Distributed Termination Detection for Dynamic Systems

D.M. Dhamdhere **Sridhar Iyer** **E.K.K.Reddy**
Dept of Computer Science
IIT Bombay
Mumbai, INDIA

Abstract

A symmetric algorithm for detecting the termination of a distributed computation is presented. The algorithm does not require global information concerning the system and does not assume any communication features barring finite delays in the delivery of messages. It permits dynamic creation and destruction of processes participating in the computation. It provides for destruction of a process by external processes, such as the OS kernel. It also provides for external processes spontaneously joining an ongoing computation. Proofs of safety and liveness are provided.

Keywords Distributed algorithms, Distributed computation, Distributed termination, Dynamic systems, Termination detection.

1 Introduction

Termination detection, i.e. determining whether a distributed computation being performed in a system has terminated, is a fundamental problem in distributed programming. The distributed computation being performed is known as the *basic computation* and the inter-process messages used for implementing it are known as the *basic messages.* For termination detection, an additional computation known as the *control computation* is superimposed on the basic computation. The messages used to implement the control computation are known as the *control messages.*

A process belonging to the computation is either *live* or *dead.* A *live* process is in the *active* state if it is currently engaged in the basic computation, or *passive* state if it is currently not performing any computation. An active process can send or receive basic messages, create other processes, become passive, or destroy processes (including itself). A passive process can receive basic messages, in which case it re-enters the *active* state, or it may be killed by another process. A killed process becomes dead and ceases to exist. Such a process can neither send or receive messages of any kind, nor become *live* or create new processes. In our work, we assume that a dead process vanishes without a trace – in particular, that it ceases to participate in the control computation.

A distributed computation is said to have terminated when all its *live* processes are in the *passive* state and no basic messages are in transit. This is called the distributed termination condition (DTC). The control computation performed by the processes for detecting the termination constitutes the *termination detection algorithm.* In this paper we develop a termination detection algorithm for dynamic systems, i.e. for systems permitting creation and destruction of processes. The algorithm is symmetric, and requires only the live processes of a computation to participate in it. The algorithm does not require global information concerning the system, and does not assume any communication features barring finite delays in the delivery of messages. It also provides for the inclusion of external processes into an ongoing computation.

1.1 Other Work

The distributed termination problem has been extensively studied. One of the earliest works in this area is an algorithm by Dijkstra and Scholten, based on the *diffusing computation* model [7]. This approach was later extended by Misra and Chandy [22] and Cohen and Lehmann [5]. The algorithms in this area can be broadly classified as "symmetric" and "asymmetric" algorithms. In the *symmetric* algorithms [9, 12, 14, 24, 26] all the processes execute identical code and any process can detect the termination. The *asymmetric* algorithms [7, 8, 11, 17, 20, 22, 29] rely on a pre-designated process for termination detection. Termination detection algorithms may also be divided into three classes depending upon the topology employed for the control communication, viz. Hamiltonian cycle [8, 24, 26], spanning tree [2, 5, 10, 11, 17, 19, 20, 29] and general networks [7, 9, 12, 14, 15, 21, 22, 27].

In the algorithms of [5, 7, 22], the processing of messages used for termination detection results in the slowing down of the actual computation. The algorithms of [12, 14, 17, 24] assume the existence of local clocks. These algorithms incur the overheads of synchronizing the local clocks using messages containing time-stamps [18]. The algorithms of [9, 24, 26] assume global information in terms of either the diameter of the network or the number of processes. In [16, 17, 20], the processes use the technique of message counting.

1.2 Dynamic Systems

Most of the algorithms mentioned above work only for *static* systems, i.e. for systems comprising of a fixed set of processes. There is relatively less work on *dynamic* systems, where processes may be created as well as destroyed while the computation is in progress. The task of termination detection in dynamic systems is more difficult because the exact number of processes participating in the computation is not known at any instant of time. Also, since processes may destroy themselves, the algorithm has to ensure that (i) the computation does not get partitioned, and (ii) a process capable of detecting termination always exists in the system. As a result, termination detection algorithms for dynamic systems are more complex than those for static systems.

The algorithms of [5, 7, 22] are concerned with special cases of dynamic systems, viz. systems with synchronous communication in which processes may be created but not destroyed. Lai [17] gives an algorithm for dynamic systems where processes may be created and destroyed, the communication may be synchronous or asynchronous and messages need not be delivered in the FIFO order. However, the algorithm is asymmetric. It also requires a process to participate in termination detection even after it has been destroyed.

1.3 Overview of the paper

We develop a symmetric termination detection algorithm for dynamic systems permitting unrestricted creation and destruction of processes. A distributed computation is assumed to be structured in the form of a set of concurrent processes $\{P_i\}$, with each process performing a specific computational task. Processes can be created or destroyed during the course of the computation. A process becomes *passive* on completing the computational task assigned to it, and awaits one of the following events : (a) assignment of a new computational task (through the receipt of a basic message from some other process), (b) receipt of a message killing it, or (c) declaration of global termination. In case (a), the process becomes *active* again, in case (b), it informs other processes of its destruction and dies, while in case (c), it simply terminates. The interprocess communication is assumed to be asynchronous with arbitrary but finite delays and possible non-FIFO delivery of messages.

Each process maintains a data structure called the *neighbour set* (NB) to store the identities of its neighbours. Whenever two processes P_i and P_j communicate with one another, their NB's are updated with each other's id's. Similar actions are performed when process P_i creates process P_j. When a process P_i kills process P_j, or P_j kills itself, P_j informs each $P_k \in NB_j$ of its destruction. This leads to the deletion of P_j from NB_k. Certain new processes may be added to NB_k to avoid network partitioning. This ensures that all existing processes participate in the termination detection.

Termination detection is performed in a symmetric manner. The basic model is that of a diffusing computation [7]. Every time a process becomes *passive*, it initiates termination detection by sending a *termination detection* (TD) message to each process in its NB. To limit the communication overheads, each TD message contains a *broadcast set* (BS) containing the id's of all processes to which the TD message has been (or is being) sent. A *passive* process receiving the message propagates it to those of its neighbours which are not already in the BS. It replies with a *ready to terminate* (RT) message only when each such neighbour replies with an RT, or when it has no neighbours to whom it should send the termination detection message. The process initiating the termination wave concludes that termination has occurred when it receives the RT messages from all its neighbours.

