JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **8**, 245–252 (1990)

A Message-Optimal Algorithm for Distributed Termination Detection

S. CHANDRASEKARAN

Engineering Division, ECC, Madras 602104, India

AND

S. VENKATESAN*

Department of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania 15260

Termination detection is among the important problems in distributed processing systems. Several algorithms have been proposed to solve the problem. However, these solutions are inefficient in terms of the number of messages used in the worst case. Chandy and Misra have shown that a lower bound on the message complexity of the termination detection problem is the number of messages used by the underlying computation. We establish a new lower bound on the message complexity of the termination detection problem and present a new algorithm for the termination detection problem which achieves these lower bounds simultaneously. © 1990 Academic Press, Inc.

1. INTRODUCTION

Computer networks are becoming popular and cost effective because of the low cost of hardware and communication. In a computer network, a global problem is typically solved with the cooperation of the processors. The processors exchange information using the *communication network*. In most applications, the input to the problem to be solved is available *distributively*—each processor having some part of the input. Thus, a solution to the problem requires executing a *distributed algorithm.*

In distributed processing systems, inferring if a distributed computation has ended is essential so that the results produced by the protocol can be used. Also, in many applications, the problem to be solved is divided into many modules, and the execution of one module cannot begin until the execution of the previous module is completed and hence it is necessary to know when a particular module has ended so that the next module may begin. The problem of deciding if a given computation has ended has to be done *distributively* as no node can decide by itself. In this paper, we first define the *distributed termination detection* problem and present a solution. We then show that our algorithm is the best algorithm possible in terms of the number of messages used in any asynchronous computer system.

There are two distributed computations taking place in the network, namely, the *underlying computation* and the *termination detection algorithm.* We are interested in inferring when the underlying computation has ended and the termination detection algorithm is used to achieve this.

We need the following definitions and rules before defining the distributed termination detection problem.

1. At any time during the execution of the distributed algorithm, a processor can be in only one of the two states: *busy,* where it is doing some computation; and *idle,* where the processor has temporarily suspended the execution of its local algorithm until it receives a message from another processor.
2. A busy processor can become idle at any time. This corresponds to the situation where the processor has completed processing all of the received messages and also its local computation.
3. An idle processor can become busy only on receipt of a message from another processor. Thus, a processor cannot spontaneously become busy (except when the distributed algorithm begins execution).
4. Only a busy processor can send messages.
5. A message can be received by a processor when the processor is in either of the two states.

We assume that each node has one processor and hence, throughout the paper, the terms node and processor will be used interchangeably whenever no confusion arises. The distributed algorithm is said to have *terminated* if every processor in the network is idle and there are no messages in transit to restart an idle processor. In the distributed termination detection problem, a particular processor (or all of the processors) must infer when the underlying computation has terminated.

* Current address: Department of Computer Science, University of Texas, Richardson, Texas 75083.

Several papers have appeared in the literature and solutions have been presented for the termination detection problem. However, some of these solutions assume the existence of a Hamiltonian circuit [3] or they use $O(|V|)$ times more messages than the underlying computation itself [3, 8, 17] (here, $|V|$ is the total number of nodes in the network). Some of them assume special properties about the underlying computation [2, 4, 14]. A building block approach to termination detection is presented by Gafni [9]. The algorithms given in [15, 11] are *uniform algorithms* where all of the nodes execute the same algorithm. However, Rana's algorithm [15] assumes the existence of clock synchronization and the algorithm of Hazari and Zedan [11] assumes that the underlying network is a ring. Francez [7] presents an algorithm in which the underlying computation is *frozen* when the termination detection algorithm is executed. The algorithm given in [13] assumes that the communication links are synchronous and the termination detection algorithm starts when the underlying computation begins. Lai [12] presents a solution to the termination detection problem in a network where the links can be non-first-in–first-out. However, this algorithm uses $O(|V|)$ times more messages than the messages of the underlying distributed computation.

