An Optimal Termination Detection Algorithm for Rings

Murat Demirbas *Anish Arora* [1]

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA

Abstract

This paper presents an optimal termination detection algorithm for rings. The algorithm is based on Safra's and Dijkstra et al.'s [1] algorithm. It optimizes the time constant of the Dijkstra & Safra algorithm from $2N - 3N$ to $0 - N$, where N denotes the number of processes in the ring. A proof of correctness of the algorithm is also presented.

Keywords : Termination detection, Distributed computing, Algorithms.

[1] Email: {demirbas, anish}@cis.ohio-state.edu; Tel: +1-614-292-1836 ; Fax: +1-614-292-2911 ;
Web: http://www.cis.ohio-state.edu/{~demirbas,~anish};
Arora is currently on sabbatical leave at Microsoft Research. This work was partially sponsored by NSA Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and a grant from Microsoft Research.

1 Introduction

Distributed systems are inherently nondeterministic due to the asynchronous execution of the parallel processes and the asynchronous nature of the communication channels. The termination detection problem in distributed systems is not a trivial task because of this nondeterminism.

The termination detection problem is as follows. A set of processes communicate via message passing on interprocess channels. Execution is asynchronous, i.e., every process executes at its own speed and messages in the channels are subject to arbitrary but finite transmission delays. Every process is either "active" or "passive". An active process can both send and receive messages, whereas a passive process can only receive messages. An active process can become passive spontaneously, whereas a passive process can become active only upon reception of a message. The system is *terminated* if all processes are passive and no message is in transit. The problem is to detect termination as and when the system terminated.

Related Work. Since termination is a stable property, one can use Chandy and Lamport's snapshot algorithm [2] to detect termination in distributed systems. However, having a time complexity of $O(N^2)$ this algorithm is far from being efficient. Also using this method it is not clear how frequently a snapshot should be taken. Taking snapshots very frequently leads to wasting the CPU time, whereas taking snapshots infrequently leads to late detection of termination.

Safra and Dijkstra et al. suggested an efficient token-based algorithm for termination detection on rings[1]. In this algorithm, the token and each process maintains an integer and a bit. The algorithm has a time complexity of $O(N)$ and detects the termination in $2N - 3N$ time. However, this is not an optimal algorithm because it is possible to detect termination at worst within N time, where N is the number of processes in the ring.

Chandy[3] proposed using a detector process to solve the termination detection problem. According to his solution, whenever a process becomes idle, it sends a message to the detector process reporting how many messages it has received and how many it has sent on each channel. The detector uses this information to detect termination by detecting whether all the channels are empty at anytime. Sivilotti[4] improved the above solution so that, when a process becomes idle, the message it sends to the detector contains the number of messages it has received in total and the number of messages it has sent to each process. These algorithms succeed in detecting the termination as and when it occurs, however, they require a storage of upto N^2 ($2 * N^2$ in [3]) integers for the detector process and N ($2 * N$ in [3]) integers for each process and message. Also, the detector constitutes a single point of failure, and is likely to be swamped with extra work.

Mitchell and Garg[5] suggested a token based algorithm which is derived from their "general predicate detector". This algorithm basically uses the same idea as Chandy's algorithm and also has a space complexity of $O(N^2)$ integers.

This paper suggests an optimal termination detection algorithm for rings. The algorithm is based on Safra's and Dijkstra et al.'s [1] algorithm. It optimizes the constant of the Dijkstra & Safra algorithm from $2N - 3N$ to $0 - N$, where N denotes the number of processes. In this algorithm, each process maintains an integer and $N + 2$ bits, and the token stores N integers and $N + 1$ bits. Also, each message sent is piggybacked by the *process_id* (lgN bits) of the sender process.

The rest of the paper is organized as follows. First, in Section 2, the Dijkstra & Safra algorithm for termination detection in rings is recalled. Then, in Section 3, two enhancements are suggested for optimizing the Dijkstra & Safra algorithm, and the optimal algorithm is built based on these two enhancements. A proof of correctness for the algorithm is given in Section 4. Finally, Section 5 gives the concluding remarks.