Each process has an unique *process identification number* (PI) which is an ordered pair (*sequence number*, *process id*). Sequence numbers are used analogous to synchronized logical clocks. The sequence number of a process is initially zero, gets incremented by one every time the process initiates termination detection and is updated whenever a control message with a higher sequence number is received. A process ignores a TD message if it is *active*, or has a PI larger than that of the initiator of the TD message. This ensures correctness of termination detection and also minimizes the control message traffic. Termination is thus detected by the process with the highest identification number.

We develop our algorithm in two stages. In the main algorithm, the receiver of a basic message is required to send an acknowledgement to the sender. Each process P_i maintains a counter (CTR_i) to record the number of basic messages sent by it for which no acknowledgements have been received. A process cannot become *passive*, or be killed, as long as its CTR is non-zero. Proofs of the safety and liveness properties of the algorithm are provided.

In the second stage (section 6), we extend the algorithm to eliminate the need for acknowledgements to the basic messages, leading to a more efficient algorithm. The proofs of the main algorithm are shown to be applicable to the extended algorithm as well. Both, the main and the extended algorithms, allow killing of a process by external processes that are not part of the computation, viz. killing by the OS kernel. An extension permitting new processes to spontaneously join ongoing computations is also reported. This has particular relevance to multi-file, multi-client transactions in data base systems [28].

2 System Model

A distributed computation is a pair

$(PMSET, CG)$, where

$PMSET$ is the non-empty set of *permanent processes* with which the system begins its execution. A permanent process exists during the entire lifetime of the computation, i.e. it cannot be killed.

$CG = (P, E)$ is an undirected graph, wherein P is a finite set of processes $\{P_i \mid P_i \text{ is } \textit{live}\}$, and E is a set of edges $\{(P_i,P_j) \mid P_i \in NB_j,\ P_j \in NB_i\}$.

At system initiation, $CG = (PMSET, E_0)$, where $E_0 = \{(P_i, P_j) \; \forall \; P_i, P_j \in PMSET\}$, i.e. all permanent processes are connected to each other. A new edge is added to CG when two processes communicate for the first time. When a new process P_i is created, a vertex P_i and an edge between P_i and its creator are added to CG. When a process is destroyed, the corresponding vertex and all edges connected to it are removed from CG and some new edges are added to avoid system partitioning. This is explained further in section 3.1.

2.1 Interprocess Communication

The physical network for interprocess communication is a connected single component graph with bi-directional links. A message contains the identity of its sender and receiver, and may be of variable length. Messages are delivered within an arbitrary but finite delay, and not necessarily in the order in which they were sent.

If a basic message is sent to a non-existent process, the system sets a *not delivered* ($NOTDEL$) flag and returns the message to the sender. It is assumed that a sender process would take appropriate action in such a situation.

2.2 Features of the processes

Each process P_i maintains the following information :

CLR_i : Colour, i.e. state, of process P_i. The possible values for colour are :

- *White* : The process is active.
- *Black* : The process is passive, has propagated a 'termination detection' (TD) message and is waiting for replies.
- *Red* : The process is passive and has sent a 'ready to terminate' (RT) reply to a 'termination detection' (TD) message.

Seq_i : Current sequence number of P_i.

S_i : Id of some permanent process. S_i is known as the *static* of P_i. S_i of a permanent process is its own id, i.e. i itself.

NB_i : Set of id's of the *neighbours* of P_i.

CTR_i : Count of basic messages sent by P_i for which no acknowledgement has been received.

PAR_i : Id of the process from which the latest termination detection message was first received by P_i.

$Deadset_i$: Set of all processes which are known to be *dead*. (In the main algorithm, this information is maintained only by the permanent processes.)

$TDSET_i$: The set of processes who have yet to reply with an RT to the latest TD message propagated by P_i.

LPI_i : Identification number in the latest control message propagated by P_i. This is the ordered pair (Seq_j, id_j) of the originator process P_j of the message, at the time when the message was originated.

Two relations are defined over the process identification numbers in the system. These are :

Greater than ('$\cdot\!>$') defined as

$$(Seq_i, i) \ \cdot\!> \ (Seq_j, j) \Longleftrightarrow ((Seq_i > Seq_j) \text{ or } ((Seq_i = Seq_j) \ and \ (i > j)))$$

Same as ('$\doteq$') defined as

$$(Seq_i, i) \ \doteq \ (Seq_j, j) \Longleftrightarrow ((Seq_i = Seq_j) \ and \ (i = j))$$

The *greater than* relation defines a total order which is useful in restricting the communication overheads. For simplicity, in the following we will use the symbols '>' and '=' for $\cdot\!>$ and $\doteq$, respectively.

2.3 Control messages

The termination detection algorithm uses the following control messages.

Acknowledgement (ACK) message : (*sender*, (ACK, Seq_{sender}), *receiver*) is an acknowledgement that *sender* received a basic message from *receiver*. Seq_{sender} is the sequence number of *sender* at the time of sending the acknowledgement.

Termination Detection (TD) message : (*sender*, (TD, PI_i, BS_{sender}), *receiver*) is used to broadcast the passive condition of some process i. *sender* is the process sending the message to *receiver*. BS_{sender} is the set of processes to whom the TD message is being propagated or has already been propagated.

Ready for Termination (RT) message : (*sender*, (RT, PI_i), *receiver*) indicates that *sender*, and each process to which it propagated the TD message with identification number PI_i, is ready to terminate.

Kill Process (KP) message : (*sender*, (KP, PI_{sender}), *receiver*) indicates that *sender* wants *receiver* to kill itself.

Killed Myself (KM) message : (*sender*, (KM, PI_{sender}, S_{sender}, NB_{sender}), *receiver*) indicates that *sender* has killed itself.

Terminate (TER) message : (*sender, (TER, BS), receiver*) indicates that the processes engaged in the computation may terminate.

3 The Main Algorithm

In the main algorithm, we assume that the receiver of a basic message sends an acknowledgement to the sender. Further, each process P_i maintains a counter (CTR_i) to record the number of basic messages sent by it, that are yet to be acknowledged. P_i can become passive, or be killed, only when CTR_i is zero.

Depending on its state, the functioning of each process is determined by a set of rules. Each rule is *atomic*, i.e., a process may not be interrupted while following a rule. Figure 1 gives the state transition diagram of a process. Figures 2, 3 and 4 give the rules followed by white, black and red processes respectively. Control messages have the formats described in section 2.3.