We consider the termination detection problem in a completely asynchronous network. Our algorithm makes no assumption about the properties of the underlying computation and no assumption is made about the topology of the network. Chandy and Misra [1] have shown that a worst case lower bound on the number of messages used by *any* termination detection algorithm in an asynchronous network is the number of messages used by the underlying computation. Let M be the number of messages used by the underlying computation. Thus, by the results of [1], $\Omega(M)$ is a (worst case) lower bound on the number of messages used by *any* asynchronous algorithm to detect the termination of the underlying computation. Let $|E|$ be the total number of links in the network. In this paper, we prove that $\Omega(|E|)$ also is a worst case lower bound on the number of messages used by *any* termination detection algorithm in our model of an asynchronous network. This lower bound, together with the lower bound of [1], establishes that $\Omega(|E| + M)$ is a lower bound on the message complexity of *any* asynchronous algorithm for the distributed termination detection problem. We then present a termination detection algorithm which uses $O(|E| + M)$ messages and hence, the *message complexity* of the termination detection problem is shown to be $\Theta(|E| + M)$.

The organization of the paper is as follows: the computation model is defined in Section 2; a new lower bound of $\Omega(|E|)$ messages is proved in Section 3; Section 4 contains an informal and a formal description of the termination detection algorithm; the proof of correctness is presented in Section 5; and the complexity analysis is presented in Section 6. Finally, Section 7 concludes the paper.

2. COMPUTATION MODEL

The following model of distributed systems is used in this paper: The distributed computer system is a network of nodes, every node having a distinct label. Each node has a set of communication links and each link connects a given node with another node of the network. These communications links are bidirectional—both of the end nodes know the existence of the link and messages may be transmitted in either direction. Call the node p a *neighbor* of the node q if p and q are connected by a communication link. A node knows the identities of its neighbors. At each node of the distributed system, there is a processor with a small amount of local memory. The local memory of one processor cannot be shared by another processor, and hence, nodes share information only by message passing.

The processors and the communication links are asynchronous (that is, the processors and the communication links incur arbitrary but finite delays in executing their function). We assume that the messages always arrive correctly and messages are always delivered in the correct order in a link (i.e., if a node v_1 sends two messages to a neighbor v_2 on the same link, then v_2 receives the two messages in the same order in which v_1 sent them). We consider point-to-point communication where a processor can generate a single message at a time and send it to one of its neighbors.

All of the nodes of the system (except the root node of the termination detection algorithm) execute the same (distributed) algorithm. At any time during the execution of the algorithm, a node may (1) send a message to a neighbor, (2) receive a message from a neighbor, (3) do some local computation, or (4) wait for a message from another node in the network. As explained earlier, a node is said to be in busy state if it is doing any of the first three actions described above. Otherwise it is in idle state.

More formally, the network is represented by a graph $G = (V, E)$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges or links. Since the links are bidirectional, E is a symmetric matrix. Let the cardinality of the node set be $|V|$ and the cardinality of the edge set be $|E|$. Our algorithm assumes the existence of a leader and a spanning tree in the network. If a leader is not available, we can run the minimum spanning tree algorithm of Gallager *et al.* [10] to elect a leader and find a spanning tree using $O(|E| + |V| \log |V|)$ messages.

Under this framework, we consider the *distributed termination detection problem.* In our model, the distributed termination detection algorithm may start its execution any time after the computation has begun. Also, the underlying computation is oblivious to the execution of the termination detection algorithm.

The performance of the distributed algorithm is measured using the worst case *communication complexity* and the worst case *time complexity*. While the *message complexity* counts the total number of messages sent over the communication links, the *bit complexity* counts the total number of *bits* sent over the links. To distinguish between the messages of the underlying computation and the messages used by the termination detection algorithm, we call the latter *control messages*. Thus, the performance of the termination detection algorithm is measured by counting the total number of control messages (and the total number of bits for bit complexity) generated by the termination detection algorithm. The *time complexity* measures the total elapsed time between the time the algorithm starts and the time the algorithm ends, assuming that messages take exactly one unit of time to traverse a link and local processing takes a negligible amount of time (our algorithm works correctly even if this assumption is not valid). It is clear that the time complexity of any algorithm is bounded above by the message complexity.

