1987

A Robust Distributed Termination Detection Algorithm

Niraj K. Sharma

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
87-726

A ROBUST DISTRIBUTED TERMINATION DETECTION ALGORITHM

Niraj K. Sharma
Bharat Bhargava

CSD-TR-726
December 1987

A ROBUST DISTRIBUTED TERMINATION DETECTION ALGORITHM

Niraj K. Sharma

and

Bharat Bhargava

Department of Computer Sciences
Purdue University, West Lafayette
IN 47907

ABSTRACT

When all the processes in a distributed program are idle, either a global termination condition exists or all the processes are deadlocked. A distributed algorithm to detect the global termination condition is presented. The algorithm is general and can be incorporated as an underlying mechanism with any distributed programming system. The proposed methodology is immune to process failures, interprocess communication link failures, and network partitioning. The technique used is based on a spanning tree formation and it uses only those interprocess communication links which are present in the application program. To support resiliency, the algorithm does not depend on some fixed topology and the spanning tree is formed dynamically to take care of topology changes occurring as a result of failures in the system.

1. INTRODUCTION

In a distributed system, termination is a property of the global state of a distributed program and the processes are aware only of their local states. To terminate a distributed program, all the processes should be idle. A process is said to be idle if it is not doing any computation and also it is not blocked for some event to occur. The problem is to design an algorithm that will be executed by all the processes to determine the termination condition without any centralized control.

The problem of detecting the distributed termination condition and distributed deadlock are different in the sense that the termination condition exists when all the processes are idle and the deadlock situation exists if all or only a subset of the processes are blocked such that there is a circular wait. There are many algorithms designed to detect the distributed deadlock condition [2, 3, 10]. The basic scheme to detect distributed deadlock is that if a process does not do any computation for some time then the process starts the execution of a deadlock detection algorithm looking for circular wait condition. Basically, these algorithms utilize the information about one process being dependent on some other process. Another difference between these two problems is in context of robustness. Consider a distributed application program which is designed in a way such that it is immune to process failures. If the application program is robust then the algorithms to detect termination and deadlock should also be robust. The event of process failure will effect these two algorithms differently. In deadlock detection algorithms, all the blocked processes waiting for a response from a failed process will abort any deadlock detection algorithm under execution. Where as in a termination detection algorithm, a failed process should be simply ignored and any termination detection algorithm under execution is not aborted.

There are many solutions suggested to solve the problem of distributed termination detection. The schemes suggested in [5, 9] are in context of CSP, a language nota-

tion proposed by Hoare [7]. One desirable property of these two schemes is that no new interprocess communication links (to be called links) are introduced. The links required to execute the application are used to detect the termination condition also. The solutions described in [1, 4] are more general. The algorithm given in [1] is applicable to any process model in which a process is able to know locally whether it is doing any computation or not, which means idle processes and blocked processes are treated alike. If this algorithm is applied to the process model suggested in CSP where it is impossible to differentiate between an idle state and a blocked state of a process locally, then it will be able to detect that none of the processes is doing computation. But it will not be able to detect whether the program has terminated or it is deadlocked. In case of the process model suggested by Hansen [6] called DP (Distributed Processes), it is possible for a process to determine locally whether it is idle or blocked. When a process is not doing any computation and if it is inside either the initial statement or any one of the global procedures then the process is blocked otherwise it is idle. With this process model, the algorithm suggested in [1] will be able to differentiate between the termination and the deadlock condition of a program by simply checking whether any one of the processes is blocked or not. The major drawback of this algorithm is that it creates new communication links in addition to the links utilized by the application. Some times the algorithms to assign processes on processors utilize the information about the link structure in the application program in order to minimize the message traffic among processors. For example, if two processes communicate quite frequently then they will be assigned to the same processor. Therefore, creation of additional links to detect termination condition will further complicate the task of assignment.

None of the schemes mentioned above will work with process failures, link failures or network partitioning. In this paper, we propose a scheme to detect distributed termination that has no centralized control, creates no additional communication links, and is immune to process failures, link failures and network partitioning. It detects the

condition when none of the processes is doing any computation. The proposed methodology is based on the formation of a spanning tree. In case of a general network structure, it is quite common to use a spanning tree as an underlying structure on whose edges all communication activities take place [8]. We will also show how the proposed scheme can be modified to differentiate between the termination and the deadlock conditions when it is applied to the DP process model. With the CSP process model, it will not be able to differentiate between the two.

2. DISTRIBUTED COMPUTATION MODEL

A distributed program consists of a collection of processes. Processes communicate only through messages. When a distributed program is started all the processes start doing computation concurrently. At the physical level, two processes might be assigned to two different processors such that there is no direct connection between the processors, however they can communicate through other processors. Therefore a link at the process level might consist of several interprocessor connections at the physical level. If the link AB between two processes A and B fails, that means the physical network is partitioned into two such that process A belongs to one partition and B to the other one. If the network is not partitioned then there will exist a direct or indirect path between any two processors.