3.1 Important features of the algorithm

Note the following important features of the algorithm :

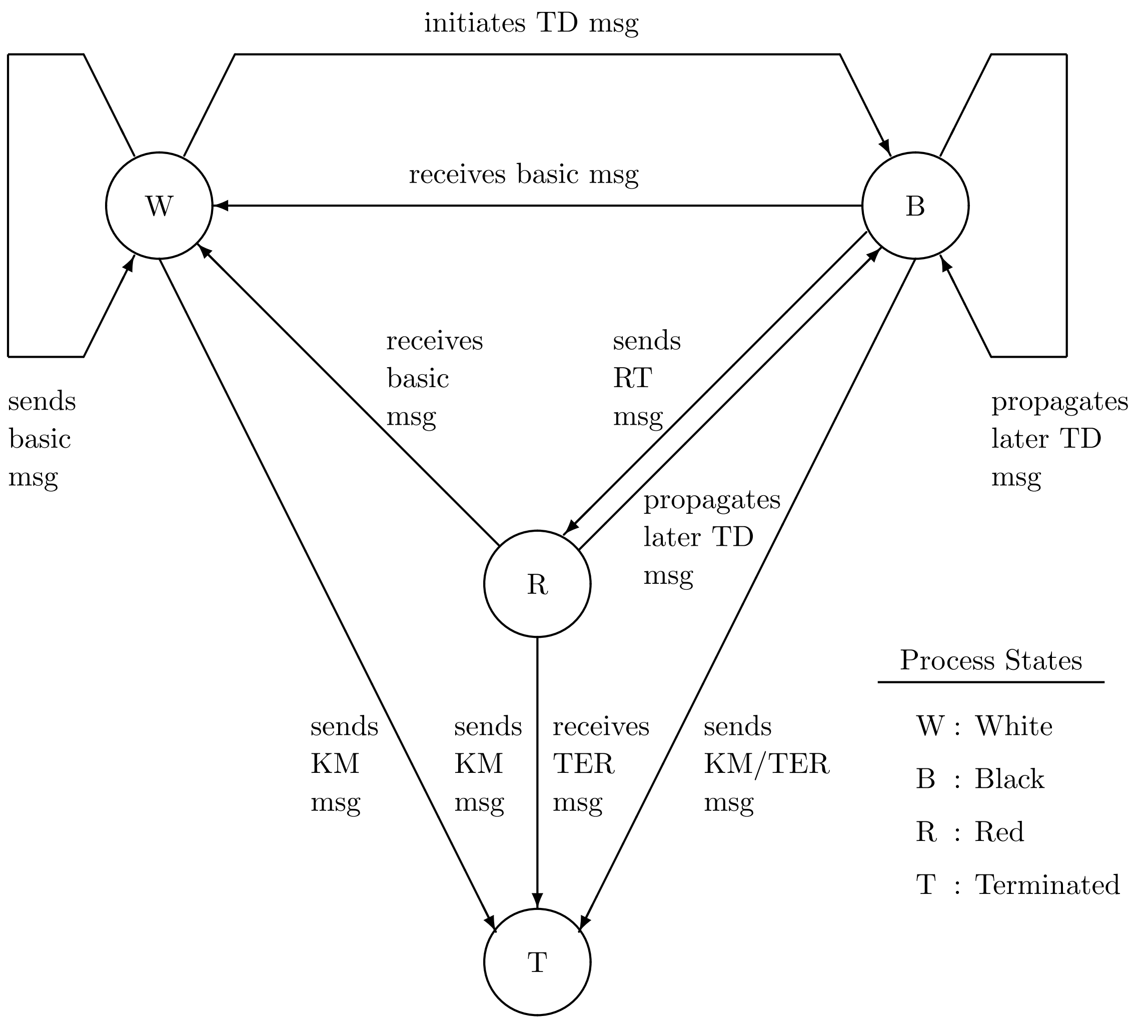

Figure 1: Process State Transition Diagram

1. The algorithm makes no specific assumptions about the network actions when the destination process of a control message is found to be dead. Appropriate processing of KM messages can handle this situation. Consider a TD message sent by process P_i to process P_j, which is killed in the meanwhile. Sometime in future, P_i (if it still exists) will receive the KM message indicating that P_j has been killed. Now, P_i simply deletes P_j from its TDSET (Rules B5, R5).

2. When a process sends an acknowledgement to a basic message, it includes its own sequence number in the acknowledgement. This is useful in avoiding spurious declaration of termination. Consider a situation in which a process P_i sends a basic message to a *red* process P_j which has replied to a TD message TD^* with an RT. The sequence number of P_j now equals the sequence number in TD^* (rules B2, R2). On receiving the acknowledgement from P_j, P_i updates its own sequence number if necessary (rule W2(e)). When P_i turns *black*, its sequence number becomes greater than that in TD^* (Rule W6). Now if P_i receives TD^* some time in the future, it will ignore the same.

3. When a process is killed,

   (a) edges are added between its static process and each of its neighbours. This ensures continued connectivity of the system graph. The fact that the static process is connected to all other permanent processes in the system is useful when many processes are killed within a short interval.

   (b) all its neighbours and all permanent processes update their sequence numbers and if passive, initiate TD messages with the new sequence numbers. These actions are necessary for the following reason : The killed process may have had the highest sequence number (and hence the highest PI) in the system. Thus, it may have rejected TD messages initiated by other processes in the system, and/or may itself have been in the process of termination detection at the point of its killing. Following its removal from the system, a process with a higher sequence number must initiate TD messages to ensure termination detection.

4. When an external process kills a process of the computation, action corresponding to the receipt of a KP message is taken.

4 An example

The example presented in this section illustrates the following features of the system's functioning :

(i) How activation of a *red* process (i.e. a process which has replied to a TD message with an RT) does not lead to a false declaration of termination,

(ii) How the system graph is transformed to preserve connectivity when a process is killed, and

(iii) How distributed termination is concluded by the algorithm.