3. A NEW LOWER BOUND

In this section, we show that *any* algorithm that solves the distributed termination detection uses at least $|E|$ messages, where $|E|$ is the number of links in the network. It should be noted that this result, like the result of [1], is valid only when the network is asynchronous.

THEOREM 1. *$\Omega(|E|)$ is a lower bound on the message complexity of the termination detection problem.*

Proof. We prove this theorem by contradiction. Assume that there exists a distributed termination detection algorithm A that uses $o(|E|)$ messages to solve the problem. Thus, there is at least one link through which no message is sent by the algorithm A. Let (i, j) be the link over which the algorithm A sends no control messages. From the definition of the termination detection problem it is clear that the underlying computation may have started before the termination detection algorithm started. For proving this theorem, we will assume that the termination detection algorithm starts some time after the underlying computation begins.

We will now use an adversary to prove that the algorithm A arrives at an incorrect answer. The adversary will send a message m (as part of the underlying computation) from i to j on the link (i, j) *before* the algorithm A is started *anywhere* in the network. As the algorithm A starts *after* the message m is sent by i, the existence of the message m on the link (i, j) can be hidden from the algorithm A by the adversary as follows: The adversary will make the link (i, j) very slow and delay the message on the link (i, j) by an arbitrary long but finite amount of time. The delivery of the message m can be delayed since the nodes and the communication links are asynchronous. Since the distributed termination detection algorithm A does not send any control message on the link (i, j), the adversary can delay the delivery of the message of the underlying computation on the link (i, j) until the algorithm A terminates. Assume that there are no messages in transit in the other links and all of the nodes are in an idle state. Thus, the distributed termination detection algorithm A will terminate and announce that the underlying computation has terminated. At this time, the adversary will deliver the message m to the node j and the node j will be active on receipt of this message. It is clear that the distributed termination detection algorithm A has erroneously decided that the computation has terminated while the underlying computation is still in progress and hence the proof follows. ■

THEOREM 2. *Any asynchronous algorithm that solves the distributed termination detection problem uses at least $\Omega(|E| + M)$ messages in the worst case.*

Proof. The proof is immediate from Theorem 1 and the lower bound of [1]. ■

By using an argument similar to the argument used in Theorem 1, we can prove that any asynchronous distributed algorithm for termination detection should use at least $2*|E|$ messages (by treating each bidirectional link as a pair of unidirectional links going in opposite directions). Thus, the following theorem can be proved easily:

THEOREM 3. *At least* $\max(M, 2|E|)$ *messages must be sent by any distributed termination detection algorithm in the worst case in an asynchronous network.*

Thus, we have proved that $\Omega(|E| + M)$ messages are to be sent by any asynchronous algorithm that detects the termination of a computation distributively. In the next section, we present a new distributed termination detection algorithm which is very easy to implement and conceptually simple. Our algorithm uses $O(|E| + M)$ messages and hence it is asymptotically message optimal.

4. DISTRIBUTED TERMINATION DETECTION ALGORITHM

We now briefly describe the termination detection algorithm given in [17]. The algorithm consists of many phases. In that solution, the existence of a rooted spanning tree of the network is assumed. The root of the tree initiates one phase of the termination detection process by turning white and sending a white token to all of its children. 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 of the leaf nodes receives a white token and turns white. A white node becomes black when it sends a message to another node. When a leaf node becomes idle, it transmits a token to its parent and the token has the same color as the leaf node. An interior node waits for a token (black or white) 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. Otherwise, it sends a black token to its parent. 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 completes one phase of the termination detection algorithm. At the end of one phase, if the root node does not infer the termination of the underlying computation, the next phase is entered and the above process is repeated until termination is detected. In the worst case, this process requires $O(M|V|)$ control messages [8] as it may be necessary to repeat the above process for each message of the underlying computation and $O(|V|)$ messages are required for one phase of the termination detection algorithm. However, the termination can be detected using fewer control messages, as shown subsequently.