2 The Dijkstra & Safra Algorithm for Termination Detection in Rings

The Dijkstra & Safra algorithm can be described as follows. Each process j maintains a count $c.j$, whose value is the number of messages sent by j minus the number of messages received by j. One process is designated as the initiator of termination detection. The initiator obtains a snapshot of the system where every process is passive by sending a token to the ring and gathering the sum of the count values of every process when that process is passive. Note that the sum of the count values of every process is equal to the number of messages in transit. If this snapshot is consistent and the number of messages in transit in this snapshot is zero then termination is detected. If not, the initiator repeats the snapshot collection.

The snapshot obtained by the initiator is *inconsistent* if the receipt of some message is counted but the corresponding send of that message is not. Consider the following scenario where the token has visited process j and has recorded $c.j$ but has not yet reached to process k. j becomes active again by receiving a message from k which was active at that moment. Then j sends a message ($msg.j.k$) to k, and after receiving this message k becomes passive. When the token reaches to k it will record $msg.j.k$ as received although it had not recorded $msg.j.k$ as sent. This is an inconsistent snapshot. An inconsistent snapshot may cause a false termination detection. If the sum of $c.j$'s equals to 0, the initiator concludes termination occurred although some processes (e.g. process j in the above scenario) are still active.

In the Dijkstra & Safra algorithm, in order to detect the inconsistency of a snapshot, a process sets its color to black on receiving a message. When the token meets a black process the token is marked as black. The initiator will consider the snapshot as inconsistent if the token is black and it will repeat the snapshot.

When N processes are arranged in a ring, process N can initiate the snapshot by sending the token to process 1. Process j can propagate the token around the ring by sending the token to process $j+1$. These are the actions of the superposed computation so they are available irrespective of the underlying computation, i.e., being passive does not prevent a process from participating in the superposed computation.

2.1 Program of process j in the Dijkstra & Safra algorithm:

Every process j maintains three variables: $c.j$, $color.j$, and $idle.j$. The value of $c.j$ is equal to the number of messages sent by j minus the number of messages received by j. $Color.j$ denotes the color of j, and can be either black or white. $idle.j$ is true iff j becomes passive. Also, the token maintains two variables: q and $token\_color$. The variable q is used for maintaining the sum of the counter values, $c.j$, of all processes it has visited after it has been (re)transmitted from the initiator, process N. The $token\_color$ denotes the color of the token and can be either black or white.

Initially, the *token_color* is white and the token sum, q, is 0. The *color* of every process is initialized to white (i.e., $\forall j :: color.j := white$), and the counter values of all processes are set to 0 (i.e., $\forall j :: c.j := 0$). Thus, the program of the Dijkstra & Safra algorithm for process j is as follows :

Dijkstra & Safra algorithm

{ `send message` } $\longrightarrow$
  $c.j := c.j + 1$;
  Send message

{ `receive message` } $\longrightarrow$
  $c.j := c.j - 1$;
  $color.j := black$;
  Receive message

{ `propagate token`, $j \neq N \ \wedge$ token is at $j \ \wedge \ idle.j$} $\longrightarrow$
  $q := q + c.j$;
  if $color.j = black$
    then $token\_color := black$;
  $color.j := white$;
  Send token to $j + 1$

{ `retransmit token`, $j = N \ \wedge$ token is at $j \ \wedge \ idle.j \ \wedge$
$\neg(q + c.j = 0 \ \wedge \ token\_color = white \ \wedge \ color.j = white)$} $\longrightarrow$
  $q := 0$;
  $token\_color := white$;
  $color.j := white$;
  Send token to 1

In the *send message* action of the program, j sends a message and increments its counter $c.j$ by 1. This corresponds to recording the message as sent.

In the *receive message* action of the program, j receives a message and decrements its counter $c.j$ by 1 which corresponds to recording the message as received. j also marks itself as black.

The *propagate token* action is executed when the token is at a process j such that j is not the initiator process and is idle. This action increments q, the sum of the counters of processes that the token has visited up to now, by the counter of j, $c.j$. If the color of j is black then the token is blackened; this marks the snapshot as inconsistent and forces the initiator to repeat the snapshot again. The color of j is set to white again for the next circulation of the token. Finally, the token is passed to the next process in the ring.

