passive process P_i is called the *dependent set* of P_i. A passive process becomes active only when its activation condition is fulfilled.

7.7.1 Model definition and assumptions

The distributed computation consists of a finite set P of processes P_i, $i = 1, \ldots, n$, interconnected by unidirectional communication channels. Communication channels are reliable, but they do not obey FIFO property. Message transfer delay is finite but unpredictable.

A passive process that has terminated its computation by executing for example an end or stop statement is said to be individually terminated; its dependent set is empty and therefore, it can never be activated.

AND, OR, and AND-OR models

There are several request models, such as AND, OR, AND-OR models. In the AND model, a passive process P_i can be activated only after a message from every process belonging to DS_i has arrived. In the OR model, a passive process P_i can be activated when a message from any process belonging to DS_i has arrived. In the AND-OR model, the requirement of a passive process P_i is defined by a set R_i of sets DS_i^1, $DS_i^2, \ldots, DS_i^{q_i}$, such that for all r, $1 \leq r \leq q_i$, $DS_i^r \subseteq P$. The dependent set of P_i is $DS_i = DS_i^1 \cup DS_i^2 \cup \ldots DS_i^{q_i}$. Process P_i waits for messages from all processes belonging to DS_i^1 or for messages from all processes belonging to DS_i^2 or for messages from all processes belonging to $DS_i^{q_i}$.

The *k* out of *n* model

In the k out of n model, the requirement of a passive process P_i is defined by the set DS_i and an integer k_i, $1 \leq k_i \leq |DS_i| = n_i$ and process P_i becomes active when it has received messages from k_i distinct processes in DS_i. Note that a more general k out of n model can be constructed as disjunctions of several k out of n requests.

Predicate fulfilled

To abstract the activation condition of a passive process P_i, a predicate $fulfilled_i(A)$ is introduced, where A is a subset of P. Predicate $fulfilled_i(A)$ is true if and only if messages arrived (and not yet consumed) from all processes belonging to set A are sufficient to activate process P_i.

7.7.2 Notation

The following notation will be used to define the termination of a distributed computation:

- $passive_i$: true iff P_i is passive.
- $empty(j, i)$: true iff all messages sent by P_j to P_i have arrived at P_i; the messages not yet consumed by P_i are in its local buffer.
- $arr_i(j)$: true iff a message from P_j to P_i has arrived at P_i and has not yet been consumed by P_i.

- ARR_i = {processes P_j such that $arr_i(j)$}.
- NE_i = {processes P_j such that $\neg$ $empty(j, i)$}.

7.7.3 Termination definitions

Two different types of terminations are defined, dynamic termination and static termination:

- **Dynamic termination** The set of processes P is said to be dynamically terminated at some instant if and only if the predicate *Dterm* is true at that moment where:

  $$Dterm \equiv \forall P_i \in P : passive_i \wedge \neg fulfilled_i(ARR_i \cup NE_i).$$

  Dynamic termination means that no more activity is possible from processes, though messages of the underlying computation can still be in transit. This definition is useful in "early" detection of termination as it allows us to conclude whether a computation has terminated even if some of its messages have not yet arrived.

  Note that dynamic termination is a stable property because once *Dterm* is true, it remains true.
- **Static termination** The set of processes P is said to be statically terminated at some instant if and only if the predicate *Sterm* is true at that moment where:

  $$Sterm \equiv \forall P_i \in P : passive_i \wedge (NE_i = \emptyset) \wedge \neg fulfilled_i(ARR_i).$$

  Static termination means all channels are empty and none of the processes can be activated. Thus, static termination is focused on the state of both channels and processes. When compared to *Dterm*, the predicate *Sterm* corresponds to "late" detection as, additionally, all channels must be empty.

7.7.4 A static termination detection algorithm

Informal description

A control process C_i, called a *controller*, is associated with each application process P_i. Its role is to observe the behavior of process P_i and to cooperate with other controllers C_j to detect occurrence of the predicate *Sterm*. In order to detect static termination, a controller, say C_a, initiates detection by sending a control message *query* to all controllers (including itself). A controller C_i responds with a message *reply*(ld_i), where ld_i is a Boolean value. C_a combines all the Boolean values received in *reply* messages to compute $td := \bigwedge_{1 \le i \le n} ld_i$. If td is true, C_a concludes that termination has occurred. Otherwise, it sends new *query* messages. The basic sequence of sending of *query* messages followed by the reception of associated *reply* messages is called a *wave*.

The core of the algorithm is the way a controller C_i computes the value ld_i sent back in a *reply* message. To ensure safety, the values $ld_1, \ldots ld_n$ must be such that:

$$\bigwedge_{1\le i\le n} ld_i \Longrightarrow Sterm$$

$$\Longrightarrow \forall P_i \in P : passive_i \wedge (NE_i = \emptyset) \wedge \neg fulfilled_i(ARR_i).$$

A controller C_i delays a response to a *query* as long as the following locally evaluable predicate is false: $passive_i \wedge (notack_i = 0) \wedge \neg fulfilled_i(ARR_i)$. When this predicate is false, the static termination cannot be guaranteed.

For correctness, the values reported by a wave must not miss the activity of processes "in the back" of the wave. This is achieved in the following manner: each controller C_i maintains a Boolean variable cp_i (initialized to true iff P_i is initially passive) in the following way:

- When P_i becomes active, cp_i is set to false.
- When C_i sends a reply message to C_a, it sends the current value of cp_i with this message, and then sets cp_i to true.

Thus, if a reply message carries value true from C_i to C_a, it means that P_i has been continuously passive since the previous wave, and the messages arrived and not yet consumed are not sufficient to activate P_i, and all output channels of P_i are empty.

Formal description

The algorithm for static termination detection is as follows. By a *message*, we mean any message of the underlying computation; *queries* and *replies* are called *control* messages.

S1: When P_i sends a message to P_j

$$notack_i := notack_i + 1$$

S2: When a message from P_j arrives to P_i

send *ack* to C_j

S3: When C_i receives *ack* from C_j

$$notack_i = notack_i - 1$$

S4: When P_i becomes active

$$cp_i := false.$$

(* A passive process can only become active when its activation condition is true; this activation is under the control of the underlying operating system, and the termination detection algorithm only observes it. *)

```
S5: When $C_i$ receives query from $C_\alpha$
  (* Executed only by $C_\alpha$ *)

          Wait until
          $((passive_i \wedge (notack_i = \emptyset) \neg fulfilled_i(ARR_i))$;
          $ld_i := cp_i$;
          $cp_i := true$;
          send $reply(ld_i)$ to $C_\alpha$

S6: When controller $C_a$ decides to detect static termination

              repeat send $query$ to all $C_i$;
                  receive $reply(ld_i)$ from all $C_i$;
                  td := $\bigwedge_{1 \le i \le n} ld_i$;
              $until$ td;
              $claim\ static\ termination$
```

Performance

The efficiency of this algorithm depends on the implementation of waves. Two waves are in general necessary to detect static termination. A wave needs two types of messages: n queries and n replies, each carrying one bit. Thus, $4n$ control messages of two distinct types carrying at most one bit each are used to detect the termination once it has occurred. If waves are supported by a ring, this complexity reduces to $2n$. The detection delay is equal to duration of two sequential wave executions.