We assume that a process can determine locally whether it is doing any computation or not. To fulfill this assumption, each process will be required to maintain a variable that is set before starting computation and reset before becoming idle or blocked.

A process in idle state can again start computation as a result of some communication initiated by some other process. If a process is executing the termination detection algorithm then it does not mean that the process can not be in idle/blocked state, because these states are determined with respect to the state of the process in the application pro-

gram. A process is said to be in **stable state** if it initiates no communication but is ready to communicate with other processes that would initiate such a communication or it is blocked. A process which is not in stable state, is said to be in **unstable state.** All the processes reach their stable states by means of some finite sequence of communication called **application communication** and no process knows that the other processes are in stable states. The termination detection algorithm will terminate the globally stable program by means of some additional communication called **extra communication.**

We make the following assumptions about messages.

- *In the absence of failures, messages do not get lost:* It can be implemented by explicit acknowledgements and retransmission of messages.

- *In case of process or link failures, messages might get lost*

- *Messages sent by a process to any other process are received in the sequence they were sent:* This requirement can be met by using sequence numbers. In the algorithm presented, this requirement makes sure that whenever a process expects two messages in response to a message they arrive in order. It is simple to remove this restriction and modify the algorithm to take care of the sequence numbers.

3. ROBUST ALGORITHM TO DETECT TERMINATION CONDITION

The need of a robust distributed termination detection algorithm arises only when the application program itself is immune to process failures, link failures, and network partioning. For example, consider an application program that consists of a process U and two printer processes P1 and P2. Process U is designed in such a way that if one of the printer processes fails or gets isolated because of link failures then the whole printing

operation is done through the other printer process. This application is immune to one printer process failure or its isolation.

A termination detection algorithm should be general in the sense that it should not depend upon the number of processes in the application program, structure of links among processes, and the knowledge of robustness of the application program with respect to certain process failures. The termination condition should be detected on the basis of the processes that have not failed and if the application program gets divided into isolated groups of processes because of network partitioning then each group should terminate separately.

The termination condition is detected by the following scheme. Each process attempts to form a spanning tree with itself as its root. Finally, only the process having the highest ID is able to make a spanning tree successfully such that rest of the processes are intermediate nodes or leaves and the edges are derived from the links existing in the application program. No new links are created. While forming trees, a process communicates with its children to find whether all the processes in their subtree are idle/blocked or not. Children in turn ask their children, and so on. When this message reaches at the leaves, a phase of messages starts towards the root. If the root finds all the answers to be affirmative, it starts a second round of messages that tells every process to terminate. Otherwise the root tries to make another tree. At a particular moment several trees might be under formation. The most recent tree with the highest ID process as its root has the highest priority. Termination occurs when all processes belong to the same tree and all of them are stable. A termination detection scheme which is immune to process and/or link failures is described in section 3.1. The correctness of the scheme is discussed in section 3.2. and a formal specification is given in the appendix.

3.1. Termination detection in the presence of failures

Each process executes the identical code to detect termination. For expressing

the algorithms we assume that the ID of the process executing the algorithm is A and B is the ID of some other process which is one of the neighbors of A with respect to the application communication channels.

The description of the messages to be used is as follows. The MAKE_TREE message is used to form a tree and it starts from the root travelling towards the leaves. This message has two integers that determine the priority of the tree. The first integer represents the ID of the root process and the second one represents the number of attempts made by the root process to make a tree. The most recent tree with the higher or equal root ID has more priority. Consider the case, when process B receives a MAKE_TREE message from process A. If B is already a node of the tree which A is trying to form then it informs A by sending a NOT_YOUR_CHILD message that it can not become its child. If B is a node of some other tree with higher ID root or same root ID but higher sequence number then B sends a ABORT message to A to tell that abort the attempt to make a tree because the formation of a higher priority tree is under progress. If none of these conditions are met then B agrees to become a child of A by sending a YOUR_CHILD message and tries to make the tree further. While forming a tree, if a process detects that the termination condition does not exist it sends a CANCEL message to its parent and cancels the formation of this tree. After this the root attempts to form another tree. Otherwise if a process based on its information determines that the termination condition exists, it sends a YES message to its parent.

The scheme presented here treats the events of link failure and process failure alike. When a link fails, we assume that the local operating systems inform the separated processes that they are unable to communicate. When a process fails, all its neighbors receive information from their respective local operating systems that they are unable to communicate with the failed process. In both the cases, the message received by processes is same. With this information, the termination detection algorithm running at a process will take care of both types of failures. To handle network partitioning, we use

a pessimistic approach. In this scheme, the termination condition for a disjoint group of processes exists if all of them are part of the same tree and they are stable. In case of network partitioning, different groups will make different trees and they will terminate separately.

In response to different events Process A executes different algorithms. When an event occurs process A might be stable or unstable. The different conceivable cases are considered below. A detailed description of these algorithms is given in the Appendix.

Algorithm 1. *When process A is unstable and initiates application communication with process B*:

Process A records the fact that it did application communication with B.