Let us analyze why the above algorithm is inefficient in terms of the message complexity. Consider the following scenario, where the node p sends a message m to the node q (Fig. 1). Assume that before the node q received the message m, it was idle and it had received white tokens from each of its children. Hence, the node q would have sent a white token to its parent before it received the message m. Now, the node p cannot send a white token to its parent until the node q becomes idle (otherwise, the termination detection algorithm will give incorrect results). To ensure this condition, the algorithm of [17] changes the color of the node p to black and makes it send a black token to its (p's) parent so that the termination detection process is performed once again. Thus, every message of the underlying computation can potentially cause the execution of one more phase of the termination detection algorithm. However, this problem can be overcome by using the technique presented below.

Main Idea

When p sends a message m to 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 some node p by a message m, then the sender p *waits* till q terminates before p can send a white token. To achieve this, the node q is required to send an acknowledgment (a *control* message) to the node p informing the node p that the set of actions triggered by the message m has been completed and that the node p *can* send a white token to its parent. Note that the node q, after being waked up by the

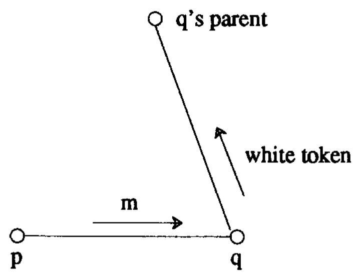

FIGURE 1

message m (from the node p), may wake up another idle node, r, which in turn may wake up other nodes. Therefore, the node q should not send an acknowledgment to p until it receives acknowledgment messages for all of the messages it sent *after* it received the message m from the node p. This restriction also applies to the node r and other nodes. Also, there may be some slow links that may transmit messages after a long delay. In fact, there may be messages that were sent before the termination detection algorithm started, and these messages may take a long time to reach their destinations. To ensure that this does not pose any problems, these messages are flushed before announcing termination.

It is clear that both the sender and the receiver keep track of each message, and a node sends a white token to its parent only after it receives an acknowledgment for every message it sent (after it knows the root node's intention to detect termination) and it receives a white token from each of its children.

Message-Optimal Termination Detection

We now present an algorithm which uses only $O(|E| + M)$ *control* messages incorporating the above idea. The above explanation presents the groundwork for our algorithm. In our algorithm, the existence of a leader or a root node v_0 and a spanning tree is assumed. Note that a leader and a (minimum) spanning tree in a distributed system can be found using $O(|E| + |V|\log|V|)$ messages [10]. In the algorithm presented below, three types of *control* messages are used—*warning, remove_entry,* and *terminated* messages.

Initially, all of the nodes in the network are in state NDT (not detecting termination) and all of the links are uncolored. Each bidirectional link is viewed as a pair of unidirectional links going in opposite directions. When the root node decides to detect the termination of the underlying computation, it spontaneously changes its state to DT (detecting termination) and sends a *warning* message on each of its outgoing links. This control message is sent so that the other nodes know of the root node's intention to detect termination and also as a request to other nodes to cooperate. The root node also colors each of its outgoing links (Fig. 2). The main reasons behind coloring the outgoing links are twofold: first, we want to make sure that the messages in transit will be received and processed by the receiving nodes before announcing the termination of the underlying computation. This is essential because the senders do not keep track of these messages, and hence the receivers should not keep track of these messages but should process them as normal messages (send and receive in state NDT). Second, the sender nodes and the receiver nodes will keep track of the messages sent over the colored links. This is to ensure that each node will not announce termination of the underlying computation until it is convinced that all of the actions triggered by every message it sent are complete.