7.7.5 A dynamic termination detection algorithm

Recall that a dynamic termination can occur before all messages of the computation have arrived. Thus, termination of the computation can be detected sooner than in static termination.

Informal description

Let C_α denote the controller that launches the waves. In addition to cp_i, each controller C_i has the following two vector variables, denoted as s_i and r_i, that count messages, respectively, sent to and received from every other process:

- $s_i[j]$ denotes the number of messages sent by P_i to P_j;
- $r_i[j]$ denotes the number of messages received by P_i from P_j.

Let S denote an $n \times n$ matrix of counters used by C_α; entry $S[i, j]$ represents C_α's knowledge about the number of messages sent by P_i to P_j.

First, C_a sends to each C_i a query message containing the vector $(S[1,i], \ldots, S[n,i])$, denoted by $S[.,i]$. Upon receiving this query message, C_i computes the set ANE_i of its non-empty channels. This is an approximate knowledge but is sufficient to ensure correctness. Then C_i computes ld_i, which is true if and only if P_i has been continuously passive since the previous wave and its requirement cannot be fulfilled by all the messages arrived and not yet consumed (ARR_i) and all messages potentially in its input channels (ANE_i). C_i sends to C_α a reply message carrying the values ld_i and vector s_i. Vector s_i is used by C_α to update row $S[i,]$ and thus gain more accurate knowledge. If $\bigwedge_{1 \le i \le n} ld_i$ evaluates to true, C_a claims dynamic termination of the underlying computation. Otherwise, C_α launches a new wave by sending *query* messages.

Vector variables s_i and r_i allow C_α to update its (approximate) global knowledge about messages sent by each P_i to each P_j and get an approximate knowledge of the set of non-empty input channels.

Formal description

All controllers C_i execute statements S1 to S4. Only the initiator C_α executes S5. Local variables s_i, r_i, and S are initialized to 0.

```
S1: When $P_i$ sends a message to $P_j$
            $s_i[j] := s_i[j] + 1$

S2: When a message from $P_j$ arrives at $P_i$
            $r_i[j] := r_i[j] + 1$

S3: When $P_i$ becomes active
            $cp_i := false$

S4: When $C_i$ receives $query(VC[1...n])$ from $C_\alpha$
        (*$VC[1...n] = S[1...n, i]$ is the $i$th column of $S$*)
        $ANE_i := \{P_j : VC[j] > r_i[j]\}$;
        $ld_i := cp_i \wedge \neg fulfilled_i(ARR_i \cup NE_i)$;
        $cp_i := (state_i = \text{passive})$;
        send $reply(ld_i, s_i)$ to $C_\alpha$
```

```
S5: When controller $C_\alpha$ decides to detect dynamic termination
    repeat for each $C_i$
            send $query(S[1 \ldots n, i])$ to $C_i$;
            (* the $i$th column of$S$ is sent to $C_i$ *)
            receive $reply(ld_i, s_i)$ from all $C_i$;
            $\forall i \in [1..n] : S[i, .] := s_i$;
            td := $\bigwedge_{1 \le i \le n} ld_i$
    until td;
    claim dynamic termination
```

Performance

The dynamic termination detection algorithm needs two waves after dynamic termination has occurred to detect it. Thus, its message complexity is $4n$, which is lower than the static termination detection algorithm since no acknowledgements are necessary. However, messages are composed of n monotonically increasing counters. As waves are sequential, *query* (and *reply*) messages between C_α and each C_i are received and processed in their sending order; this FIFO property can be used in conjunction with Singhal–Kshemkalyani's differential technique to decrease the size of the control messages. The detection delay is two waves but is shorter than the delay of the static termination algorithm as acknowledgements are not used.

7.8 Termination detection in the atomic computation model

Mattern [12] developed several algorithm for termination detection in the atomic computation model.

Assumptions

1. Processes communicate solely by messages. Messages are received correctly after an arbitrary but finite delay. Messages sent over the same communication channel may not obey the FIFO rule.
2. A *time cut* is a line crossing all process lines. A time line can be a straight vertical line or a zigzag line, crossing all process lines. The time cut of a distributed computation is a set of actions characterized by a fact that whenever an action of a process belongs to that set, all previous actions of the same process also belong to the set.
3. We assume that all atomic actions are totally globally ordered i.e., no two actions occur at the same time instant.

7.8.1 The atomic model of execution

In the *atomic model* of the distributed computation, a process may at any time take any message from one of its incoming communication channels, immediately change its internal state, and at the same instant send out zero or more messages. All local actions at a process are performed in zero time. Thus, consideration of process states is eliminated when performing termination detection.

In the atomic model, a distributed computation has terminated at time instant t if at this instant all communications channels are empty. This is because execution of an internal action at a process is instantaneous.

A dedicated process, P_1, the initiator, determines if the distributed computation has terminated. The initiator P_1 starts termination detection by sending control messages directly or indirectly to all other processes. Let us assume that processes $P_1, \ldots, P_n$ are ordered in sequence of the arrival of the control message.

7.8.2 A naive counting method

To find out if there are any messages in transit, an obvious solution is to let every process *count* the number of basic messages sent and received. We denote the total number of basic messages P_i has sent at (global) time instant t by $s_i(t)$, and the number of messages received by $r_i(t)$. The values of the two local counters are communicated to the initiator upon request. Having directly or indirectly received these values from all processes, the initiator can accumulate the counters. Figure 7.9 shows an example, where the time instants at which the processes receive the control messages and communicate the values of their counters to the initiator are symbolized by striped dots. These are connected by a line representing a "control wave," which induces a time cut.