Figure 5 shows a system consisting of three processes. A is the only permanent process, hence it is the static for C and D. Let processes A and D be *white*, process C be *black* and let no basic or control messages be in transit. Let process A turn *black* and let $Seq_A = x$ such that x is the largest sequence number in the system and $PI_A = (x, \text{A})$ is the maximum of all PI's in the system. The TD message from A is received by C which replies with an RT message and turns *red*. Hence $PI_C.Seq = PI_A.Seq = x$ (Rule B2). Now, consider the following sequence of events :

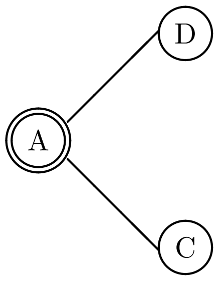

Figure 5: Initial configuration of the system

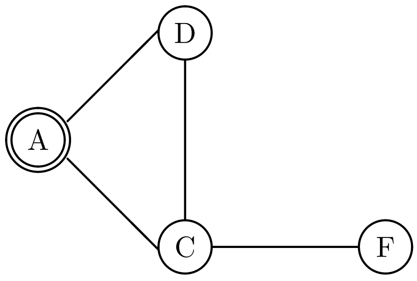

Figure 6: System after creation of process F

1. C receives a basic message from D, sends ACK and becomes active.
2. C creates a new process F, and sets $Seq_F = Seq_C$ (Rule W3). The system now looks as shown in Figure 6.
3. D receives the ACK for the basic message sent to C and sets $Seq_D = Seq_C = x$ (Rule W2(e)).
4. D becomes *black* and sends its own TD messages with $PI = (x + 1,$ D$)$ to C and A (Rule W6).
5. D receives the TD initiated by A and ignores it because its own PI is higher. (See steps 3 and 4.) Hence A cannot conclude termination, thus illustrating feature (i) mentioned above.
6. C kills itself sending KM messages to A, D and F (Rule W5).
7. A receives the KM message from C, and adds F to NB_A. It sets $Seq_A = x + 1$, initiates new TD messages and sends them to D and F (Rule B5).
8. D receives the KM message from C. C is now removed from $TDSET_D$. No processes are added to NB_D, since $S_C =$ A is already in NB_D.
9. A receives the TD sent by D (with $PI = (x + 1,$ D$)$), and ignores it because its own PI is higher (Rule B2).
10. F receives KM from C with $PI = (x,$ C$)$. It sets $PI_F.Seq = x + 1$ and adds A to NB_F (rule W2(c)). The new configuration of the system is the connected graph shown in Fig. 7. This illustrates feature (ii) mentioned above.
11. F becomes *black* and sends a TD message with $PI = (x + 2,$ F$)$ to A.

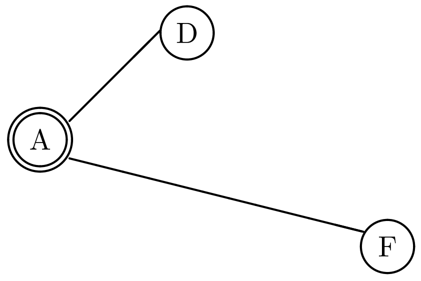

Figure 7: System configuration after process C is killed

12. F receives A's TD with $PI = (x + 1, \text{A})$ and ignores it because its own PI is higher (Rule B2).
13. D receives A's TD message and replies with an RT.
14. A receives F's TD, and propagates it to D.
15. D receives F's TD from A and replies with an RT.
16. Finally, A returns RT to F, and F concludes termination, thus illustrating feature (iii) mentioned at the start of this section.

5 Proofs of correctness

5.1 Preliminaries

To prove the correctness of a distributed termination detection algorithm, one must prove two different types of properties about it, viz. *safety* and *liveness*. Informally, a safety property stipulates that some 'bad thing' does not happen when an algorithm is used. Examples of safety properties include mutual exclusion, deadlock freedom, etc. A liveness property stipulates that a 'good thing' eventually happens during execution. Examples of liveness properties include starvation freedom, termination, etc.

To define the liveness and safety properties, we assume a predicate $P_i.TD$ which evaluates to *true* when process P_i detects termination. The liveness and safety properties of the algorithm are now stated as follows:

Safety ... $P_i.TD \;\Rightarrow\; DTC$

Liveness ... $DTC \leadsto P_j.TD$ *for some process* P_j

where '$\Rightarrow$' and '$\leadsto$' are the operators *implies* and *leads to* and DTC stands for the distributed termination condition. We prove the correctness of the algorithm using superposed variables [4] and temporal logic [23]. We use '$\Diamond A$' to denote "predicate A *eventually* becomes TRUE".

Superposed Variables

These variables are introduced exclusively for use in the proofs. The algorithm does not use them.

- $P \;\rightarrow \{\text{Processes } P_i \mid P_i \text{ is } live\}$.

- $CG \rightarrow$ communication graph (P, E) where $E = \{(P_i, P_j) \mid P_i \in NB_j \ \bigvee P_j \in NB_i\}$
- $BS_i \rightarrow$ broadcast set in the TD message sent by P_i.
- $TD\_path(P_i, P_j)$ is the path along which P_j received the TD sent by P_i for the first time. Note that all processes P_k on this path are passive, $PI_i \geq PI_k$, and $\text{PAR}_k \ \forall\ k$ and PAR_j lie along this path.
- $passive\_time(P_i, PI_i) \rightarrow$ time at which process P_i with the process identification number PI_i became passive.
- $quiescent\_time(PI_i) \rightarrow$ smallest instant of time t_i at which no TD/RT messages with the id PI_i are in transit.

Predicates

- $DTC \rightarrow$ Distributed termination has occurred.
- $DTC_t \rightarrow$ DTC has occurred at a time $< t$.
- $P_i.TD \rightarrow$ process P_i declares termination.
- $active(P_i) \rightarrow$ process P_i is active.
- $passive(P_i) \rightarrow$ process P_i is passive.
- $sent(P_i, msg, P_j) \rightarrow$ P_i has sent *msg* to P_j.
- $received(P_i, msg, P_j) \rightarrow$ P_j has received *msg* from P_i.
- $quiescent(PI_i) \rightarrow$ No TD/RT messages with the id PI_i are in transit.

5.2 Safety and Liveness

Lemma 1 *$\neg DTC \Rightarrow \exists P_i \in P$ such that $active(P_i)$.*

Proof $\neg$DTC $\Rightarrow \exists P_i \in P$ such that active(P_i) $\bigvee$ basic message in transit.
Now, basic message in transit $\Rightarrow \exists P_i$ such that $CTR_i \neq 0$ ...Rule W1.
Hence, basic message in transit $\Rightarrow \exists P_i \in P$ such that active(P_i).
Hence, $\neg$DTC $\Rightarrow \exists P_i \in P$ such that active(P_i).
□

Lemma 2 *If $(passive(P_j) \bigwedge \neg DTC_t)$ where $t = quiescent\_time(PI_j)$, then $\exists P_i \in P - \{P_j\}$ such that $\diamond$ $received(-, (TD, PI_j, -), P_i) \bigwedge \neg sent(P_i, (RT, PI_j), -)$.*

Proof Consider a path $P_j \ \ldots \ P_l - P_k - P_i$ such that $\exists$ TD_path(P_j, P_l), $PI_j > PI_l$, P_l, P_k are passive and P_i is active. Consider two cases concerning the set of processes $\{P_a \mid \text{active}(P_a)\}$ during the interval passive_time(P_j, PI_j) to quiescent_time(PI_j).