The *retransmit token* action is enabled only if the token is at process N which is the initiator, and process N is idle and the snapshot is inconsistent. The snapshot is consistent if and only if : the sum of the counters of all the processes is 0 (no messages are in transit), the token is white (no black process was observed during the snapshot), and process N is also white.

If one of these conditions fails, node N retransmits the token in order to take a new snapshot as follows. First the sum of the count values of processes visited is reset to 0. Then the color of the token and process N is reset to white. Finally, the token is sent to process 1 to restart the snapshot.

Termination is detected iff none of the four actions is enabled. A formal proof of correctness for the Dijkstra & Safra algorithm can be found in [1].

3 An Optimal Algorithm for Termination Detection in Rings

In this section, we first suggest two enhancements for optimizing the Dijkstra & Safra algorithm. The first enhancement is to use enumeration bits, and the second one is to have multiple initiators. Each of these reduces the worst case time complexity of the algorithm to $2N$. In Section 3.3, we combine both these techniques to reduce the worst case complexity to N, and present our optimal algorithm.

3.1 First Enhancement (Enumeration bits)

The Dijkstra & Safra algorithm has the overhead of blackening every process that receives a message. However, it is not the case that every message reception violates the consistency of the snapshot. In Figure 1, the token has visited processes 1, 2, and 3 but not reached to process 4 or 5. A message sent from 2 to 5, i.e. m_1, clearly violates the consistency of the snapshot: *the receiving of* m_1 *will be recorded even though the sending of* m_1 *was not recorded.* By the same token, m_2 also violates the consistency of the snapshot. However, a message sent from 5 to 4, i.e. m_4, does not violate the consistency, since the token has not reached processes 4 and 5 yet. When the token reaches 4, the receipt of the message will be recorded and when it reaches to 5 the corresponding sending of the message will also be recorded. By analogy, it follows that m_3 also does not violate the consistency of the snapshot. Note also that m_5 (respectively m_6) does not violate the consistency of the snapshot, since the send and receive of m_5 (resp. m_6) will be recorded when the token visits processes 2 and 3 in the next cycle. The messages m_7 and m_8 do not violate the consistency of the snapshot: the sending of m_7 and m_8 will be recorded when the token visits 4 and 5, and the reception for both messages will be recorded in the next cycle of the token. Therefore, the reception of messages $m_3 \dots m_8$ should not blacken the receiving process. However, the Dijkstra & Safra algorithm blindly blackens every process that receives a message. Due to this fact, at least one more cycle of the token is required to detect termination in the Dijkstra & Safra algorithm.

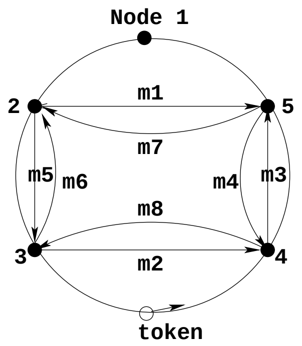

Figure 1:

We overcome this inefficiency in the Dijkstra & Safra algorithm by using *enumeration.* Every process has an enumeration bit; initially this enumeration bit is 0 for every process. The token will inverse the enumeration bits of the processes it visits. At the first turn the processes visited by the token will have enumeration 1, whereas the processes which are not yet visited will have enumeration 0. As seen from Figure 2 the enumeration bits separate the processes visited by the token from the ones which have not been visited yet. At the end of the first turn all the enumeration bits are set to 1. On the second turn the processes visited by the token will have enumeration 0 and the ones which are not yet visited will have enumeration 1.