If the accumulated values at the initiator indicate that the sum of all the messages received by all processes is the same as the sum of all messages

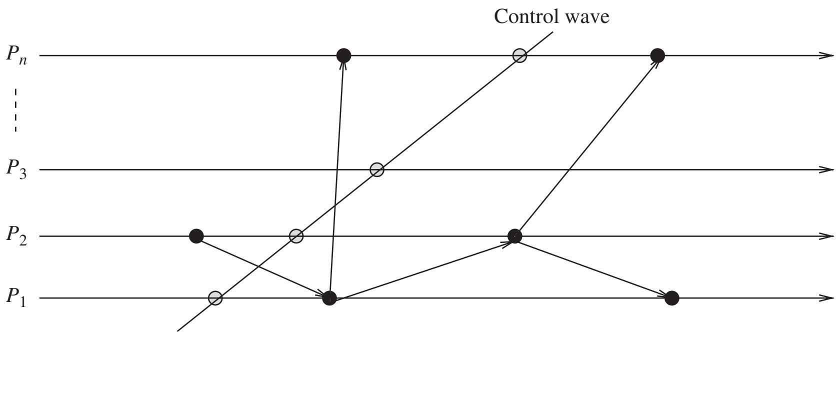

Figure 7.9 An example showing a control wave with a backward communication [12].

sent by all processes, it may give an impression that all the messages sent have been received, i.e., there is no message in transit.

Unfortunately because of the time delay of the control wave, this simple method is not correct. The example in Figure 7.9 shows that the counters can become corrupted by messages "from the future," crossing from the right side of the control wave to its left.

The accumulated result indicates that one message was sent and one received although the computation has not terminated. This misleading result is caused by the fact that the time cut is inconsistent. A time cut is considered to be inconsistent, if when the diagonal line representing it is made vertical, by compressing or expanding the local time scales, a message crosses the control wave backwards.

However, this naive method for termination detection works if the time cut representing the control wave is consistent.

Various strategies can be applied to correct the deficiencies of the naive counting method:

- If the time cut is inconsistent, restart the algorithm later.
- Design techniques that will only provide consistent time cuts.
- Do not lump the count of all messages sent and all messages received. Instead, relate the messages sent and received between pairs of processes.
- Use techniques like freezing the underlying computation.

7.8.3 The four counter method

A very simple solution consists of counting twice using the naive counting method and comparing the results. After the initiator has received the response from the last process and accumulated the values of the counters R^* and S^* (where $R^* := \sum_{\forall i} r_i(t_i)$ and $S^* := \sum_{\forall i} s_i(t_i)$), it starts a second control wave (see Figure 7.10), resulting in values R'^* and S'^*. The system is terminated if values of the four counters are equal, i.e., $R^* = S^* = R'^* = S'^*$. In fact, a slightly stronger result exists: if $R^* = S'^*$, then the system terminated at the end of the first wave (t_2 in Figure 7.10).

Let t_2 denote the time instant at which the first wave is finished, and t_3 ($\geq t_2$) denote the starting time of the second wave (see Figure 7.10).

1. Local message counters are monotonic, that is, $t \leq t'$ implies $s_i(t) \leq s_i(t')$ and $r_i(t) \leq r_i(t')$. This follows from the definition.
2. The total number of messages sent or received is monotonic, that is, $t \leq t'$ implies $S(t) \leq S(t')$ and $R(t) \leq R(t')$.
3. $R^* \leq R(t_2)$. This follows from (1) and the fact that all values r_i are collected before t_2.
4. $S'^* \geq S(t_3)$. This follows from (1) and the fact that all values s_i are collected after t_3.
5. For all t, $R(t) \leq S(t)$. This is because the number of messages in transit $D(t) := S(t) - R(t) \geq 0$.

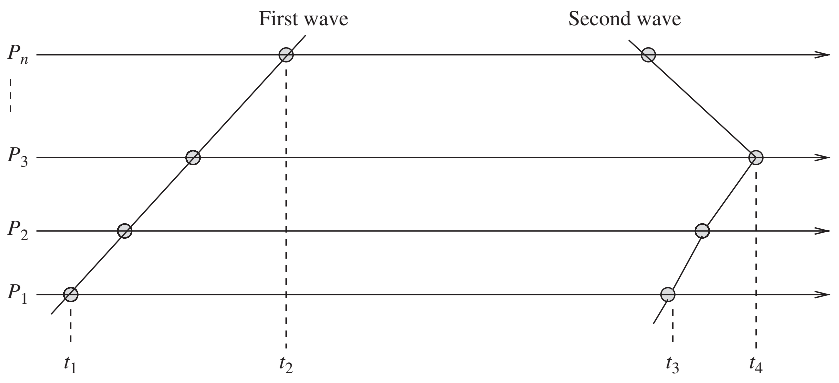

Figure 7.10 An example showing two control waves [12].

Now we show that if $R^* = S'^*$, then the computation had terminated at the end of the first wave:

$$\begin{aligned} R^* = S'^* &\Longrightarrow R(t_2) \geq S(t_3) \\ &\Longrightarrow R(t_2) \geq S(t_2) \\ &\Longrightarrow R(t_2) = S(t_2) \end{aligned}$$

That is, the computation terminated at t_2 (at the end of the first wave).

If the system terminated before the start of the first wave, it is trivial that all messages arrived before the start of the first wave, and hence the values of the accumulated counters will be identical. Therefore, termination is detected by the algorithm in two "rounds" after it had occurred. Note that the second wave of an unsuccessful termination test can be used as the first wave of the next termination test. However, a problem with this method is to decide when to start the next wave after an unsuccessful test – there is a danger of an unbounded control loop.

7.8.4 The sceptic algorithm

Note that the values of the counters obtained by the first wave of the four counter method can become corrupted if there is some activity at the right of the wave. To detect such activity, we use *flags* which are initialized by the first wave, and set by the processes when they receive (or alternatively when they send) messages. The second wave checks if any of the flags have been set, in which case a possible corruption is indicated. A general drawback is that at least two waves are necessary to detect the termination.

It is possible to devise several variants based on the *logical control topology*. If the initiator asks every process individually, it corresponds to a star topology. It is possible to implement the sceptic algorithm on a ring; however, symmetry is not easily achieved since different waves may interfere when a single flag is used at each process. A spanning tree is also an interesting control configuration. Echo algorithms used as a parallel graph traversal method induce two phases. The “down” phase is characterized by the receipt of a first control message which is propagated to all other neighbors, and the “up” phase by the receipt of the last of the echoes from its neighboring nodes. These two phases can be used as two necessary waves of the sceptic method for termination detection.

7.8.5 The time algorithm

The time algorithm is a single wave detection algorithm where termination can be detected in one single wave after its occurrence at the expense of increased amount of control information or augmenting every message with a timestamp. In the time algorithm, each process has a *local clock* represented by a counter initialized to 0.

A control wave started by the initiator at time i, accumulates the values of the counters and “synchronizes” the local clocks by setting them to $i+1$. Thus, the control wave separates “past” from “future.” If a process receives a message whose *timestamp* is greater than its own local time, the process has received a message from the future (i.e., the message crossed the wave from right to left) and the message has corrupted the counters. After such a message has been received, the current control wave is nullified on arrival at the process.