A node p, on receiving a *warning* message from its neighbor q, colors the incoming link (q, p). Also, if the node p is in state NDT, it changes its state to DT and sends a *warning* message on each of its outgoing links. Whenever a node changes its state from NDT to DT, it colors all of its outgoing links (Fig. 3).

When a node p in state DT sends a message (generated by the underlying computation) to its neighbor q, it keeps track of this information by pushing the entry TO(q) on its local stack (Fig. 4). It should be noted that a node in state DT will send messages over colored links since all of the outgoing links are colored by the procedure *start*.

When a node x receives a message (which is generated by the underlying computation) from the node y on the link (y, x) that is colored by x, the node x knows that the sender (the node y) keeps track of the message in its local storage and the sender needs an acknowledgment from the receiver. Hence, the receiver node x keeps track of this information by pushing the entry FROM(y) on its local stack (Fig. 5).

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

When a node p becomes idle, it examines its stack from the top. For every entry of the form FROM(q), it deletes the entry and sends the *remove_entry* message to the node q. The node p repeats this until it encounters an entry of the form TO(x) on the stack. Let this operation be called *stack_cleanup* (Fig. 6). The idea behind this step is to inform those nodes that sent a message to p that the actions triggered by their message to p are complete.

When a node x receives a *remove_entry* message from its neighbor y, the 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 that information. The node x on receipt of the control message *remove_entry* from the node y, examines its stack from the top and deletes the first entry of the form TO(y) from the stack. Further, if the node x is idle, it also performs the *stack_cleanup* operation (Fig. 7).

A node sends the *terminate* message to its parent when it satisfies all of 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 local stack is empty.
4. It has received the *terminate* message from each of its children (this rule obviously does not apply to the leaf nodes).

When the root node satisfies all of the above conditions, it concludes that the underlying computation has indeed terminated. In the next section, the correctness of our algorithm is established.

```
Procedure start;

(* performed by the root node when it decides to
  detect termination of the underlying computation *)

begin
    mystate ← DT;

    for each outgoing network link ln do
        begin
            color ln;
            Send a warning message on ln
        end
end;
```

FIGURE 2

```
Procedure receive_warning;

(* performed when a node p, receives a warning message from its neighbor q *)

begin
    color the incoming link (q,p);
    if (mystate <> DT) then
        begin
            mystate ← DT;
            for each outgoing link ln do
                begin
                    color ln;
                    send a warning message on ln
                end
        end
end;
```

FIGURE 3

```
Procedure send_message(q : neighbor);

(* performed when a node p wants to send a message to its neighbor q *)

begin
    Push TO(q) on the stack;
    send the actual message to node q
end;
```

FIGURE 4

```
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;
```

FIGURE 5

```
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;
```

FIGURE 7

5. PROOF OF CORRECTNESS

To prove the correctness of the termination detection algorithm, we proceed as follows: We first show that the algorithm announces termination whenever the underlying computation terminates. The next step is to show that if the algorithm detects termination, then the underlying computation has indeed terminated.

LEMMA 1. *The algorithm given above detects termination whenever the underlying computation terminates.*

Proof. When every node is idle and there are no messages in transit, *remove_entry* messages are sent by every node to acknowledge the actual messages (of the distributed algorithm) it received from its neighbors. Eventually, all of the stacks will become empty. Also, the *warning* messages generated by each node reach their destination. Thereafter, starting with the leaf nodes, the *terminate* message is passed upward. Eventually the root node will receive the *terminate* message from all of its children. Since the root is also idle, its stack is empty and it has received the *warning* messages from each of its neighbors, the root node will conclude termination. ■

LEMMA 2. *By the time a node generates the* terminate *message, all of its neighbors are in state* DT.