1. $\{P_a\}$ remains unchanged :
   (a) Let $PI_k > PI_j$. Then $\neg\diamond$ sent$(P_k, (RT, PI_j), -)$ ... Rule B2.
   (b) P_k is killed :
   Hence, sent$(P_k,\ (KM, PI_k, S_k, NB_k),\ P_x)$ $\forall P_x \in \{P_l, S_k, P_i\}$ ... Rule B4.
      (i) P_l receives the KM message after sending the TD message with id PI_j to P_k :
      $\neg\diamond$ sent$(P_l, (RT, PI_j), -)$ ... Rule B5.

(ii) P_l receives the KM message before receiving the TD message with id PI_j :
$\diamond$ sent$(P_l, (TD, PI_j, -), S_k)$.
Now, either $\neg\diamond$ sent$(S_k, (RT, PI_j), -)$ analogous to cases 1(a), 1(b)(i) above, or $\diamond$ received$(S_k, (TD, PI_j), P_i)$ which reduces to case (c) below.

(c) Otherwise : $\exists$ TD_path(P_j, P_i). Now, $\neg\diamond$ sent$(P_i, (RT, PI_j), -)$ since active(P_i).

2. $\{P_a\}$ grows in size :
Let process P_k become active.

(a) P_k becomes active before receiving the TD message with id PI_j :
$\neg\diamond$ sent$(P_k, (RT, PI_j), -)$ since active(P_k).

(b) P_k is activated after it propagates the TD to its neighbours :
$PI_k.Seq = PI_j.Seq \ldots$ Rules B2, R2.
Let P_g be the sender of the basic message that activated P_k. Now either,

(i) $PI_g.Seq > PI_j.Seq$, if P_g received the acknowledgement from P_k and subsequently turned passive (Rules W2(e), W6), or

(ii) active(P_g).

Hence $\neg$sent$(P_g, (RT, PI_j), -)$.

From 1 and 2, $\exists P_i$ such that $\neg$sent$(P_i, (RT, PI_j), -)$ along every path $P_j \ \ldots \ P_l - P_k - P_a$ where P_a is an active process. Since $\neg\text{DTC}_t \Rightarrow \{P_a\} \neq \phi$, at time t, the lemma follows.
□

Theorem 1 (Safety) $\quad P_j.TD \ \Rightarrow \ DTC.$

Proof

Let $P_j.TD \ \bigwedge \ \neg DTC$.
$\Rightarrow$ passive$(P_j) \bigwedge \neg DTC_t$, where $t =$ quiescent_time(PI_j).
$\Rightarrow \diamond$ received$(-, (TD, PI_j, -), P_i) \ \bigwedge \ \neg$sent$(P_i, (RT, PI_j), -) \ldots$ lemma 2.
$\Rightarrow \neg P_j.TD \ldots$ Rule B3.

This is a contradiction, hence $P_j.TD \Rightarrow DTC$.
□

Lemma 3 *If $(passive(P_j) \bigwedge DTC_t)$ where $t = passive\_time(P_j, PI_j)$, then either*

(a) $\diamond$ received$(-, (TD, PI_j, -), P_i)$, where $PI_i > PI_j$, or

(b) $\diamond$ $P_j.TD$.

Proof From arguments similar to those in lemma 1, for each path $P_j - P_1 - P_2 - \cdots - P_n$, either
$\diamond$ received$(-, (TD, PI_j, -), P_i)$, where $PI_i > PI_j$, or
$\diamond$ received$(-, (TD, PI_j, -), P_n)$, where $((PI_n < PI_j) \ \bigwedge \ (\{NB_n - BS\} = \phi))$.
Hence, either

(a) $\diamond$ received$(-, (TD, PI_j, -), P_i)$, where $PI_i > PI_j$, or

(b) $\diamond$ received$(-, (TD, PI_j, -), P_k) \ \forall P_k \in P - \{P_j\} \ldots$ Rules B2, R2.
$\diamond$ sent$(P_k, (RT, PI_j), PAR_k) \ \forall P_k \in P - \{P_j\} \ldots$ Rules B2, R2 and B3, R3.
$\diamond$ received$(P_l, (RT, PI_j), P_j) \ \forall P_l \in NB_j$.
$\diamond$ $P_j.TD$.

□

Lemma 4 *If* $(passive(P_j) \bigwedge DTC_t)$ *where* $t = passive\_time(P_j, PI_j)$*, then* $\exists\, P_i$ *s.t.* $(quiescent\_time(PI_i) > passive\_time(P_j, PI_j)) \bigwedge (PI_i > PI_l\ \forall P_l \in P - \{P_i\})$.

Proof From lemma 3, either

$\diamond$ $P_j.TD$, which proves the lemma, or

$\diamond$ received$(-, (TD, PI_j, -), P_i)$, where $PI_i > PI_j$.

Now received$(-, (TD, PI_j, -), P_i) \Rightarrow \exists\, TD_path(P_j, P_i)$. Let this path be $P_j - \ldots - P_k - P_i$. Hence $PI_k < PI_j < PI_i$ before P_k receives TD message with id PI_j. There are two cases :

(a) P_i has not propagated any other TD after initiating its own TD messages :
Let quiescent(PI_i).
Since $P_k \in NB_i$, $\diamond$ received$(P_i, (TD, PI_i, -), P_k)$.
Hence $PI_k.Seq = PI_i.Seq \ldots$ Rule B2.
However, this contradicts the condition $PI_k < PI_j < PI_i$ before P_k receives P_j's TD message.
Hence, $\neg$received$(P_i, (TD, PI_i, -), P_k)$. Hence $\neg$quiescent(PI_i).

(b) P_i has propagated the TD sent to it by some process P_l :
Let quiescent(PI_l). This gives rise to two cases :

  (i) $P_k \in NB_l$. Analogous to case (a), this leads to a contradiction. Hence $\neg$quiescent(PI_l).