Formal description

Every process P_j $(1 \leq j \leq n)$ has a local message counter *COUNT* (initialized to 0) that holds the value $s_j - r_j$, a local discrete *CLOCK* (initialized to 0), and a variable *TMAX* (also initialized to 0) that holds the latest send time of all messages received by P_j.

The psuedo code for process P_j is shown in Algorithm 7.4.

A control message consists of four parameters: the (local) time at which the control round was started, the accumulator for the message counters, a flag which is set when a process has received a basic message from the future ($TMAX \geq TIME$), and the identification of the initiating process. The first component of a basic message is always the timestamp.

For each single control wave, any basic message that crosses the wave from the right side of its induced cut to its left side is detected. Note that different control waves do not interfere; they merely advance the local clocks further. Once the system is terminated, the values of the *TMAX* variables remain fixed and since for every process P_j, $TMAX_j \leq max\ CLOCK_i$ $(1 \leq i \leq n)$, the process with the maximum clock value can detect global termination in one round. Other processes may need more rounds.

```
(a) When sending a basic message to $P_i$:
(1)        $COUNT \leftarrow COUNT + 1$;
(2)        send <$CLOCK$,...> to $P_i$;
        /* timestamped basic message */

(b) When receiving a basic message <$TSTAMP$,...>:
(3)        $COUNT \leftarrow COUNT - 1$;
(4)        $TMAX \leftarrow max(TSTAMP, TMAX)$;
(5)        /* process the message */

(c) When receiving a control message <$TIME$, $ACCU$, $INVALID$, $INIT$>:
(6)        $CLOCK \leftarrow max(TIME, CLOCK)$: /* synchronize the local closk */
(7)        if $INIT = j$ /* complete round? */
(8)            then if $ACCU = 0$ and not $INVALID$
(9)                then "terminated" else "try again";
(10)               endif;
(11)           else send <$TIME$, $ACCU + COUNT$, $INVALID$ or
                   $TMAX \geq TIME$, $INIT$> to $P_{(j \bmod n)+1}$;
(12)       end_if;

(d) When starting a control round:
(13)        $CLOCK \leftarrow CLOCK + 1$;
(14)        send <$CLOCK$, $COUNT$, $false$, $j$> to $P_{(j \bmod n)+1}$;
```

Algorithm 7.4 The time algorithm [12].

7.8.6 Vector counters method

Vector counters method of termination detection consists of counting messages in such a way that it is not possible to mislead the accumulated counters.

The configuration used is the ring with n processes where every process P_j ($1 \leq j \leq n$) has a $COUNT$ vector of length n, where $COUNT[i]$ ($1 \leq i \leq n$) denotes the ith component of the vector. A circulating control message also consists of a vector of length n. For each process P_j, the local variable $COUNT[i]$ ($i \neq j$) holds the number of basic messages that have been sent to process P_i since the last visit of the control message. Likewise, the negative value of $COUNT[j]$ indicates how many messages have been received from any other process. At any (global) time instant, the sum of the kth components of all n $COUNT$ vectors including the circulating control vector equals the number of messages currently on their way to process P_k, $1 \leq k \leq n$. This property is maintained invariant by the implementation given below. For simplicity, we assume that no process communicates with itself, P_{n+1} is identical to P_1, an operation on a vector is defined by the operating on each of its components, and 0^* denotes the null vector.

The psuedo code for process P_j is shown in Algorithm 7.5.

```
COUNT is initialized to 0*
(a) When sending a basic message to $P_i$ ($i \neq j$):
(1)     $COUNT[i] \leftarrow COUNT[i] + 1$;

(b) The following instructions are executed at the end of all local actions
    triggered by the receipt of a basic message:
(2)      $COUNT[j] \leftarrow COUNT[j] - 1$;
(3)      if $COUNT[j] = 0$ then
(4)          if $COUNT = 0^*$
                 then "system terminated"
(5)              else send accumulate <COUNT> to $P_{j+1}$;
(6)              $COUNT \leftarrow 0^*$;
(7)          end_if;
(8)      end_if;

(c) When receiving a control message "accumulate ≤ACCU>":
(9)      $COUNT \leftarrow COUNT + ACCU$;
(10)     if $COUNT[j] \leq 0$ then
(11)         if $COUNT = 0^*$
                 then "system terminated"
(12)             else send accumulate <COUNT> to $P_{j+1}$;
(13)             $COUNT \leftarrow 0^*$;
(14)         end_if;
(15)     end_if;
```

Algorithm 7.5 Vector counters algorithm [12].

An initiator P_i starts the algorithm by sending the control message "accumulate <0*>" to P_{i+1}. A mechanism is needed to ensure that every process is visited at least once by the control message, i.e., that the control vector makes at least one complete round after the start of the algorithm.

Every process counts the number of outgoing messages individually by incrementing the counter indexed by the receiver's process number (line 1); the counter indexed by its own number is decremented on receipt of a message (line 2). When a process receives the circulating control message, it accumulates the values in the message to its *COUNT* vector (line 9). A check is then made (line 10) to determine whether any basic messages known to the control message have still not arrived at P_j. If this is the case ($COUNT[j]>0$), the control message is removed from the ring and regenerated at later time (line 5) when all expected messages have been received by P_j. For this purpose, every time a basic message is received by a process P_j, a test is made to check whether $COUNT[j]$ is equal to 0 (line 3). Note that lines 4–15 are only executed when the control vector is at P_j. Note that there is at most one process P_j with $COUNT[j]>0$, and if this is the case at P_j, the control vector "waits" at process P_j (lines 11–13 are not executed and the control vector remains at P_j).

If the control message is not required to wait at nodes for outstanding basic messages, the algorithm can be simplified considerably by removing lines 3–8 as well as lines 10 and 15.

Performance

The number of control messages exchanged by this algorithm is bounded by $n(m+1)$, where m denotes the number of basic messages, because at least one basic message is received in every round of the control message, excluding the first round. Therefore, the worst case communication complexity for this algorithm is $O(mn)$.

7.8.7 A channel counting method

The channel counting method is a refinement of the vector counter method in the following way: a process keeps track of the number of messages sent to each process and keeps track of the number of messages received from each process, using appropriate counters.

Each process P_j has n counters, $C^+_{j1}, \ldots, C^+_{jn}$, for outgoing messages and n counters, $C^-_{1j}, \ldots, C^-_{nj}$, for incoming messages. C^-_{ij} is incremented when P_j receives a message from process P_i, and C^+_{jk} is incremented when P_j sends a message to P_k. Upon demand, each process informs the values of the counters to the initiator. The initiator reports termination if $C^-_{ij} = C^+_{ij}$ for all i,j.