Algorithm 2. *When process A is in unstable state and receives MAKE_TREE message from B:*

The arrival of a MAKE_TREE message means a new tree is under formation. Depending on the priority of the new tree there are following three cases.

- If the new tree has lower priority than the tree of which process A is already a node, then send ABORT message to B to abort the formation of the new tree.
- If the new tree has higher priority, process A sends YOUR_CHILD message to inform process B that it is now a node of the new tree. Right now process A is unstable therefore termination condition can not exist. So, process A sets a boolean variable FORM_TREE and it does not forward the request to form the tree to other neighbors. When process A changes its state to the stable one, it will propagate the request further by executing the algorithm 12.
- If the priority of the new tree and the previous tree of which A is a node is same that means process A has already received this MAKE_TREE message via some other path, therefore A is already a node of the tree being formed. To inform B about it, A sends NOT_YOUR_CHILD message to B.

Algorithm 3. *When process A is stable and receives a MAKE_TREE message from B:*
Same as Algorithm 2 except that if new tree is to be formed then forward the MAKE-TREE message further to its neighbors to form the rest of the tree instead of postponing it. When responses from all the neighbors are received, A sends its response back to B. If process A is a leave of the tree then it will send a response back to process B by calling the procedure **respond_to_parent** and A will not forward the MAKE_TREE message.

Algorithm 4. *When process A is unstable and receives a NOT_YOUR_CHILD message from B:*
This message must have been sent in response to a MAKE_TREE message from A to B. If it is in context of the latest tree, then simply record that the response from process B has been received and set a variable PENDING_MESSAGES and postpone further processing of this message till A becomes stable. If the message is not in context of the latest tree of which A is a node then discard the message.

Algorithm 5. *When process A is stable and receives a NOT_YOUR_CHILD message from B:*
Same as in Algorithm 4. except that instead of postponing its processing, A calls the procedure **respond_to_parent** which will send a response to A's parent in case this was the last message from the neighbors A was waiting for.

Algorithm 6. *When process A is unstable or stable and receives a YOUR_CHILD message from B:*
If this message is in context of the latest tree of which A is a node then record the fact that B is a child node else discard the message.

Algorithm 7. *When process A is in unstable condition and receives a YES message from*

B:

This message has been sent by B in response to a MAKE_TREE message from A to inform that as far as B and all its children are concerned the termination condition exists. If this message is in context of the latest tree of which A is a node then it records the fact that YES message came from B and set the variable PENDING_MESSAGES to enable its processing later on because right now A is unstable else A discards the message.

Algorithm 8. *When process A is in stable condition and receives a YES message from B:*
Same as Algorithm 7. except that instead of postponing its processing, A calls the procedure **respond_to_parent.**

Algorithm 9. *When process A is unstable or stable and receives an ABORT message from process B:*
This message has been sent by B to abort the formation of a tree in response to a MAKE_TREE message from A. If A is still a node of this tree then pass this message to its parent and disconnect itself from the tree. If A is the root of this tree then simply disconnect itself.

Algorithm 10. *When process A is unstable and receives a CANCEL message from process B:*
This message has been sent by B in response to a MAKE_TREE message from A to cancel the formation of the tree because B has detected that the termination condition does not exist and the tree is to be formed again. If process A is still a node of this tree then record the fact that CANCEL message was received from B and to process this message further when A becomes stable, set PENDING_MESSAGES to true.

Algorithm 11. *When process A is stable and receives a CANCEL message from process*

B:

Same as Algorithm 10. except that instead of postponing its processing, A calls the procedure **respond_to_parent**.

Algorithm 12. *When process A changes its state from unstable to stable:*

When a process is unstable, one of the following two operations might get suspended. The first one is the operation of making a tree and the second one is acting upon the messages coming from children in response to a MAKE_TREE message sent before becoming unstable. If FORM_TREE is true then it means the first operation is suspended. To start it, process A sends MAKE_TREE messages to all the neighbors except its parent. If there is no neighbor except the parent then process A calls the procedure **respond_to_parent** to send a reply to its parent. If PENDING_MESSAGES is true then it means the second operation is suspended and a few messages are to be processed and an appropriate response is to be sent to the parent. To do both these tasks, process A calls the procedure **respond_to_parent.**

Algorithm 13. *When process A is unstable or stable and gets a message from the local operating system that A can not communicate with process B:*

If A is not able to communicate with B then it means that either process B has failed or the link AB has failed. It leads to the following three cases.

- B is the parent of A: It is possible that the network is partitioned into two sets such that B and the root of the tree are in one partition and A in the other one. If root is in the first partition then there is no process in the second partition to initiate the tree formation to detect the termination condition separately. In the second partition, at least A knows about this possibility. Therefore, A starts the formation of a tree with itself as its root. If A is unstable then it postpones it by making the variable FORM_TREE true. Similarly, there might be other neighbors of B that also start making trees of their own. In the end,

only the process having the highest ID among these processes will succeed. If the network has not got partitioned then the attempt of forming a new tree by all these processes will compete with the attempt by the original root that existed before the fault occurred. Again the process with the highest ID will win.