  (ii) $P_k \notin NB_l$. Now, $P_k \notin BS$ of the TD message sent by P_l to P_i. Hence,
  $\diamond$ received$(P_i, (TD, PI_l, -), P_k)$.
  Analogous to case (a), this implies $\neg$quiescent(PI_l).

□

Theorem 2 (Liveness) $DTC \leadsto P_x.TD$*, for some* $P_x \in P$*.*

Proof Follows directly from lemmas 4 and 3.
□

6 Elimination of acknowledgements to basic messages

Termination of a distributed computation implies that

(a) all processes participating in the computation are passive, and

(b) no basic messages are in transit.

In the main algorithm presented in section 3, basic messages have acknowledgements, and a process can become passive (or be killed) only after receiving acknowledgements for all basic messages sent by it. Thus, receipt of the RT messages by the initiator of a termination wave satisfies conditions (a) and (b) simultaneously.

Communication efficiency can be increased if acknowledgements to basic messages are removed (already control messages do not have acknowledgements). However, provision will have to be made to satisfy condition (b), since a process could become passive (or be killed) even while some basic message(s) sent by it are in transit. This can be achieved within the framework of the current algorithm by implementing the following simple principle :

When a process P_i *sends a TD message to a process* P_j*,* P_j *will act upon it (possibly leading to an RT reply) only if*

(i) no basic messages between P_i and P_j are in transit, and
(ii) no basic messages sent to P_j by a dead process are in transit.

To ensure safety of termination detection, it is now necessary that TD messages should traverse *all* edges in the system graph. We achieve this by abandoning the notion of the broadcast set. Thus, a node sends TD messages to all its neighbours. Receipt of the required number of RT messages by the initiator of a TD wave now satisfies conditions (a) and (b) simultaneously.

6.1 Summary of the changes

The following changes are made in the main algorithm to incorporate the abolition of acknowledgements to basic messages :

1. The concept of *broadcast set* is removed from the algorithm. Hence, when process P_i propagates a TD received from P_j, it sends it to *all* neighbours excluding P_j.

2. For each process $P_j \in NB_i$, P_i maintains the following information :

   $msgsent_{i,j}$: number of basic messages sent by P_i to P_j, and
   $msgreceived_{i,j}$: number of basic messages received by P_i from P_j,

   where the subscript i is used merely for notational clarity. (Each process only needs single-dimensioned arrays for the counters.)

3. $msgsent_{i,j}$ is made a part of all control messages sent from process P_i to P_j. $msgreceived_{i,j}$ is also made a part of the TD and RT messages sent to P_j. The new formats of the various messages are as follows:

   TD message : $(sender, (TD, PI_k, msgsent_{sender,receiver}, msgreceived_{sender,receiver}), receiver)$
   where P_k is the process initiating the TD message.
   RT message : $(sender, (RT, PI_k, msgsent_{sender,receiver}, msgreceived_{sender,receiver}), receiver)$
   KP message : $(sender, (KP, PI_{sender}, msgsent_{sender,receiver}), receiver)$
   KM message : $(sender, (KM, PI_{sender}, S_{sender}, NB_{sender}, msgsent_{sender,receiver}), receiver)$
   TER message : $(sender, TER, receiver)$

4. $Deadset_j$ is maintained by each process P_j. For each process $P_d \in Deadset_j$, P_j maintains the following information :

   $deadsentmsg_{j,d}$: number of basic messages sent to P_j by P_d.

   This information is obtained from $msgsent_{sender,receiver}$ in the KM message from P_d.

5. On receiving a TD message, every *black* or *red* process P_i implements the principle described at the beginning of this section using the *msgsent* and *msgreceived* fields of the TD message and the counters $msgsent_{i,sender}$, $msgreceived_{i,sender}$ and $deadsentmsg_{i,d}$ from its own data base. Thus, the normal processing of a TD message is performed only if no basic messages are in transit between its sender and receiver, and between a dead process and receiver.

6. Before returning an RT message to PAR_k, a *black* process P_k ensures that no basic messages sent to it by a dead neighbour are still in transit. This is for any neighbour processes that may have been killed after the TD message was received by P_k.

7. If a TD message is not acted upon, its *black* or *red* recipient simply updates its sequence number and ignores the TD message.

Table 1 summarises the modifications to the actions performed by the processes. All other actions remain the same as in the main algorithm.

Table 1 : Actions of the processes in the extended algorithm
(*Note :* P_s is the message sender, P_r is the receiver.)

process colour	message received	conditions	actions
Any	Basic		Update $msgreceived_{r,s}$.
White	KP		Perform rule W2(d) without waiting.
White	KM		Update $Deadset_r$ and $deadsentmsg_{r,s}$.
Black	KM	(i) $msgsent_{s,r} = msgreceived_{r,s}$ (ii) Otherwise	Update $Deadset_r$, $deadsentmsg_{r,s}$ and perform rule B5. Update Seq_r, NB_r, $Deadset_r$ & $deadsentmsg_{r,s}$.
Black	TD	(i) $msgsent_{s,r} = msgreceived_{r,s} \bigwedge msgsent_{r,s} = msgreceived_{s,r} \bigwedge deadsentmsg_{r,d} = msgreceived_{r,d}\ \forall\ d \in Deadset_r$ (ii) Otherwise	Perform rule B2. Update Seq_r.
Black	RT	$deadsentmsg_{r,d} = msgreceived_{r,d}\ \forall\ d \in Deadset_r$	Perform rule B3.
Red	KM	Same as for a Black process.	Similar to a Black process (perform rule R5).
Red	TD	(i) Same as (i) for a black process. (ii) Otherwise	Perform rule R2. Update Seq_r (if necessary).
Red	RT	As in the Main Algorithm.	As in the Main Algorithm.

6.2 Proof of the extended algorithm

We use the following lemmas analogous to the lemmas of section 5 to prove the *safety* and *liveness* properties of the extended algorithm.

Lemma 5 $\neg DTC \Rightarrow \exists P_i \in P$ *such that* $(active(P_i) \bigvee (msgsent_{i,k} \neq msgreceived_{k,i} \;\forall\; P_k \in P \bigvee msgsent_{k,i} \neq msgreceived_{i,k} \;\forall P_k \in P \cup Deadset_i))$.