Proof. A node generates the *terminate* message only after receiving the *warning* message from each of its neighbors. The neighbors have to be in state DT before sending the *warning* message. ■

```
Procedure stack_cleanup;

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

Procedure idle;

(* performed as soon as the node becomes idle *)

begin
    stack_cleanup
end;
```

FIGURE 6

LEMMA 3. *If the algorithm detects termination then the underlying computation has indeed terminated.*

Proof. To prove this lemma, we prove the following result: If the underlying computation has not terminated, then the root node cannot conclude termination. Since the underlying computation has not terminated, there must be at least one node, say p_0, which is still active. We denote this configuration by $\{p_0\}$. There are two possible cases to be considered:

Case 1. The node p_0 has not sent the *terminate* message to its parent. In this case, each ancestor of p_0 cannot send the *terminate* message to its parent. Thus, the root node will be waiting for the *terminate* message from its child which is an ancestor of p_0. Thus, the root node cannot conclude termination.

Case 2. The node p_0 has sent the *terminate* message to its parent and was later reactivated by some other (*active*) node. In this case, consider the last message m_1 received by p_0. Let p_1 be the sender of the message m_1. The configuration now becomes $\{p_0, m_1, p_1\}$. Now consider the node p_1. Again, there are two possible cases for the node p_1. Consider the first case where the node p_1, at the time of sending the message m_1 to p_0, has not sent the *terminate* message to its parent. In this case, the stack of p_1 is not empty (it contains at least the entry TO(p_0)). Thus, as long as p_0 is active, p_1's stack will not be empty and consequently, p_1 cannot send the *terminate* message to its parent. This means that the root node cannot conclude termination. On the other hand, it is possible that p_1 had sent the *terminate* message to its parent before it sent the message m_1 to p_0. This situation is possible only if p_1 was idle and was later reactivated by another node *after* it sent the *terminate* message. Consider the last message m_2 that p_1 received from p_2 (say) before it sent the message m_1 to p_0. The configuration now becomes $\{p_0, m_1, p_1, m_2, p_2\}$. The analysis of the node p_2 is similar to that of p_1. If p_2 has not sent the *terminate* message, then the proof follows. Otherwise, we continue to trace the messages backward until we reach the node p_k satisfying the following conditions:

1. The configuration is $\{p_0, m_1, p_1, m_2, p_2, \ldots, p_{k-1}, m_k, p_k\}$, as shown in Fig. 8. Here, all of the p_i's need not to be distinct. Thus p_i and p_j may be the same nodes for $i \neq j$.

2. The node p_i $(0 \leqslant i \leqslant k-1)$ has sent the *terminate* message to its parent before receiving the message m_{i+1} (which reactivated the node p_i) from p_{i+1}.

3. The message m_{i+1} was the last message received by p_i $(0 < i \leqslant k-1)$ before p_i sent the message m_i to p_{i-1}.

4. The node p_k has not sent the *terminate* message to its parent.

Clearly, such a node p_k should exist as there are only a finite number of messages. Let us consider the state of the node p_k when it sent the message m_k to the node p_{k-1}. We first show that p_k must have been in state DT at that time. The node p_{k-1} had sent the *terminate* message to its parent before it received the message m_k (by condition 2 above). Also, the node p_{k-1} should have received the *warning* message from every neighbor (in particular, the node p_k) before it sent the *terminate* message. This implies that the node p_{k-1} should have received the *warning* message from p_k before it received the message m_k. In other words, the node p_k sent the *warning* message to p_{k-1} before it sent the message m_k. Thus, the node p_k must have been in state DT when it sent the message m_k.

Since the node p_k was in state DT when it sent the message m_k, its local stack will contain the entry TO(p_{k-1}). As the stack of p_k is not empty (as long as p_0 is active), the node p_k cannot send a *terminate* message to its parent. Therefore, the root node cannot infer termination. ∎

THEOREM 4. *The termination detection algorithm announces termination if and only if the underlying computation terminates.*

Proof. The proof follows from the proofs of Lemmas 1 and 3. ∎

Segall [16] presents a unified approach to the formal description and validation of several distributed protocols. Thus, our algorithm can also be described formally, and the correctness can be established along the lines of [16]. In the next section, the communication and time complexity are analyzed.