- B is a child of A: If the network is partitioned then A is in the partition in which the root exists. Therefore, A simply cancels the formation of this tree and the root (the root may be A itself) will try to make another attempt. A fresh attempt is required to take care of the disturbances created by the failure.
- A and B are not related: Under this case there may be two possibilities. The first one is that A is not a node of any tree. In this case, A starts forming a tree with itself as its root. Again, if A is unstable it will postpone it. The second case is when A is a node of some tree which does not have the edge AB. In this case, A takes no action because if A and B are not able to communicate, it will not effect the tree as it does not have the edge AB.

```
Procedure respond_to_parent;
begin
    if process A has received responses of the MAKE_TREE messages
      from all the neighbors or process A does not have any
      neighbor except its parent
      {response can be the message NOT_YOUR_CHILD or ABORT or a
      pair of YOUR_CHILD and YES messages or a pair of YOUR_CHILD
      and CANCEL. If the response is the ABORT message, control
      will never reach here, see algorithm 9.}
    then
      if A did not perform any application communication during
          the formation of the current tree and (responses from
          all children is YES message or A has no child)
```

then

All the processes in the subtree below process A are stable. If A is the root then the termination condition exists otherwise A sends YES message to its parent

else {termination condition does not exist}

if A is the root, start the formation of a new tree otherwise A is an intermediate node in the tree and it sends CANCEL message to its parent to abort the formation of the current tree

end.

3.2. Correctness proofs

Lemma 1. *At any instance, consider a set S of the processes that have sent YES message towards the root while forming a tree T (a tree is identified by its root and the sequence number). If any one of the processes belonging to S, namely X, is made unstable by a process Y which does not belong to S then process Y is a node of T or of some other higher priority tree.*

Proof. If Y has made X unstable then Y is a neighbor of X. Since X has sent the YES message to form T and Y has not that means at the time when X sent the YES message X was neither parent nor child of Y. According to Algorithm 2., it is possible only when Y is already a node of T. After some time, when Y makes X unstable, either Y is still a node of T or it might be a node of a tree having higher priority than T because only a higher priority tree can detach Y from T. □

Theorem 1. *When a process X has sent YES message to form a tree T and it is made unstable later, then the termination of the program can not occur.*

Proof. Let S be the set of the processes that have sent YES message while forming T. Since X has sent YES message therefore X is a member of S. Let the ID of the process that has made X unstable be Y. Y can either be a member of S or not. If Y is not in S then Y is a node of T or some other higher priority tree T' according to Lemma 1. Now if Y is not in S and is a node of T that means Y has not sent YES message as yet and in future it will send CANCEL message instead of YES message because it has talked to X. It means the program can not be terminated. Now consider the case when Y in not in S and is a node of T'. In this case, it is evident that X is not a node of T' therefore the program can not be terminated without consulting X. The only case left is if Y is in S. If Y is in S then it must have been made unstable by some other process which again might be a member of S and so on. Ultimately, there has to be a process outside S that started this wave and this case has already been covered above. □

Theorem 2. *Till a process is unstable, the program will not get terminated.*

Proof. A program can get terminated only when all the processes have sent YES message. While a process is unstable it can never send a YES message. But if it sent YES message before becoming unstable then according to Theorem 1. the program can never get terminated. It means that while a process is unstable the program can never get terminated. □

Theorem 3. *After the event of any number of process failures, the rest of the processes will be able to terminate themselves.*

Proof. Whenever a process X fails, different events take place depending upon what relationship X has with its neighbors. Let process Y be one of the neighbors of X. If y is the parent of X then Y sends CANCEL message towards root that means the tree will be

formed again. If Y is a child of X then Y starts making its own tree to take care of the possibility of network partitioning. Note that X might be the root. Consider the possibility when Y is neither a child nor a parent of X. It means that Y is not a node of any tree or the edge XY does not exist in any of the trees being formed now. In the former case, Y starts the formation of a new tree and in the latter one Y does nothing because the failure of X does not effect the formation of a tree as far as Y is concerned, however some other process which is either a child or a parent of X will take some action. In every possible situation, all the processes that have not failed will participate in the formation of some tree which means that they will get terminated when the termination condition occurs. □

Theorem 4. *After any number of link failures, all the processes will be able to achieve termination.*

Proof. If a link XY connecting process X and Y fails then it means the network has become partitioned into two disjoint portions. As far as processes X and Y are concerned, they both think that the other process has failed and the case of process failures is covered by Theorem 3. □

4. PERFORMANCE CONSIDERATIONS

In the termination detection schemes described above, all processes try to make separate trees in the beginning. In fact there is no need of doing so. When the processes are started any one of the processes can be given the responsibility of making a tree. Its advantage is that there will be less message traffic. It will not effect the robustness of the scheme, because at any time if the process responsible to make a tree fails its neighbors will start executing Algorithm 13. One of them will ultimately be successful in making a tree. We allowed all the processes to start making trees in the beginning just to treat all

of them alike.