Proof Follows directly from definition of DTC.
□

Lemma 6 *If* $(passive(P_j) \bigwedge \neg DTC_t)$ *where* $t = quiescent\_time(PI_j)$, *then* $\exists P_i \in P - \{P_j\}$ *such that* $\Diamond\; received(-, (TD, PI_j, -), P_i) \bigwedge \neg sent(P_i, (RT, PI_j), -)$.

Proof The proof is analogous to lemma 2 except for the following :

2 (b) . . .

Let P_g be the sender of the basic message that activated P_k. Hence $P_k \in NB_g$. Now, Either (active(P_g) $\bigvee PI_g > PI_j$) when P_g receives the TD message with id PI_j, or P_g propagates the TD message to P_k, and

(i) active(P_k), or

(ii) $PI_k.Seq > PI_j.Seq$, if passive(P_k) (Rule W6).

□

Lemma 7 *If* $(passive(P_j) \bigwedge DTC_t)$ *where* $t = passive\_time(P_j, PI_j)$, *then either*

(a) $\Diamond\; received(-, (TD, PI_j, -), P_i)$, *where* $(PI_i > PI_j \bigvee \quad (msgsent_{i,k} \neq msgreceived_{k,i} \;\forall P_k \in P \bigvee msgsent_{k,i} \neq msgreceived_{i,k} \;\forall P_k \in P \cup Deadset_i))$, *or*

(b) $\Diamond\; P_j.TD$.

Proof The proof is analogous to lemma 3.
□

Lemma 8 *If* $(passive(P_j) \bigwedge DTC_t)$ *where* $t = passive\_time(P_j, PI_j)$, *then* $\exists\; P_i$ *s.t.* $(quiescent\_time(PI_i) > passive\_time(P_j, PI_j)) \bigwedge (PI_i > PI_l \;\forall P_l \in P - \{P_i\})$.

Proof The proof is analogous to lemma 4, except for the following simplification :

(b) P_i has propagated the TD sent to it by some process P_l :
Let quiescent(PI_l). Hence,
$\Diamond$ received($P_i, (TD, PI_l, -), P_k$).
Analogous to case (a), this implies $\neg$quiescent(PI_l).

□

Proofs of the Safety and Liveness theorems are identical to those for the main algorithm.

6.3 Some safety issues

A subtle restriction exists in the use of the extended algorithm which is not shared by the main algorithm. This arises from the fact that the neighbour relations in the system may not be symmetric at all times. Consider a process P_i wishing to send a message to some process $P_j \notin NB_i$. P_i adds P_j to NB_i while sending the message, whereas P_j adds P_i to NB_j only after receiving the message. What if P_i is killed in the meanwhile ? The main algorithm specifies that a process may be killed only after receiving acknowledgements for all basic messages sent by it. In the absence of acknowledgements, a safety problem could arise if P_i is killed before P_j receives its message and P_j replies to a TD message with an RT in the meanwhile. This problem does not arise if $P_i \in NB_j$ when P_j receives a TD message.

Two alternatives exist for handling such situations. A special kind of basic message may be introduced for communicating with a process which is not a neighbour. The recipient of such a message would be required to send an acknowledgement. The sender process would continue to maintain CTR for such messages, and a process can be killed only when its $CTR = 0$. Thus, the communication overheads of an acknowledgement are incurred only for such messages.

Alternatively, a process may inherit a neighbour set from its creator, and may augment it only in a manner guaranteed to preserve the symmetry of neighbour relations in the system (See Rule W3 in figure 2). This is not hard to implement since a process needs to know the identity of the receiving process in order to send a message to it. (The only exceptions are resource controllers or other server processes which are typically accessed through name servers in a distributed system. Such processes typically do not have to participate in termination detection !)

7 External processes joining an ongoing computation

In multi-file multi-client transactions new servers can spontaneously join an ongoing transaction dynamically [28]. To implement this, we require a new process P_n wanting to join the computation to send an *entry request (ER)* message to a process P_g belonging to the computation. P_g may be in one of the following states.

White : This implies that the computation has not yet terminated, so P_n can be allowed to join the computation straightaway.

Black : This implies that some process P_i has initiated termination detection and P_g, having propagated the TD message, is waiting for RT messages. In this case P_n is allowed to join in the computation and the TD message is sent to P_n also.

Red : This implies that P_g has sent an RT message and is waiting for a TER message. The computation would have terminated if the TER message is received by P_g. If, on the other hand, P_g receives a basic message or a TD message with higher PI, the computation has not yet terminated and P_n can be allowed to join in. Hence the entry request is recorded and an appropriate action is taken based on the next message received by P_g.

Dead : In this case P_n cannot join the computation.

The following assumptions have to be made regarding P_n in order to ensure the above.

1. P_n knows the id of at least one of the processes already in the computation.
2. P_n sends an *entry request (ER)* message to one of the processes, identifying itself as an external process wanting to join the computation.
3. P_n does not join the computation until it receives an *entry granted (EG)* message permitting it to do so.

4. Upon joining the computation, P_n follows the same termination detection algorithm.

The formats of the new messages are as follows :

Entry Request (ER) message : $(P_n,\ (ER),\ P_g)$ where P_n is the new process wanting to join the computation, and P_g is a process participating in the computation.

Entry Grant (EG) message : $(P_g,\ (EG, Seq_g, S_g, PMSET),\ P_n)$ where P_g is a process of the computation which grants entry to the new process P_n.

When granted entry, P_n sets its sequence number to Seq_g and its static to S_g, the static of the granting process. It also records the PMSET for its own use. Each *red* process P_i maintains an additional data structure,

$ERSET_i$: The set of id's of the processes that have requested entry since P_i last became red,

so that it can issue the EG messages appropriately.

The converse situation of a process P_i involved in the computation requesting an external process P_x to join in the computation is handled more easily. P_i can convey S_i, Seq_i and $PMSET$ to P_x, and P_x must agree to abide by the termination detection algorithm.

8 Concluding Remarks

The distributed termination detection algorithm developed in this paper has many advantages over the earlier work in the field. The algorithm is symmetric, and the communication is asynchronous with arbitrary but finite delays and possible non-FIFO delivery of messages. No acknowledgements are required for the basic or control messages exchanged by the processes. Thus, the only communication assumption is that a basic message sent to a non-existent process bounces back to its sender, with the flag NOTDEL set to TRUE. In the following, we comment on how this assumption is not particularly restrictive in practice.