6. COMPLEXITY ANALYSIS

We now analyze the number of *control messages* used by our 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 $|E|$ links, the total number of *warning* messages generated by the algorithm is $2*|E|$. For every message generated by the underlying computation (after the start of the termination detection algorithm), exactly one *remove_entry* 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 $|V|$ nodes and $|V|-1$ tree edges, only $|V|-1$ *terminate* messages are sent. Hence, the total number of messages generated by the algorithm is $2*|E| + |V| - 1 + M$. Thus, the message complexity of our algorithm is $O(|E| + M)$ as $|E| > |V| - 1$ for any connected network. From the lower bound of Theorem 2, it is clear that our algorithm is asymptotically optimal in the number of messages used. Our algorithm also achieves an improvement of $|V|$ over the previous algorithms.

We now consider the *bit* complexity of our algorithm. Note that the *warning*, *remove_entry* and *terminated* messages are of constant size as they do not contain any parameters and hence the *bit* complexity of our algorithm is also $O(|E| + M)$.

THEOREM 5. *The worst case* message *and the* bit *complexities of the termination detection algorithm are both* $O(|E| + M)$.

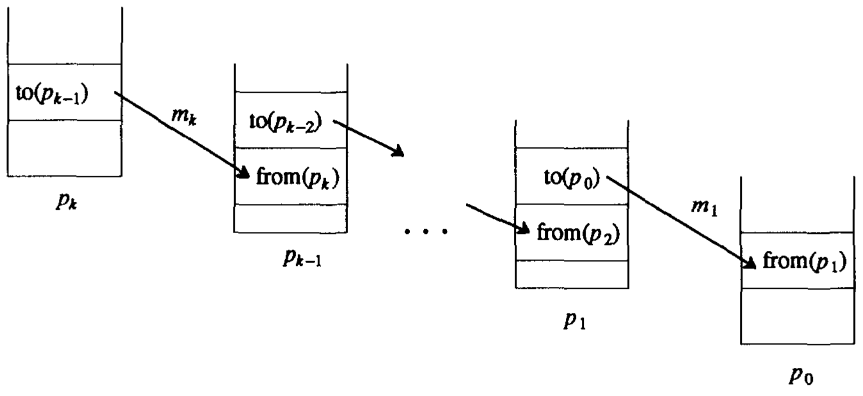

FIGURE 8

The worst case *time* complexity of our algorithm can be as high as $O(M + |V|)$ as we can think of a configuration with M nodes, and it takes M time to infer the termination and $|V| - 1$ time to announce the result to the other nodes. The following result for the distributed termination detection problem is immediate from Theorems 1 and 5 and the lower bound of [1].

THEOREM 6. *The message complexity of the termination detection problem under the asynchronous communication model is* $\Theta(|E| + M)$.

7. CONCLUSION

We have provided a new approach to the termination detection problem. As shown in the previous sections, the algorithm presented in this paper is asymptotically message optimal. If the bit complexity is considered, the algorithm uses *only* $O(M + |E|)$ bits for termination detection and hence, it is a very efficient algorithm in terms of the number of bits used by the termination detection algorithm. Our algorithm is very easy to implement because of its simplicity. Also, no assumption is made about the underlying computation and the network is assumed to be an asynchronous network and hence, our algorithm works correctly in synchronous networks, ring networks, etc. Our algorithm could take $O(M + |V|)$ *time* in the worst case, as shown in the previous section. It may be interesting to see if the *time* complexity can be improved while having the same *message* complexity.

One possible extension would be to devise fault-tolerant algorithms for the termination detection problem. The case of link failure can be handled more easily than the case of node failure as given by [6]. The link failure case can be handled trivially as follows: In the worst case, each message on a link can be replaced by a broadcast on the network (assuming that the network is connected in spite of the failure of some links). However, this will drastically increase the message complexity of the termination detection algorithm and a better approach is required to reduce the message complexity.