If a process is unstable, it postpones the processing of the messages related to the formation of a tree till it becomes stable. This causes delay in forming a tree when any one of the process is unstable, which in turn reduces the message traffic.

Let N be the number of processes in an application program and L be the number of leaves in a tree T formed to detect the termination condition. In the absence of any failure, there will be N nodes and (N-1) edges in T if it gets formed completely. In an attempt to form a tree, the minimum number of messages exchanged will be equal to 2 when the first child of the root sends CANCEL message back to the root and the maximum number of messages exchanged will be 3(N-1) because to form one edge 3 messages are exchanged. In an attempt to form a tree when all the processes are stable, total number of messages exchanged to detect the termination condition will be 3(N-1). The time complexity is $O(\log_k N)$, where k is the average number of children of any node in T except the leaves and it is equal to (N-1)/(N-L). Exact amount of extra communication or its upper bound is difficult to estimate as it depends upon the relative speed of different processes, how many times various trees are formed, and how failures occur. In the scheme presented in [1] which is not robust and introduces new communication links, the total number of messages required to detect the termination condition when all the processes are stable will be (N-1) and the time complexity will be O(N). In the algorithms presented in [5] and [9], a spanning tree is made; therefore, their time complexity will also be $O(\log_k N)$ when all the process are stable.

5. DISCUSSION

The robust scheme presented in this paper is general and it can be applied to any process model to detect whether all the nonfaulty processes are doing any computation or not. If it is applied to DP, then by maintaining one more variable at each process to store

whether the process is blocked or not and then conveying this information to the root along with YES message will enable the root to differentiate between termination and deadlock conditions. It is possible because in DP a process can find out locally whether it is blocked or not. With CSP, it is not possible. The solutions for the process model of CSP are suggested in [5] and [9].

Our scheme can be modified to take care of the processes or links that become alive after being dead for some time. First we need to modify the NEIGHBOR variable in all the effected processes and then if these processes are nodes of some tree then abort that tree and start the next attempt. If a process is not part of any tree then do nothing. Meanwhile, if the termination wave has already been started then coming up of a process or link will not have any effect as the program termination is in progress.

REFERENCES

[1] R.K. Arora and N.K. Sharma," A methodology to solve distributed termination problem," Information Systems, vol. 8, no. 1, pp. 37-39, 1983.

[2] K.M. Chandy and J. Mishra," Distributed deadlock detection," ACM Trans. on Computer Syst., vol. 1, no. 2, pp. 144-156, May, 1983.

[3] I. Cidon, J.M. Jaffe, and M. Sidi,"Local distributed deadlock detection by cycle detection and clustering," IEEE Trans. on Soft. Eng., vol. SE-13, no. 1, pp. 3-14, Jan. 1987.

[4] E.W. Dijkstra, and C.S. Scholten,"Termination detection for diffusing computations," Inf. Process. Lett., vol. 11, no. 1, pp. 1-4, Aug. 1980.

[5] N. Francez,"Distributed termination," ACM Trans. on Program. Lang. and Syst., vol. 2, no. 1, pp. 42-55, Jan. 1980.

[6] P.B. Hansen,"Distributed processes: a concurrent programming concept," Comm. ACM, vol. 21, no. 11, pp. 934-941, Nov. 1978.

[7] C.A.R. Hoare,"Communicating Sequential Processes," Comm. ACM, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[8] E. Korach, S. Moran and S. Zaks, "The optimality of distributed constructions of minimum weight and degree restricted spanning trees in a complete network of processors" Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 277-286, August, 1985.

[9] J. Misra and K.M. Chandy,"Termination detection of diffusing computations in communicating sequential processes," ACM Trans. on Program. Lang. and Syst., vol. 4, no. 1, pp. 37-43, Jan. 1982.

[10] R. Obermarck,"Distributed deadlock detection algorithm," ACM Trans. on Database Systems, vol. 7, no. 2, pp. 187-208, June 1982.

APPENDIX

A detailed description of the algorithms is as follows. For expressing the algorithm we assume that the ID of the process executing the algorithm is A and B is the ID of some other process which is one of the neighbors of A with respect to the application communication channels. Different variables maintained by process A are:

FORM_TREE - It is of type boolean. When process A is in unstable state, it knows that the global termination condition can not exist. At this moment, if it receives a message from one of its neighbors to participate in the activity of a spanning tree formation it simply sets the FORM_TREE variable and does not propagate the wave of the tree formation further. This action will reduce the amount of extra communication. After some time, when A changes its state from stable to unstable, it checks the value of FORM_TREE. If it is true, then it becomes an intermediate node of the tree and propagates the request to make tree to its potential children. FORM_TREE is initialized to true to enable each process to start making a tree with itself as a root. In the end, the tree started by the process having the highest ID succeeds and the rest are aborted. If a root process fails at any time, some other process starts forming another tree.