The algorithm possesses many features aimed at reducing the overheads of control message traffic. Unlike [17], only the existing processes of a computation need to participate in the termination detection. The total order on processes defined by the process identification numbers (PI) is used to restrict the propagation of TD messages. No acknowledgements are required for the control messages. Further, during implementation we can either use the concept of the *broadcast set* as in the main algorithm, or eliminate acknowledgements to the basic messages as discussed in section 6. These possibilities offer further reduction in the communication overheads.

The algorithm permits free creation and destruction of processes during the computation. The latter is particularly important in a distributed data base environment where orphan elimination becomes necessary [13]. While it is customary to assume that a process is killed by another process of the same computation, this is not a restriction in the algorithm. Thus, the algorithm permits killing of a process by an external process, viz. the OS kernel. So long as the killed process follows the protocol of sending KM messages as in rules W2(c), B5, R5, the safety and liveness properties of the algorithm would continue to hold. The extension of section 7 permits new processes to join ongoing computations spontaneously. This has particular relevance to multi-file, multi-client transactions in data base systems [28].

Finally, some points concerning the implementation of the algorithm. While PMSET is a data structure accessible to all processes of the computation, it is free from the problems connected with the use of global variables. This is because this is a *read only* variable for all processes. A simple way to implement it would be to make it a process parameter specified at the time of process creation. All permanent processes of a computation would be created with the value of PMSET. When a new process is created, this value can be simply passed to it. The error flag

NOTDEL can also be implemented without requiring any special provisions. The flag is redundant if processes only communicate with permanent processes or known neighbours, a condition readily satisfied by nested transactions in a distributed data base environment. If a process P_j known to be a neighbour of P_i is killed by the time a message from P_i reaches it, the KM message from P_j would alert P_i to this fact (rules W2(c) and B5). Hence the NOTDEL flag can be dispensed with in most situations, leading to a further simplification.

References

[1] K. R. Apt, Correctness proofs of distributed termination algorithms, *ACM Transactions on Programming Languages and Systems,* vol. 8, no. 3, pp. 388-405, 1986.

[2] S. Chandrasekaran and S. Venkatesan, A message-optimal algorithm for distributed termination detection, *Journal of Parallel and Distributed Computing*, vol. 8, no. 3, pp. 245-252, 1990.

[3] K. M. Chandy and L. Lamport, Distributed snapshots : determining global states of distributed systems, *ACM Transactions on Computer Systems*, vol. 3, pp. 63-75, 1985.

[4] K. M. Chandy and J. Misra, *Parallel program design : a foundation*, Addison Wesley, 1988.

[5] S. Cohen and D. Lehmann, Dynamic systems and their distributed termination, *Proceedings of the first Annual ACM Symp. on Principles of distributed computing*, Ottawa, 1982, pp. 29-33.

[6] D. M. Dhamdhere, E. K. K. Reddy and S. R. Iyer, Distributed termination detection for dynamic systems, *TR-081-92*, IIT Bombay, (1992).

[7] E. W. Dijkstra and C. S. Scholten, Termination detection for distributed computations, *Information Processing Letters*, vol. 11, no. 1, pp. 1-4, 1980.

[8] E. W. Dijkstra, W. H. J. Feijen and A. J. M. van Gasteren, Derivation of a termination algorithm for distributed computations, *Information Processing Letters*, vol. 16, pp. 217-219, 1983.

[9] O. Eriksen, A termination detection protocol and its formal verification, *Journal of Parallel and Distributed computing*, vol. 5, pp. 82-91, 1988.

[10] N. Francez, Distributed termination, *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 42-55, 1980.

[11] N. Francez and M. Rodeh, Achieving distributed termination without freezing, *IEEE Trans. on Software Engg.*, vol. 8, no. 3, pp. 287-292, 1982.

[12] S. Haldar and D. K. Subramanian, A fully distributed termination detection algorithm in an arbitrary network, *Technical Report, IISc CSA-89-14*, Indian Institute of Science, Bangalore, 1989.

[13] M. P. Herlihy and M. S. McKendry, Time-Stamp based orphan elimination, *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 825-831, 1989.

[14] S. T. Huang, A fully distributed termination detection scheme, *Information Processing Letters*, vol. 29, no. 1, pp. 13-18, 1988.

[15] S. T. Huang, Termination detection by using distributed snapshots, *Information Processing Letters*, vol. 32, pp. 113-119, 1989.

[16] D. Kumar, A class of termination detection algorithms for distributed computations, *Maheshwari, N. (ed), Lecture Notes in Computer Science, no. 206*, Springer-Verlag, pp. 73-100.

[17] T. H. Lai, Termination detection for dynamic distributed systems with non-first-in-first-out communication, *Journal of Parallel and Distributed computing*, vol. 3, pp. 577-599, 1986.

[18] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[19] I. Lavellee and G. Roucairol, A fully distributed spanning tree algorithm, *Information Processing Letters*, vol. 23, pp. 55-62, 1986.

[20] F. Mattern, Algorithms for distributed termination detection, *Distributed Computing*, vol. 2, pp. 167-175, 1987.

[21] F. Mattern, An efficient distributed termination test, *Information Processing Letters*, vol. 31, pp. 203-208, 1989.

[22] J. Misra and K. M. Chandy, Termination detection of diffusing computations in communicating sequential processes, *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1, pp. 37-43, 1982.

[23] S. Owicki and L. Lamport, Proving liveness properties of concurrent programs, *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 455-495, 1982.

[24] S. P. Rana, A distributed solution to the distributed termination problem, *Information Processing Letters*, vol. 17, no. 1, pp. 43-46, 1983.

[25] M. Raynal, Distributed algorithms and protocols, *John Wiley and Sons*, 1988.

[26] J. L. Richier, Distributed termination in CSP – symmetric solutions with minimal storage, *Mehlhorn, K.(ed), Lecture Notes in Computer Science, no. 182*, pp. 267-278.

[27] S. Ronn and H. Saikkonen, Distributed termination detection with counters, *Information Processing Letters*, vol. 34, pp. 223-227, 1990.

[28] L. Svobodova, File-servers for Network-based distributed systems, *ACM Computing Surveys*, vol. 16, no. 4, pp. 353-396, 1984.

[29] R. W. Topor, Termination solution for distributed computing, *Information Processing Letters*, vol. 18, no. 1, pp. 33-36, 1984.

[30] M. C. van Wezel and G. Tel, An assertional proof of Rana's algorithm, *Information Processing Letters*, vol. 49, pp. 227-233, 1994.