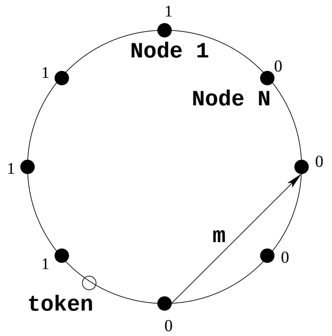

Figure 2:

The messages that violate consistency are those sent by processes that have already been visited by the token to the processes that have not been visited yet; the receiving of these messages will be recorded even though their sending was not recorded. We can employ the enumeration bits for speeding up the termination detection as follows. Any process that is sending a message piggybacks its *enumeration bit* and its *process id* to the message. *A process j receiving a message blackens itself if and only if the enumeration bit of j is not the same as the enumeration bit stored in the message and j's process id is greater than the process id attached to the message.*

For the purposes of this algorithm, an enumeration bit is sufficient and there is no need for sequence numbers. Suppose that a message proceeds very slowly and reaches its destination after 1 or more cycles of the token. In the odd cycles of the token, as seen in Figure 3.b, the late arrival of the message does not violate the consistency because the token has already recorded its send, and its receipt will be recorded in the next cycle. Since $q + c.N = 1$ in Figure 3.b, another turn is required. Note that the even cycles of the token while a message was in transit do not create any difference from the original scheme.

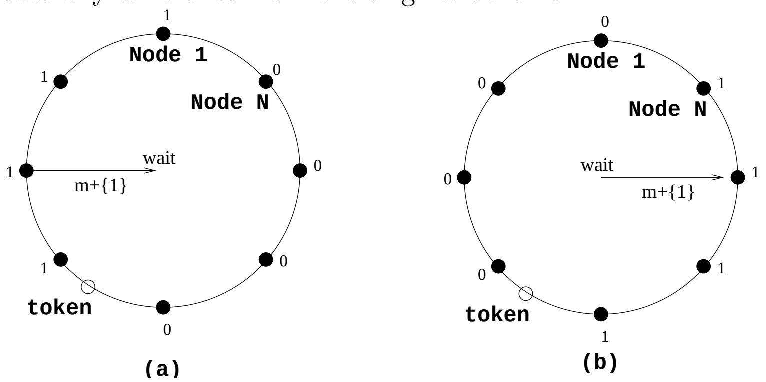

Figure 3:

Some optimization is possible by this technique, but in some cases the termination cannot be detected in $0 - N$ time. In Figure 4, process 2 sends a message to process 7 and 7 blackens itself. Suppose that termination occurs at this moment in the first cycle. The Dijkstra & Safra algorithm with the enhancement suggested in this section can detect the termination only at the end of the second cycle.

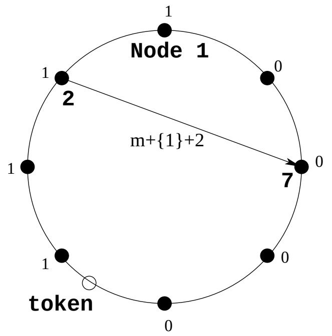

Figure 4:

3.2 Second Enhancement (Multiple initiators)

Another inadequacy of the Dijkstra & Safra algorithm is that it has a fixed initiator, namely process N. The algorithm uses a scalar, q, to hold the sum of $c.j$ values of the processes that the token has visited after it has been (re)transmitted by process N. A vector can be employed to maintain the sums with respect to different initiators, $[q.1, q.2, \ldots, q.N]$. Thus, we have N initiators.

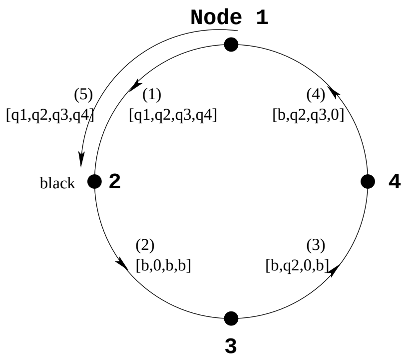

Figure 5:

In figure 5, process 2 becomes black just before the token reaches that node. When the token reaches process 2, since 2 is black, it marks all the other sums as black, and sets $q2$ to 0 to restart the snapshot. The other processes will also retransmit their corresponding $q.j$'s because they are marked as black. They also propagate the token and update $q.j$'s in the same way as in the Dijkstra & Safra algorithm. When the token reaches to process 2 again, if $q.1$ is not marked as black and $q.2 + c.2 = 0$ and process 2 is idle and not black, then termination is detected.

This technique also achieves some optimization, but still $2N$ time is required in some cases as seen in Figure 6.