CHILDREN - It is a set of process ID's that are process A's children. It is initialized to **nil**, because initially in the absence of a tree there is no child-parent relationship among adjacent processes.

PARENT - It is used to store the ID of the parent of process A and it is initialized to **nil**.

NEIGHBOR - It is a set of process ID's. It is initialized with the ID's of all the neighbors of process A with respect to the application program.

TREE_ROOT - It is used to store the ID of the root of the tree that process A is trying to form currently. It is initialized to the ID of process A, because in the beginning A will try to form a tree with itself as its root. If this attempt fails, TREE_ROOT will get modified.

SEQUENCE - An attempt to make a tree with the highest ID process as its root might not succeed if the

global termination condition does not exist. In this case another attempt is made to form a tree. It is possible to get the messages related to different attempts intermingled. SEQUENCE is an integer that is used to keep a count of the number of attempts made to form the tree identified by TREE_ROOT. It helps in discarding the messages related to some previous attempt. It is initialized to one.
ID - It contains the ID of process A.
DIDNOT_TALK - shows whether this process has initiated application communication with some other process or not. It is initialized to true.
PENDING_MESSAGES - It is of type boolean initialized to false. It shows whether there are any messages (related to extra communication) pending with process A that are yet to be processed. The reason for having pending messages is that when a process is unstable and receives messages to form a tree it does not act upon them until it becomes stable. This delay will reduce the frequency of tree formation when a process is unstable and hence the message traffic will be less.

To reduce message traffic, initially only one process can be given the responsibility to start the formation of a tree by initializing its variables as mentioned above and the variables of the rest of the processes will be initialized as follows:

CHILDREN := **null**;
TREE_ROOT := **nil**;
SEQUENCE := 0;
PARENT := **nil**;
FORM_TREE := false;
PENDING_MESSAGES := false;

Process A communicates with its neighbors using the following messages.

a) [MAKE_TREE, SEQ_NO, ROOT, FROM] - This message is sent from process A to its potential children and in return A expects message b) or message f) or a pair of messages out of c), d), and e) described below. MAKE_TREE is the name of the message. Here ROOT is equal to TREE_ROOT and SEQ_NO is equal to SEQUENCE. FROM is process A's ID. Basically this message is sent by process A to propagate the formation of a tree further, where ROOT identifies the root process and SEQ_NO represents the attempt number.

b) [NOT_YOUR_CHILD, SEQ_NO, ROOT, FROM] - Process A sends this message in response to message a) if A has already become a node of the same tree as identified by the SEQ_NO and ROOT fields of the message a). FROM is process A's ID.

c) [YOUR_CHILD, SEQ_NO, ROOT, FROM] - Process A sends this message in response to message a) if A is a node of some other lower priority tree. FROM is process A's ID.

d) [CANCEL, SEQ_NO, ROOT, FROM] - Process A sends this message to its parent if A detects that global termination condition is not true and this tree should be formed again from the beginning. A comes to know that global termination condition does not exist if it receives a CANCEL message from one of its children.

e) [YES, SEQ_NO, ROOT, FROM] - Process A sends this message to its parent if A receives YES message from all its children in response to the MAKE_TREE message it sent to them.

f) [ABORT, SEQ_NO, ROOT, FROM] - Process A sends this message in response to message a) if A is already a node of some higher priority tree.

In response to different events Process A executes different algorithms as described below.

Algorithm 1. When process A is unstable and initiates application communication with process B:
DIDNOT_TALK := false

Algorithm 2. When process A is in unstable state and receives [MAKE_TREE, SEQ_NO, ROOT, 'B'] message from B:
{First check if a new tree is to be formed or not by comparing the root ID and sequence number with that of previous tree being formed at process A}
if ROOT > TREE_ROOT or (ROOT = TREE_ROOT and SEQ_NO > SEQUENCE)
then
begin {a new tree to be formed and the previous one to be ignored}
PARENT := FROM;
TREE_ROOT := ROOT;
SEQUENCE := SEQ_NO;
CHILDREN := null;
{Right now process A is unstable therefore termination condition can not exist. So, process A sets FORM_TREE and does not forward the request to form the tree to other neighbors. When process A changes its state to the stable one, it will propagate the request further by executing the algorithm 12.}
FORM_TREE := true;
PENDING_MESSAGES := false;
{Process A tells process B that it is now one of the nodes of the tree by sending the YOUR_CHILD message}
send [YOUR_CHILD, SEQ_NO, ROOT, 'A'] to B;
end
else {new tree not be formed}
if ROOT = TREE_ROOT and SEQ_NO = SEQUENCE then
{process A has already received this MAKE_TREE message via some other path, therefore A is already a node of the tree being formed}
send [NOT_YOUR_CHILD, SEQ_NO, ROOT, 'A'] to B
else if ROOT < TREE_ROOT then
{abort the formation of a new tree, because root of the previous tree has got higher ID}
send [ABORT, SEQ_NO, ROOT, 'A'] to B;

Algorithm 3. When process A is stable and receives [MAKE_TREE, SEQ_NO, ROOT, 'B'] message from B:
{Same as Algorithm 2 except that if new tree is to be formed then forward the message further to form the rest of the tree instead of postponing it. If the set (NEIGHBOR - PARENT) is empty then process A is a leave of the tree and it should send a response back to process B and A should not forward the MAKE_TREE message}
Execute Algorithm 2;
if FORM_TREE then
begin
if the set (NEIGHBOR - PARENT) is not empty then
send [MAKE_TREE, SEQUENCE, TREE_ROOT, 'A'] to all the processes that are members of the set (NEIGHBOR - PARENT)
else call respond_to_parent;
FORM_TREE := false;
end.