The method becomes more practical if it is combined with the echo algorithm, where test messages flow down on every edge of the graph and echoes proceed in the opposite direction. The value of C^-_{ij} is transmitted upwards from process P_j to P_i in an echo; whereas, a test message sent by P_i to P_j carries the value of C^+_{ij} with it. A process receiving a test message from another process (the activator), propagates it in parallel with any other process to which it sent basic messages whose receipts have not yet been confirmed. If it has already done this, or if all basic messages sent out have been confirmed, an echo is immediately sent to the activator. There are no special acknowledgement messages. A process P_i receiving the value of C^-_{ij} in an echo, knows that all messages it sent to P_j have arrived if the value of C^-_{ij} equals the value of its own counter C^+_{ij}. An echo is only propagated towards the activator if an echo has been received from each subtree and all channels in the subtrees are empty.

Formal description

Each process P_j has the following arrays of counters:

1. $OUT[i]$: counts the number of basic messages sent to P_i.
2. $IN[i]$: counts the number of basic messages received from P_i.

3. *REC*[*i*]: records the number of its messages that P_j knows have been received by P_i.

OUT[*i*] corresponds to C^+_{ji} and *IN*[*i*] to C^-_{ij}. A variable *ACTIVATOR* is used to hold the index number of the activating process and a counter *DEGREE* indicates how many echoes are still missing.

The psuedo code for process P_j is shown in Algorithm 7.6.

```
{OUT, IN, and REC are initialized to 0* and DEGREE to 0.}

(a) When sending a basic message to $P_i$:
(1)      $OUT[i] \rightarrow OUT[i]+1$;

(b) When receiving a basic message from $P_i$:
(2)      $IN[i] \leftarrow IN[i]+1$;

(c) On the receipt of a control message test $< m >$ from $P_i$ where $m \leq IN[i]$:
(3)      if $DEGREE > 0$ or $OUT = $ REC /* already engaged or
          subtree is quiet */
(4)            then send echo <$IN[i]$> to $P_i$;
(5)            else $ACTIVATOR \leftarrow i$; /* trace activating process */
(6)            $PROPOGATE$ /* and test all subtrees */
(7)      end_if;

(d) On the receipt of a control message echo $< m >$ from $P_i$:
(8)      $REC[i] \leftarrow m$;
(9)      $DEGREE \leftarrow DEGREE - 1$; /* decrease missing echoes counter */
(10)     if $DEGREE = 0$ then
                    /* last echo checks whether all subtrees are quiet */
(11)           $PROPAGATE$;
(12)     end_if;
(13)     if $DEGREE = 0$ then /*all echoes arrived, everything quiet */
(14)           send echo <$IN[ACTIVATOR]$> to $P_{ACTIVATOR}$;
(15)     end_if;

(e) The procedure PROPAGATE called at lines 6 and 11 is defined as follows:
(16) procedure PROPAGATE:
(17)        loop for $K = 1$ to $n$ do
(18)              if $OUT[K] \neq REC[K]$ then /* confirmation missing */
(19)                    send test <$OUT[K]$> to $P_k$; /* check subtree */
(20)                    $DEGREE \leftarrow DEGREE + 1$;
(21)              end_if;
(22)        end_loop;
(23) end_procedure;
```

Algorithm 7.6 Channel counting algorithm [12].

Variable *DEGREE* is incremented when a process sends a test message (line 20) and it is decremented when a process receives an *ECHO* message (line 9). If $DEGREE > 0$, it means the node is "engaged" and a test message is immediately responded to with an echo message (line 4). An echo is also returned for a test message if $OUT = REC$ (line 3), i.e., if process sent no messages at all or if all messages sent out by it have been acknowledged. Lines 10–15 insure that an echo is only returned if the arrival of all basic messages has been confirmed and all computations in the subtree finished. This is done by sending further test messages (via procedure *PROPAGATE*) after the last echo has arrived (lines 10–12). These test messages visit any of the subtree root processes that have not yet acknowledged all basic messages sent to them. The procedure *PROPAGATE* increases the value of the variable *DEGREE* if any processes are visited, thus preventing the generation of an echo (lines 13–15).

To minimize the number of control messages, test messages should not overtake basic messages. To achieve this, test messages carry with them a count of the number of basic messages sent over the communication channel (line 19). If a test messages overtakes some basic messages (and it is not overtaken by basic messages), its count will be greater than the value of the IN-counter of the receiver process. In this case, the test message is put on hold and delivered later when all basic messages with lower count have been received (guard $m \leq IN[i]$ in point (c) insures this).

The initiator starts the termination test only once, as if it had received a test $< 0 >$ message from some imaginary process P_0. On termination detection, instead of eventually sending an echo to P_0, it reports termination. Test messages only travel along those channels that have been used by basic messages; processes that did not participate in the distributed computation are not visited by test messages. For each test message, an echo is eventually sent in the opposite direction.

Performance

At least one basic message must have been sent between the two test messages along the same channel. This results in an upper bound of $2m$ control messages, where m denotes the number of basic messages. Hence, the worst case communication complexity is $O(m)$. However, the worst case should rarely occur, particularly if the termination test is started well after the computation started. In many situations, the number of control messages should be much smaller than m. The exact number of control messages involved in channel counting is difficult to estimate because it is highly dependent on communication patterns of the underlying computation.

7.9 Termination detection in a faulty distributed system

An algorithm is presented that detects termination in distributed systems in which processes fail in a fail-stop manner. The algorithm is based on the weight-throwing method. In such a distributed system, a computation is said to be terminated if and only if each healthy process is idle and there is no basic message in transit whose destination is a healthy process. This is independent of faulty processes and undeliverable messages (i.e., whose destination is a faulty process). Based on the weight-throwing scheme, a scheme called flow detecting scheme is developed by Tseng [20] to derive a fault-tolerant termination detection algorithm.

Assumptions

Let $S = P_1, P_2, \ldots, P_n$ be the set of processes in the distributed computation. C_{ij} represents the bidirectional channel between P_i and P_j. The communication network is asynchronous. Communications channels are reliable, but they are non-FIFO. At any time, an arbitrary number of processes may fail. However, the network remains connected in the presence of faults. The fail-stop model implies that a failed process stops all activities and cannot rejoin the computation in the current session. Detection of faults takes a finite amount of time.

7.9.1 Flow detecting scheme

Weights may be lost because a process holding a non-zero weight may crash or a message destined to a crashed process is carrying a weight. Therefore, due to faulty processes and undeliverable messages carrying weights, it may not be possible for the leader to accumulate the total weight of 1 to declare termination. In the case of a process crash, the lost weight must be calculated. To solve this problem, the concept of flow invariant is used.