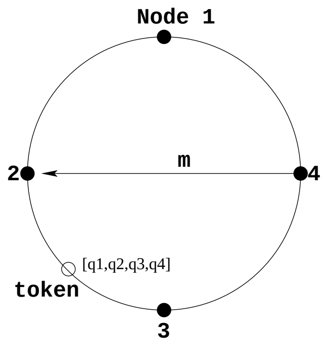

Figure 6:

3.3 The Optimal Algorithm

The second enhancement has an inadequacy; a black process marks all the counts in the token (i.e., $\forall j :: q.j$) as black. This leads to at least another circulation of the token before the termination is detected.

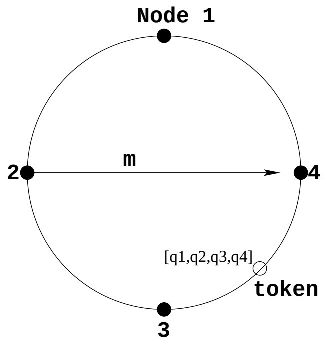

Figure 7:

In the above figure, process 2 becomes active by receiving a message from process 4, then it sends a message, m, to 4 and blackens 4. This message sending clearly violates the consistencies of both $q.1$ and $q.4$ since they have not recorded the sending of m. However, m does not violate the consistencies of $q.2$ or $q.3$ because $q.2$ and $q.3$ will record the reception of m when the token reaches 4, and they will also record the send of m when the token reaches 2. Thus, termination can be detected in one turn if we do not blacken $q.2$ and $q.3$ at process 4.

This goal can be achieved by merging the previous two enhancements and adding some extra information to the processes. In the final algorithm: The token stores a vector to hold the sum of the counters of processes, $[q.1, q.2, \ldots, q.N]$ as it was the case in the second enhancement. Furthermore, every node has an enumeration bit as it was the case in the first enhancement. Since, we are now dealing with multiple starting points, we use a vector instead of a scalar to represent the color of a process. Every process maintains a color vector which is used to keep track of the color of that process with respect to multiple initiators.

The program is similar to Dijkstra & Safra program. However, since we have multiple initiators instead of a single initiator, every "propagate action" is also a "retransmit action". Thus, those two actions are combined into one action. Also, when a process j receives a message m, we use j's enumeration bit (i.e., $enum.j$) and the enumeration bit in m (i.e., $enum.m$) and the process id attached to m (i.e., $sender\_id.m$) to decide whether or not to blacken j's copy of the color of a process l (i.e., $color.j.l$).

Initially, the token color is $white$ and the token sums are 0 for all processes (i.e. $\forall j :: token\_color.j := white \ \wedge \ q.j := 0$). The $color$ vector at every process is initialized to all $white$ (i.e. $\forall j,k :: color.j.k := white$), and the counter values of all processes are set to 0 (i.e., $\forall j :: c.j := 0$). Every process j initializes its enumeration bit to be "0" (i.e. $\forall j :: enum.j := 0$). Initially, the token is sent to process 1. The program for process j is as follows :

Termination Detection in Rings

$black = 1; white = 0;$

{ `send message` } $\longrightarrow$ $c.j := c.j + 1;$
Send $< message, enum.j, j >$

{ `receive message` m } $\longrightarrow$
$c.j := c.j - 1;$
if ($enum.j \neq enum.m \ \wedge \ sender\_id.m < j$)
then $(\forall k : j \leq k \leq N \ \wedge \ 1 \leq k < sender\_id.m : color.j.k := black)$;
Receive m

{ `propagate token`, token is at $j \ \wedge \ idle.j \ \wedge$
$\neg(q.j + c.j = 0 \ \wedge \ token\_color.j = white \ \wedge \ color.j.j = white$) } $\longrightarrow$
($\forall k$:: if $color.j.k = black$
then $token\_color.k := black$);
($\forall k$: $token\_color.k \neq black$: $q.k := q.k + c.j$);
$q.j := 0;$
$token\_color.j := white;$
($\forall k$:: $color.j.k := white$);
$enum.j := \neg(enum.j);$
Send token to $(j \ mod \ N) + 1$

The send message action of this algorithm is very similar to the Dijkstra & Safra algorithm, but the enumeration bit and process id of j are also sent with the message.