Algorithm 4. When process A is unstable and receives [NOT_YOUR_CHILD, SEQ_NO, ROOT, 'B'] message from B:
{If the message is in context of the latest tree, then simply record that response from process B has been received and set PENDING_MESSAGES to true to postpone further processing of this message till A becomes stable}
if ROOT = TREE_ROOT **and** SEQ_NO = SEQUENCE **then**
    record the fact that response from B in context of the tree identified
    by TREE_ROOT and SEQUENCE has been received and set PENDING_MESSAGES
    to **true**
else discard the message.

Algorithm 5. When process A is stable and receives [NOT_YOUR_CHILD, SEQ_NO, ROOT, 'B'] message from B:
if ROOT = TREE_ROOT **and** SEQ_NO = SEQUENCE **then**
    record the fact that NOT_YOUR_CHILD message was sent from B in context
    of the tree identified by TREE_ROOT and SEQUENCE
    and call respond_to_parent
else discard the message.

Algorithm 6. When process A is unstable or stable and receives [YOUR_CHILD, SEQ_NO, ROOT,'B'] message from B:
if ROOT = TREE_ROOT **and** SEQ_NO = SEQUENCE **then**
    CHILDREN := CHILDREN + 'B'
else discard the message.

Algorithm 7. When process A is in unstable condition and receives [YES, SEQ_NO, ROOT, 'B'] message from B:
if ROOT = TREE_ROOT **and** SEQ_NO = SEQUENCE **then**
    record the fact that YES message came from B in context of the tree
    identified by TREE_ROOT and SEQUENCE and set PENDING_MESSAGES
    to enable its processing later on because right now A is unstable
else discard the message.

Algorithm 8. When process A is in stable condition and receives [YES, SEQ_NO, ROOT, 'B'] message from B:
if ROOT = TREE_ROOT **and** SEQ_NO = SEQUENCE **then**
    record the fact that YES message came from B in context of the tree
    identified by TREE_ROOT and SEQUENCE and call respond_to_parent
else discard the message.

Algorithm 9. When process A is unstable or stable and receives [ABORT, SEQ_NO, ROOT, 'B'] message from process B:
{This message has been sent by B to abort the formation of the tree identified by SEQ_NO and ROOT. Process A acts upon this message if it is still a node of this tree}
if ROOT = TREE_ROOT **and** SEQ_NO = SEQUENCE **then**
begin {A is a node of the tree to be aborted}
    FORM_TREE := false;
    PENDING_MESSAGES := false;
    CHILDREN := **null**;
    TREE_ROOT := **nil**;

```
        SEQUENCE := 0;
        if ROOT <> ID then
                {A is an intermediate node not the root of the tree being
                aborted, therefore send the ABORT message towards the root}
                send [ABORT, SEQ_NO, ROOT, 'A'] to PARENT
end
else {A is not a node of the tree to be aborted}
        discard the message.
```

Algorithm 10. When process A is unstable and receives [CANCEL, SEQ_NO, ROOT, 'B'] message from process B:
{This message has been sent by B to cancel the formation of the tree identified by SEQ_NO and ROOT, because B has detected that the termination condition does not exist and the tree is to be formed again. Process A acts upon this message if it is still a node of this tree}

```
if ROOT = TREE_ROOT and SEQ_NO = SEQUENCE then
        {A is a node of the tree to be canceled}
        record the fact that CANCEL message was received from B in context of
        the tree identified by TREE_ROOT and SEQUENCE and to process this
        message further when A becomes stable, set PENDING_MESSAGES to true
else discard the message.
```

Algorithm 11. When process A is stable and receives [CANCEL, SEQ_NO, ROOT, 'B'] message from process B:
{This message has been sent by B to cancel the formation of the tree identified by SEQ_NO and ROOT, because B has detected that the termination condition does not exist and the tree is to be formed again. Process A acts upon this message if it is still a node of this tree}

```
if ROOT = TREE_ROOT and SEQ_NO = SEQUENCE then
begin {A is a node of the tree to be canceled}
        record the fact that CANCEL message came from B;
        call respond_to_parent;
end
else discard the message.
```