The concept of flow invariant

Define $H \subseteq S$ as the set of all healthy processes. Define *subsystem* H to be part of the system containing all processes in H and communication channels connecting two processes in H. According to the concept of flow invariant, the weight change of the subsystem during time interval I, during which the system is doing computation, is equal to (weights flowing into H during I) $-$ (weights flowing out of H during I). To implement this concept, a variable called net_i is assigned to each process P_i belonging to H. This variable records the total weight flowing into and out of the subsystem H. Initially, $\forall i\ net_i = 0$. The following flow-detecting rules are defined:

Rule 1: Whenever a process P_i which belongs to H receives a message with weight x from another process P_j which does not belong to H, x is added to net_i.

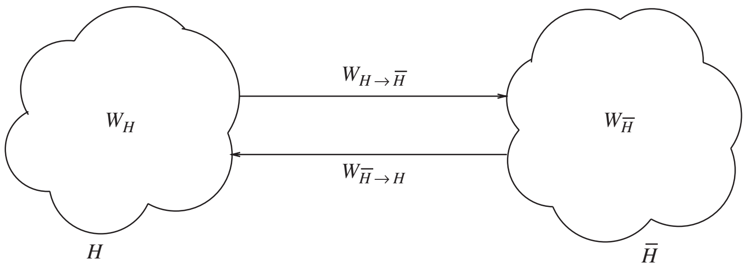

Figure 7.11 Healthy and faulty process sets and message flow between them [20].

Rule 2: Whenever a process P_i which belongs to H sends a message with weight x to a process P_j which does not belong to H, x is subtracted from net_i.

Let W_H be the sum of the weights of all processes in H and all in-transit messages transmitted between processes in H:

$$W_H = \sum_{P_i \in H} (net_i + 1/n),$$

where $1/n$ is the initial weight held by each process P_i.

Let $\overline{H} = S–H$ be the set of faulty processes. The distribution of weights is divided into four parts:

W_H: weights of processes in H.
$W_{\overline{H}}$: weights of processes in $\overline{H}$.
$W_{H \to \overline{H}}$: weights held by in-transit messages from H to $\overline{H}$.
$W_{\overline{H} \to H}$: weights held by in-transit messages from $\overline{H}$ to H.

This is shown in Figure 7.11. $W_{\overline{H}}$ and $W_{H \to \overline{H}}$ are lost and cannot be used in the termination detection.

7.9.2 Taking snapshots

In distributed systems, due to the lack of a perfectly synchronized global clock, it is not possible to get a global view of the subsystem H and hence it may not possible to determine W_H. We obtain $\overline{W_H}$, which is an estimated value of W_H, by taking snapshots on the subsystem H and by using the above equation for W_H.

However, note that weights in $W_{\overline{H} \to H}$ carried by in-transit messages may join H and change W_H. To obtain a stable value of W_H, channels from $\overline{H}$ to H are disconnected before taking snapshots of H. Once a channel is disconnected, a healthy process can no longer send or receive messages along it.

A snapshot on H is the collection of net_i's from all processes in H. A snapshot is said to be consistent if all channels from H to $\overline{H}$ are disconnected before taking the snapshot (i.e., recording the values of net_i).

A snapshot is taken upon a snapshot request by the leader process. The leader uses the information in a consistent snapshot and equation to compute W_H to calculate $\overline{W}_H$. Snapshots are requested when a new faulty process is found or when a new leader is elected. It should be noted that $\overline{W}_H$ is an estimate of the weight remaining in the system. This is because processes can fail and stop any time and there may not exist any point in real time in the computation where H is the healthy set of processes. Suppose H' is the set of healthy processes at some point in time in the computation after taking the snapshot. If $H = H'$, then $\overline{W}_H = W_{H'}$; otherwise, $\overline{W}_H \geq \mathrm{W}_{H'}$ must be true, because of the fail-stop model of processes. This eliminates the possibility of declaring termination falsely. Thus, the leader can safely declare termination after it has collected $\overline{W}_H$ of weight.

7.9.3 Description of the algorithm

The algorithm combines the weight-throwing scheme, the flow detecting scheme and a snapshot-recording scheme [20]. Process P_i elects itself the leader if it knows that all P_j, $j < i$, are faulty. The leader process takes snapshots and estimates remaining weight in the system.

Data structures

The following data structures are used at process P_i, $i=1, ...,n$:

- l_i is the identity of the leader known to P_i. Initially $1_i = 1$.
- w_i is the weight currently held by P_i. Initially $w_i = 1/n$.
- s_i is the systems total weight assumed by P_i. P_i will try to collect this amount of weight. Initially, s_i=1.
- $NET_i[1, \ldots ,n]$ is an array of real numbers. $NET_i[j]$ keeps track of the total weight flowing into P_i from P_j. Initially, $NET_i[j] = 0$ for all $j = 1, \ldots ,n$.
- F_i is a set of faulty processes. A process P_j belongs to F_i if and only if P_i knows that P_j is faulty and P_i has disconnected its channel to P_j. Initially, F_i is a null set.
- SN_i is a set of processes. When P_i initiates a snapshot, SN_i is a set of processes to which P_i sends snapshot requests. A process P_j belonging to SN_i is removed from SN_i if P_i receives a reply from P_j or if P_i finds P_j is faulty. No new snapshot is started unless SN_i is an empty set. Initially, SN_i is a null set, which implies no snapshot is in progress.
- t_i is used for temporarily calculating the total remaining weight while a snapshot is in progress.
- c_i is a boolean, used for temporarily calculating the consistency of a snapshot.

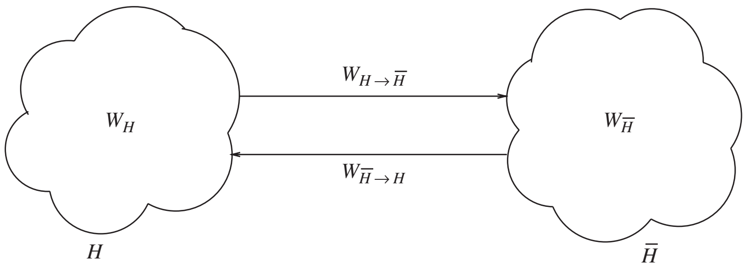
$W_{H\rightarrow\overline{H}}$
W_H
$W_{\overline{H}}$
$W_{\overline{H}\rightarrow H}$
H
$\overline{H}$

Figure 7.11 Healthy and faulty process sets and message flow between them [20].