The receive message action updates the $color$ vector of that process. $color.j$ denotes the color of j with respect to multiple initiators. Whenever j receives a message m from l such that the enumeration bits of j and m differ and j's process id is greater than that of l, then m violates the consistency of the snapshot for those processes that are in between j and l in the direction of token propagation. Thus, j sets $color.j.k$ to $black$ for any process k such that $j \leq k \leq N \ \wedge \ 1 \leq k < l$.

The guard of the propagate token action is very similar to that of the "reinitialize token" action of the Dijkstra & Safra algorithm. This action is enabled only if the token is at j, j is idle, and the snapshot is inconsistent. The snapshot is consistent only if the token color for j is not black, j is not a black node, and $q.j + c.j = 0$. In this action, we blacken the $token\_color.k$ for any process k for which j is black (i.e., $color.j.k = black$), and also add the counter of j, $c.j$, to $q.k$ for the remaining white k's. Then, the token sum for j (i.e. $q.j$) is set to 0 and the token color for j (i.e. $token\_color.j$) is set to white. Finally, the color of j with respect to all processes (i.e. $\forall k$:: $color.j.k$) is set to white, the enumeration bit is inversed, and the token is passed to the next process.

4 Proof of Correctness for the Algorithm

In a detection problem, let X be the "detection predicate" and W be the "witness predicate". W detects X iff :

- $W \Rightarrow X$
- X eventually leads to W

Let I denote the invariant of the optimal algorithm, X be the "termination predicate" and W, the "witness predicate". The correctness of the terminaton detection program can be proven by proving:

- $(I \wedge W) \Rightarrow X$
- $(I \wedge X)$ eventually leads to W

Detection Predicate:
$X = (\forall j :: idle.j) \wedge$ (number of mesgs sent - number of mesgs received = 0)
Invariant:
$I = ((Sum j :: c.j) =$ number of mesgs sent $-$ number of mesgs received)
$\quad \wedge (\forall initiator :: (Q.initiator \vee R.initiator \vee S.initiator \vee T.initiator))$
$Q.initiator = (\forall j$: j is in the visited region with respect to the $initiator$: $idle.j)$
$\quad \wedge\ q.initiator = (Sum j$: j is in the visited region wrt. the $initiator$: $c.j)$
$R.initiator = q.initiator + (Sum j : j$ is in the unvisited region wrt. the $initiator$: $c.j) > 0$
$S.initiator = (\exists j : j$ is in the unvisited region wrt. the $initiator$: $color.j.initiator = black)$
$T.initiator = token\_color.initiator$ is $black$
Witness Predicate:
$W = (\exists j ::$ (token is at $j) \wedge (idle.j) \wedge (color.j.j = white) \wedge (c.j + q.j = 0) \wedge (token\_color.j = white))$

$Q.initiator$ implies that no message has violated the snapshot with respect to the $initiator$; all the processes that are in the visited region were idle and $q.initiator$ took their sum correctly. This optimistic predicate is not preserved by the receive action of the processes which are in the visited region with respect to the $initiator$.

When $Q.initiator$ is violated $R.initiator$ becomes true. $R.initiator$ means that the processes in the unvisited region had sent messages to the visited region so that the overall sum, $q.initiator$, would be greater than 0. However, $R.initiator$ is not preserved by the receive action of the processes that are in the unvisited region with respect to the $initiator$.

In order to detect this violation we use the *color* vector; $S.initiator$ becomes true and there exists a process j in the unvisited region such that $color.j.initiator$ is *black*.

$S.initiator$ can also be falsified since the token whitens the nodes it visits. However, in this case predicate $T.initiator$ is truthified. That is, $token\_color.initiator$ becomes *black*.

Proof of $(I \wedge W) \Rightarrow X$

Let j be the (unique) process where the token is residing. Since W holds observe that 1)token is at j, 2)$idle.j$, 3)$color.j.j = white$, 4)$c.j + q.j = 0$, and 5)$token\_color.j = white$.

Also, note the following: $(1 \wedge 3) \Rightarrow \neg S.j$, $(1 \wedge 4) \Rightarrow \neg R.j$, $5 \Rightarrow \neg T.j$.