Algorithm 12. When process A changes its state from unstable to stable:

```
if FORM_TREE then
begin {a new tree is to be formed}
        if the set (NEIGHBOR - PARENT) is not empty then
                send [MAKE_TREE, SEQUENCE, TREE_ROOT, 'A'] to all the processes
                that are members of the set (NEIGHBOR - PARENT)
        else call respond_to_parent;
        FORM_TREE := false;
end
else {no new tree to be formed}
        if PENDING_MESSAGES then
        begin
                {a few messages are yet to be processed and an appropriate
                 response is to be sent to PARENT}
                PENDING_MESSAGES := false;
                call respond_to_parent
        end
```

Algorithm 13. When process A is unstable and gets a message from the local operating system that A can not communicate with process B:

```
{If A is not able to communicate with B then it means that either process B has failed or the link AB has
failed}
NEIGHBOR := NEIGHBOR - 'B';
if PARENT is equal to 'B' then
begin {B is the parent of A}
      {It is possible that the network has partitioned into two such that
       B and the root of the tree are in one partition and A in the other
       one. If root is in the first partition then there is no process in the
       second partition to initiate the tree formation to detect the
       termination condition separately. In the second partition, at least A
       knows about this possibility. Therefore, A prepares itself to start a
       tree with itself as root. Similarly, there might be other processes
       that also start making different trees of their own. In the end, only
       the process having the highest ID among these processes will succeed.
       If the network has not got partitioned then the attempt of forming a
       new tree by all these processes will compete with the attempt by the
       original root that existed before the fault occurred. Again the process
       with the highest ID will win.}
      FORM_TREE := true
end
else
      if 'B' is in the set CHILDREN then
      begin  {B is a child of A}
            {If the network is partitioned then A is in the partition in
             which the root exists. Therefore, A simply cancels the
             formation of this tree and the root will try to make another
             attempt. A fresh attempt is required to take care of the
             disturbances created by the failure.}
            CHILDREN := null;
            TREE_ROOT := nil;
            SEQUENCE := 0;
            FORM_TREE := false;
            PENDING_MESSAGES := false;
            DIDNOT_TALK := true;
            send [CANCEL, SEQUENCE, TREE_ROOT,'A'] to PARENT;
            PARENT := nil;
      end
      else
            if TREE_ROOT is nil then
                  {TREE_ROOT is nil and A is not a node of any tree.
                   A starts making its own tree}
                  FORM_TREE := true
            else
                  {A is a node of some tree which does not have edge AB.
                   Therefore, if A and B are not able to communicate it
                   will not effect the formation of the tree}
                  discard the message;
if FORM_TREE is true then
```

```
begin {A starts the formation of a new tree with itself as its root}
      TREE_ROOT := 'A';
      SEQUENCE := SEQUENCE + 1;
      {We do not initialize SEQUENCE to 1 because meanwhile if A detects
       another failure then the next tree will have the same priority as this one}
      CHILDREN := null;
      PARENT := nil;
      PENDING_MESSAGES := false;
      {If A is stable now then we start making a new tree now else we do
       nothing and when A will change its state it will execute algorithm 12.
       to form a new tree}
      if process A is stable then
        if the set NEIGHBOR is not empty then
            begin
                  send [MAKE_TREE, SEQUENCE, TREE_ROOT, 'A'] to all
                  the processes that are members of the set NEIGHBOR;
                  FORM_TREE := false;
             end
        else call respond_to_parent {to terminate itself};
end;

Procedure respond_to_parent;
begin
      if responses of MAKE_TREE message have been received from
        all the processes in the set (NEIGHBOR - PARENT) or
        set (NEIGHBOR - PARENT) is empty
        {response can be the message NOT_YOUR_CHILD or ABORT or a
        pair of YOUR_CHILD and YES messages or a pair of YOUR_CHILD
        and CANCEL. If the response is the ABORT message, control
        will never reach here, see algorithm 9.}
      then
        if DIDNOT_TALK and (responses from all children
        is YES message or CHILDREN set is null)
        then
            {All the processes in the subtree below process A are
            stable. If A is the root then the termination condition
            exists otherwise A sends YES message to its parent}
            if ID = TREE_ROOT then start termination wave
            else send [YES, SEQUENCE, TREE_ROOT, 'A'] to PARENT
        else {termination condition does not exist}
            if ID = TREE_ROOT then
            begin {A is the root, start the formation of a new tree}
                  SEQUENCE := SEQUENCE + 1;
                  send [MAKE_TREE, SEQUENCE, TREE_ROOT, 'A'] to
                  all the processes that are members of the set (NEIGHBOR - PARENT);
                  DIDNOT_TALK := true
            end
            else {a is an intermediate node in the tree}
            begin
                  send [CANCEL,SEQUENCE,TREE_ROOT,'A'] to PARENT;
```

```
                DIDNOT_TALK := true;
                call reset;
            end
end.

Procedure reset;
{This procedure enables the formation of a new tree in case of failures}
begin  CHILDREN := null;
       TREE_ROOT := nil;
       SEQUENCE := 0;
       PARENT := nil;
end;
```