Rule 2: Whenever a process P_i which belongs to H sends a message with weight x to a process P_j which does not belong to H, x is subtracted from net_i.

Let W_H be the sum of the weights of all processes in H and all in-transit messages transmitted between processes in H:

$$W_H = \sum_{P_i \in H} (net_i + 1/n),$$

where $1/n$ is the initial weight held by each process P_i.

Let $\overline{H} = S–H$ be the set of faulty processes. The distribution of weights is divided into four parts:

W_H: weights of processes in H.
$W_{\overline{H}}$: weights of processes in $\overline{H}$.
$W_{H\rightarrow\overline{H}}$: weights held by in-transit messages from H to $\overline{H}$.
$W_{\overline{H}\rightarrow H}$: weights held by in-transit messages from $\overline{H}$ to H.

This is shown in Figure 7.11. $W_{\overline{H}}$ and $W_{H\rightarrow\overline{H}}$ are lost and cannot be used in the termination detection.

7.9.2 Taking snapshots

In distributed systems, due to the lack of a perfectly synchronized global clock, it is not possible to get a global view of the subsystem H and hence it may not possible to determine W_H. We obtain $\overline{W}_H$, which is an estimated value of W_H, by taking snapshots on the subsystem H and by using the above equation for W_H.

However, note that weights in $W_{\overline{H}\rightarrow H}$ carried by in-transit messages may join H and change W_H. To obtain a stable value of W_H, channels from $\overline{H}$ to H are disconnected before taking snapshots of H. Once a channel is disconnected, a healthy process can no longer send or receive messages along it.

A snapshot on H is the collection of net_i's from all processes in H. A snapshot is said to be consistent if all channels from H to $\overline{H}$ are disconnected before taking the snapshot (i.e., recording the values of net_i).

When P_i is not the leader, it sends its weight to the leader process in a control message. A4 describes P_i's response on receiving a control message. In all actions A1–A4, $NET_i[1 \ldots n]$ records the weight-flowing information. In A5, leader P_i announces the termination.

Actions F1 to F4 in Algorithm 7.8 deal with faults and take snapshots of the system.

```
(* Actions for detecting a fault when no snapshot is in progress *)
F1: ($P_i$ detecting $P_j$ faulty) $\wedge$ ($P_j \notin F_i$) $\wedge$ ($SN_i = \emptyset$) $\rightarrow$
        disconnect the channel from $P_i$ to $P_j$;
        $F_i := F_i \cup \{P_j\}$;
        $l_i = min\{k \mid P_k \in S - F_i\}$;
        if ($l_i = i$), then call snapshot(); end if;

(* Actions on receiving a snapshot request *)
F2: ($P_i$ receiving Request($F_j$) from $P_j$) $\rightarrow$
        $l_i := j$;
        for every $P_f$ belonging to $F_j - F_i$, disconnect the channel $C_{i,f}$;
        $F_i := F_i \cup F_j$;
        Send a Reply($F_i$, $NET_i[1 \ldots n]$) to $P_j$;

(* Actions on receiving a snapshot response *)
F3: ($P_i$ receiving Reply($F_j$, $NET_j[1 \ldots n]$ from $P_j$) $\rightarrow$
        if ($F_i \neq F_j$) $\vee$ $\neg c_i$ then
           for every $P_f$ belonging to $F_j - F_i$, disconnect the channel $C_{i,f}$;
              $F_i = F_i \cup F_j$;
              $c_i = false$;
        else
                $t_i = t_i + 1/n + \sum_{P_f \in F_j} NET_j[f]$;
        end if;
        $SN_i = SN_i - \{P_j\}$;
        if $SN_i = \emptyset$ then
              if $c_i$ then $s_i := t_i$ else call snapshot(); end if;
        end if;

(* Actions for detecting a fault when a snapshot is in progress *)
F4: ($P_i$ detecting $P_j$ faulty) $\wedge$ ($SN_i \neq \emptyset$) $\rightarrow$
        Disconnect the channel $C_{i,j}$;
        $F_i := F_i \cup \{P_j\}$;
        $c_i := false$;
        $SN_i := SN_i - \{P_j\}$;
        if $SN_i = \emptyset$ then call snapshot(); end if;

(* Snapshot-taking procedure *)
Procedure snapshot() (* assuming the caller is $P_i$ *)
        Begin
        $SN_i = S - F_i - \{P_i\}$; (* processes that will receive requests *)
        $\forall P_k \in SN_i$, send a $Request(F_i)$ to $P_k$;
        $t_i := 1/n + \sum_{P_f \in F_i} NET_i[f]$;
        $c_i := true$;
    end;
```

Algorithm 7.8 Snapshot algorithm [20].

F1 is triggered when P_i detects for the first time that a process P_j is faulty and no snapshot is currently in progress. The channel from P_i to P_j is disconnected. Then P_i elects a healthy process with least i.d. as its leader. If process P_i itself is the leader, then it invokes a snapshot procedure to initiate a snapshot.

In the *snapshot*() procedure, first SN_i is set to the set of processes to which the Request()s are to be sent and sends a *Request*() to these processes. This prevents F1 from being executed until the snapshot finishes. Assuming that the current healthy process set is $S - F_i$ and this snapshot is consistent, more weight is added to t_i as P_i receives *Reply*() messages from other processes.

F2 describes P_i's response on receiving a *Request*() message from P_j. P_i disconnects channels to faulty processes and sends a *Reply*() message to P_j, which sent the *Request*() message.

The initiator of the snapshot P_i waits for each P_j belonging to SN_i for either a *Reply*() coming from P_j or P_j being detected as faulty.

If a *Reply*() is received from P_j, F3 is executed. F3 describes P_i's actions on receiving such a snapshot response. The consistency of the snapshot is checked. If the snapshot is still consistent, t_i is updated. Then the barrier SN_i is reduced by one. If the barrier becomes null and the snapshot is consistent, s_i is updated to t_i. If the snapshot is not consistent, another snapshot is initiated.

The snapshot initiator P_i executes F4 when it detects a process $P_j \in SN_i$, is faulty and a snapshot is in progress. Another snapshot is started only when $SN_i = \emptyset$. Such a procedure is repeated until a consistent snapshot is obtained. Because of the fail-stop model of processes, the number of healthy processes is a non-increasing function of time and eventually the procedure will terminate.

7.9.4 Performance analysis

If k processes become faulty, at most $2k$ snapshots will be taken. Each snapshot costs at most $n-1$ *Request*()s and $n-1$ *Reply*()s. Thus, the message overhead due to snapshots is bounded by $4kn$.

If M basic messages are issued, processes will be activated by at most M times. So processes will not turn idle more than $M + n$ times. So at most $M+n$ control messages $C(x)$ will be issued.