In order for the invariant, I, to be true, $Q.j$ should hold, because the other three predicates (i.e. $S.j, R.j$, and $T.j$) are falsified. The invariant also states that $I_1)((Sumj :: c.j) = \#$ mesgs sent $- \#$ mesgs received).

Finally, $(1 \wedge 2 \wedge Q.j \wedge 4 \wedge I_1) \Rightarrow X$. □

Proof of $(I \wedge X)$ leads to W within one circulation of the token

Given that $(I \wedge X)$ holds we have $(\forall j :: idle.j)$ and $((Sumj :: c.j) = 0)$. Also, the only enabled action is the *propagate token* action, since termination has occurred. Let j be the process where the token is residing when termination occurrs. Observe from the program that $(q.j = 0)$, $(color.j.j = white)$, and $(token\_color.j = white)$ when the token is propagated from j to $(j \ mod \ N) + 1$. The propagate action at any process other than j cannot modify $color.j.j$.

We claim that when termination occurs $(\forall k :: color.k.j = white)$. Assume for a contradiction that $(\exists k :: color.k.j = black)$. There are three cases to consider, 1) $k < j$, 2) $k = j$, and 3) $k > j$.

1. For case 1, if $color.k.j$ is set to *black* before the token visited k, then $color.k.j$ is set to *white* when the token reaches to k. Else if the token has visited k and has not advanced through j, then $color.k.j$ cannot be blackened by a message from i s.t. $1 \leq i \leq k$, since $enum.i = enum.k$. Also note from the *receive* action that k cannot be blackened by a message from l where $k \leq l$. In sum, $(\forall k : k < j : color.k.j = white)$ when termination occurs.

2. Case 2 is trivial; since the token is at j from the program actions $color.j.j = white$.

3. Case 3 is also not possible because the blackening at k is done for a process l iff $(k \leq l \leq N \ \vee \ 1 \leq l \leq sender\_id) \wedge (enum.sender\_id \neq enum.k)$. If $sender\_id \leq j$ then obviously $color.k.j$ is not blackened. If $sender\_id > j$ then $enum.sender\_id = enum.k$ since the token is at j. In sum, $(\forall k : k > j : color.k.j = white)$ when termination occurs.

Since $(\forall k :: color.k.j = white)$, and the only enabled action is the *propagate token* action, $token\_color.j$ cannot be blackened by any process k.

Finally, when the token reaches to j again $(c.j + q.j = (Sumj :: c.j) = 0)$ holds. Therefore, within one circulation of the token the witness predicate "$W=$ (token is at j) $\wedge$ $(idle.j)$ $\wedge$ $(color.j.j = white)$ $\wedge$ $(c.j + q.j = 0)$ $\wedge$ $(token\_color.j = white)$" is truthified at j. □

Note that, termination can be detected in $(N - d)$ time if at the moment of termination there exists an initiator i such that the token has visited d processes after it has been propagated from i and the snapshot to be taken by the token is consistent with respect to i.

5 Concluding Remarks

In this paper, we presented an optimal termination detection algorithm for rings. The algorithm was based on the Dijkstra & Safra algorithm and the two enhancements suggested in Section 3.

The algorithm is proved to detect termination in $0 - N$ time, whereas the Dijkstra & Safra algorithm can detect termination in $2N - 3N$ time. The improvement in the constant is due to the fact that this algorithm starts counting for every node and avoids unnecessary invalidations of the consistent snapshots, whereas the Dijkstra & Safra algorithm has only one starting point and marks some consistent snapshots as invalid.

References:

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

2. K.M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no.1, pp.63-75, 1985.

3. K.M. Chandy, "A theorem on termination of distributed systems," Technical Report TR-87-09, Department of Computer Sciences, University of Texas at Austin, March 1987.

4. P. Sivilotti, "A More Efficient Algorithm for Termination Detection," Unpublished research notes, February 1996.

5. J.R. Mitchell, V.K. Garg, "Deriving distributed algorithms from a general predicate detector," Proc. *The Nineteenth Intl. Computer Software and Applications Conference*, Dallas, Texas, August 1995, pp. 268 – 273.