For the node failure case, different failure modes can be considered. A generally accepted and commonly used failure mode is fail-stop [5], where the faulty processors simply halt. In the fail-stop failure mode also, there are many network parameters to be considered (processors being synchronous/asynchronous, links being synchronous/asynchronous, broadcast capability, message order, etc.). However, there are only certain cases in which consensus can be achieved. We should choose only those cases where consensus can be reached as any termination detection algorithm implicitly solves the consensus problem.

Received August 18, 1987; revised August 17, 1988

REFERENCES

1. Chandy, K. M., and Misra, J. How processes learn. *Distrib. Comput.* **1,** 1 (1986), 40–52.
2. Cohen, S., and Lehmann, D. Dynamic systems and their distributed termination. *Proc. ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, 1982.
3. Dijkstra, E. W., Feijen, W. H., and van Gasteren, A. J. Derivation of a termination detection algorithm for distributed computations. *Inform. Process. Lett.* **16,** 5 (June 1983), 217–219.
4. Dijkstra, E. W., and Scholten, C. S. Termination detection for diffusing computations. *Inform. Process. Lett.* **11,** 1 (Aug. 1980), 1–4.
5. Dolev, D., Dwork, C., and Stockmeyer, L. On the minimal synchronism needed for consensus. *J. Assoc. Comput. Mach.* **34,** 1 (Jan. 1987), 77–97.
6. Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *J. Assoc. Comput. Mach.* **32,** 2 (Apr. 1985), 374–382.
7. Francez, N. Distributed termination. *ACM Trans. Programming Languages Systems* **2,** 1 (Jan. 1980), 42–55.
8. Francez, N., and Rodeh, M. Achieving distributed termination without freezing. *IEEE Trans. Software Engrg.* **SE-8,** 3 (May 1982), 287–292.
9. Gafni, E. Perspectives on distributed network protocols: A case for building blocks. *Proc. IEEE Military Computer Conference,* Monterey, CA, 1986.
10. Gallager, R. G., Humblet, P. A., and Spira, P. M. A distributed algorithm for minimum weight spanning trees, *ACM Trans. Programming Languages Systems* **5,** 1 (Jan. 1983), 66–77.
11. Hazari, C., and Zedan, H. A distributed algorithm for distributed termination. *Inform. Process. Lett.* **24,** 5 (Mar. 1987), 293–297.
12. Lai, T. H. Termination detection for dynamically distributed systems with non-first-in-first-out communication. *J. Parallel Distrib. Comput.* **3,** 4 (Dec. 1986), 577–599.
13. Lozinskii, E. L. A remark on distributed termination. *Proc. International Conference on Distributed Computing Systems,* 1985.
14. Misra, J., and Chandy, K. M. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Programming Languages Systems* **4,** 1 (Jan. 1982), 37–43.
15. Rana, S. P. A distributed solution of the distributed termination problem. *Inform. Process. Lett.* **17,** 1 (July 1983), 43–46.
16. Segall, A. Distributed network protocols. *IEEE Trans. Inform. Theory* **IT-29,** 1 (Jan. 1983), 23–35.
17. Topor, R. W. Termination detection for distributed computations, *Inform. Process. Lett.* **18,** 1 (Jan. 1984), 33–36.

S. CHANDRASEKARAN received the B.E degree from Karnataka Regional Engineering College and the M.E degree from the Indian Institute of Science in 1977 and 1979, respectively. He joined ECC, Madras, in 1984, where he is currently a senior engineer. His interests include distributed processing systems.

S. VENKATESAN received the B.Tech. and the M.Tech. degrees from the Indian Institute of Technology, Madras, in 1981 and 1983, respectively, and the M.S. and Ph.D degrees in computer science from the University of Pittsburgh in 1985 and 1988. He joined the University of Texas at Dallas in January 1989, where he is currently an assistant professor of computer science. His interests are in distributed processing systems, distributed algorithms, and testing and debugging distributed programs.