Thus, the message complexity of the algorithm is $O(M + kn + n)$.

The termination detection delay is bounded by $O(k+1)$. The termination detection delay is defined as the maximum number of message hops needed, after the remination has occurred, by the algorithm to detect the termination.

7.10 Chapter summary

A distributed computation is terminated if every process is locally terminated and there is no message in transit between any processes. Determining if a distributed computation has terminated is a fundamental problem in distributed systems. Detection of the termination of a distributed computation is a non-trivial task since no process has complete knowledge of the global state.

A number of algorithms have been developed to detect the termination of a distributed computation. These algorithms are based on the concepts of snapshot collection, weight throwing, spanning-tree, etc. In this chapter, we described a set of representative termination detection algorithms. Brzezinski *et al.* developed a very general model of the termination a distributed computation where the reception of a single message may not be enough to activate a passive process. Instead, the condition of activation is more general and a passive process requires reception of a set of messages to become active. Mattern developed several algorithm for termination detection in the atomic computation model. Tseng developed a weight-throwing algorithm to detect termination in distributed systems which allows processes to fail in a fail-stop manner.

Termination detection is a fundamental problem and it finds applications at several places in distributed systems.

7.11 Exercises

Exercise 7.1 Huang's termination detection algorithm could be redesigned using a counter to avoid the need of splitting weights. Present an algorithm for termination detection that uses counters instead of weights.

Exercise 7.2 Design a termination detection algorithm that is based on the concept of weight throwing and is tolerant to message losses. Assume that processe do not crash.

Exercise 7.3 Termination detection algorithms assume that an idle process can only be activated on the reception of a message. Consider a system where an idle process can become active spontaneously without receiving a message. Do you think a termination detection algorithm can be designed for such a system? Give reasons for your answer.

Exercise 7.4 Design an efficient termination detection algorithm for a system where the communication delay is zero.

Exercise 7.5 Design an efficient termination detection algorithm for a system where the computation at a process is instantaneous (that is, all proceses are always in the idle state.)

7.12 Notes on references

The termination detection problem was brought to prominence in 1980 by Francez [5] and by Dijkstra and Scholten [4]. Since then, a large number of termination detection algorithms having different features and for a variety of logical system configurations have been developed. A termination detection algorithm that uses distributed snapshot is discussed in [8]. A termination detection algorithm based on weight throwing is discussed in [9]. A termination detection algorithm based on weight throwing was first developed by Mattern [13]. Dijkstra *et al.* [3] present a ring-based termination detection algorithm. Topor [19] adapts this algorithm to a spanning tree configuration. Chandrasekaran and Venkatesan [2] present a message optimal termination detection algorithm. Brzezinski *et al.* [1] define a very general model of the termination problem, introduce the concept of static and dynamic terminations, and develop algorthms to detect static and dynamic terminations. Mattern developed [12] several algorithms for termination detection for the atomic model of computation. An algorithm for termination detection under faulty processes is given by Tseng [20]. Mayo and Kearns [14, 15] present efficient termination detection based on roughly synchronized clocks. Other algorithms for termination detction can be found in [6, 10, 11, 16–18, 21].

Many termination detection algorithms use a spanning tree configuration. An efficient distributed algorithm to construct a minimum-weight spanning tree is given in [7].

References

[1] J. Brzezinski, J. M. Helary, and M. Raynal, Termination detection in a very general distributed computing model, *Proceedings of the International Conference on Distributed Computing Systems*, Poland, 1993, 374–381.

[2] S. Chandrasekaran and S. Venkatesan, A message-optimal algorithm for distributed termination detection. *Journal of Parallel and Distributed Computing*, 1990, 245–252.

[3] E. W. Dijikstra, W. H. J. Feijen, and A. J. M. van Gasteren, Derivations of a termination detection algorithm for distributed computations, *Information Processing Letters*, **16**(5), 1983, 217–219.

[4] E. W. Dijkstra and C. S. Scholten, Termination detection for distributed computations, *Information Processing Letters*, **11**(1), 1980, 1–4.

[5] N. Francez, Distributed termination, *ACM Transactions on Programming Langauges*, **2**(1), 1980, 42–55.

[6] N. Francez and M. Rodeh, Achieving distributed termination without freezing, *IEEE Transaction on Software Engineering*, May, 1982, 287–292.

[7] R. G. Gallager, P. Humblet, and P. Spira, A distributed algorithm for minimum weight spanning trees, *ACM Transactions on Programming Langaugaes and Systems*, January, 1983, 66–77.

[8] Shing-Tsaan Huang, Termination detection by using distributed snapshots, *Information Processing Letters*, **32**, 1989, 113–119.

[9] S. T. Huang, Detecting termination of distributed computations by external agents, *Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989, 79–84.

[10] D. Kumar, A class of termination detection algorithms for distributed computations, *Proceedings of the 5th Conference on Foundation of Software Technology and Theoretical Computer Science*, New Delhi, LNCS 206, 1985, 73–100.

[11] T. H. Lai, Termination detection for dynamically distributed systems with non-first-in-first-out communication, *Journal of Parallel and Distributed Computing*, December, 1986, 577–599.

[12] F. Mattern, Algorithms for distributed termination detection, *Distributed Computing*, **2**, 1987, 161–175.

[13] F. Mattern, Global quiescence detection based on credit distribution and recovery, *Information Processing Letters*, **30**(4), 1989, 195–200.

[14] J. Mayo and P. Kearns, Distributed termination detection with roughly synchronized clocks, *Information Processing Letters*, **52**(2), 1994, 105–108.

[15] J. Mayo and P. Kearns, Efficient distributed termination detection with roughly synchronized clocks, *Parallel and Distributed Computing and Systems*, 1995, 305–307.

[16] J. Misra and K. M. Chandy, Termination detection of diffusing computations in communication sequential processes, *ACM Transactions on Programming Languages and Systems*, January, 1982, 37–42.

[17] S. P. Rana, A distributed solution of the distributed termination problem, *Information Processing Letters*, **17**(1), 43–46.

[18] S. Ronn and H. Saikkonen, Distributed termination detection with counters, *Information Processing Letters*, **34**(5), 1990, 223–227.

[19] R. W. Topor, Termination detection for distributed computations, *Information Processing Letters*, **18**(1), 1984, 33–36.

[20] Yu-Chee Tseng, Detecting termination by weight-throwing in a faulty distributed system, *Journal of Parallel Distributed Computing*, **25**(1), 1995, 7–15.

[21] Yu-Chee Tseng and Cheng-Chung Tan, On termination detection protocols in a mobile distributed computing environment, *Proceedings of ICPADS*, 1998, 156–163.