CS5320: Distributed Computing Project Report

Jonathan Marbaniang (cs16mtech11006) &

Mrinal Aich (cs16mtech11009)

Introduction:

In this report, we present two Termination Detection algorithms namely,

1. *Message-optimal Algorithm for Distributed Termination Detection*
2. *Delay-optimal Distributed Termination Detection algorithm.*

These are referred IEEE papers, given in the references. Each algorithm has its own property and usefulness. We've also analysed and compared the performance of both algorithms based on average response time per node and overhead of average Control Messages.

Algorithm 1:

Title: A More Efficient Message-Optimal Algorithm for Distributed Termination Detection.
Authors: Ten-Hwang, Lai Yu-Chee Tseng, Xuefeng Dong.
Published in: Parallel Processing Symposium, 1992. Proceedings., Sixth International

The paper has proposed a Termination Detection algorithm that has better message complexity than Spanning Tree(Topor's) and Message-optimal termination detection algorithms studied in the literature.

Idea:

- Let S = {P1, P2, P3,....Pn} be the process set of the underlying system which are organized as a spanning tree T, that can be done at compile time. Let P1 be the root of T.
- A process can either be **ACTIVE** or **PASSIVE.**
- A PASSIVE process can become ACTIVE only after if receives an APPLICATION message from another process.
- Termination Detection can only be initiated by the “root” node.
- Two types of control messages are used in this algorithm i.e **START messages and FINISHED(k) messages**, where k is an integer.

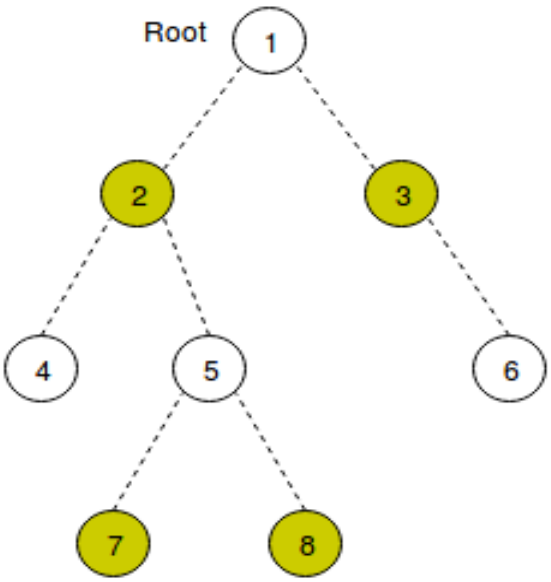


Figure 1: Initial States of each node.

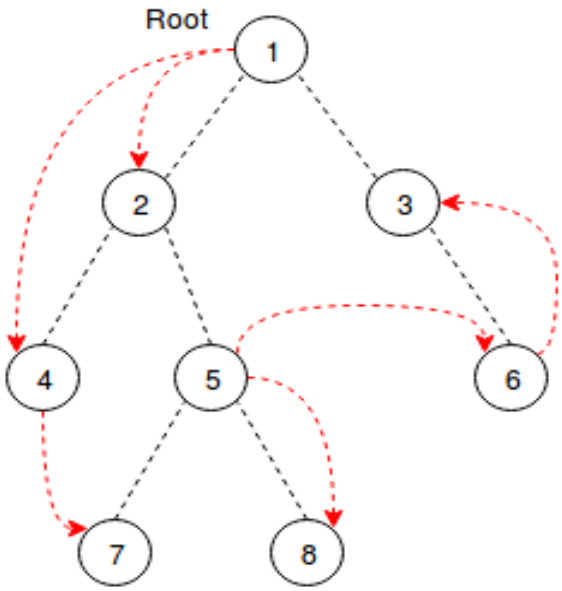


Figure 2: Nodes become active on receiving APPLICATION messages

- The START messages are used to “start” the execution of the algorithm. The root starts the algorithm by sending a START message to every other process along the edges of T.
- The word 'start' is used in the sense of starting to use control messages.
- Before the algorithm starts to operate each process keeps a count of the number of messages it sends i.e. only simple, passive bookkeeping is being done.
- Until the root issues a START message, no control message has ever been sent between processes.
- When a Non-Root process **receives a START message, it enters into Termination Detection Mode** and propagates the START message to it's children, if any, along the edges of the Spanning Tree.

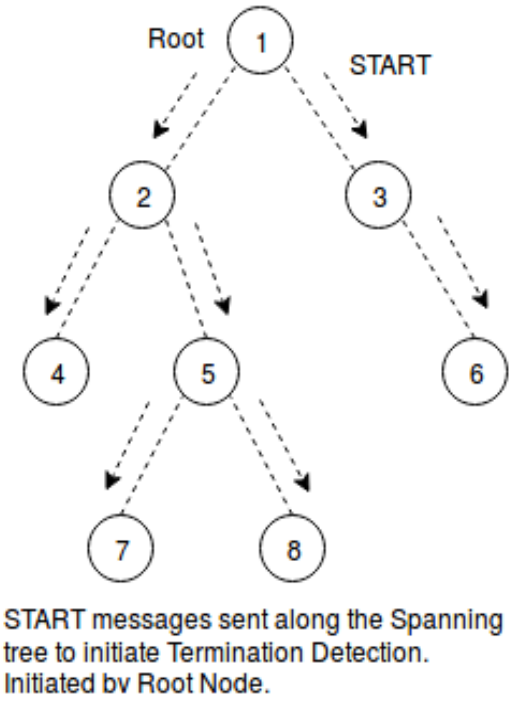


Figure 3: START messages propagated along the Spanning Tree

- All the processes keep on exchanging APPLICATION messages with each other. Each process **keeps count of the number of APPLICATION messages** it has sent to other processes in an **'out'** counter.

- The **FINISHED(k) message** is used in the message-counting scheme to inform a sender process that k APPLICATION messages sent by it have been finished or acknowledged.

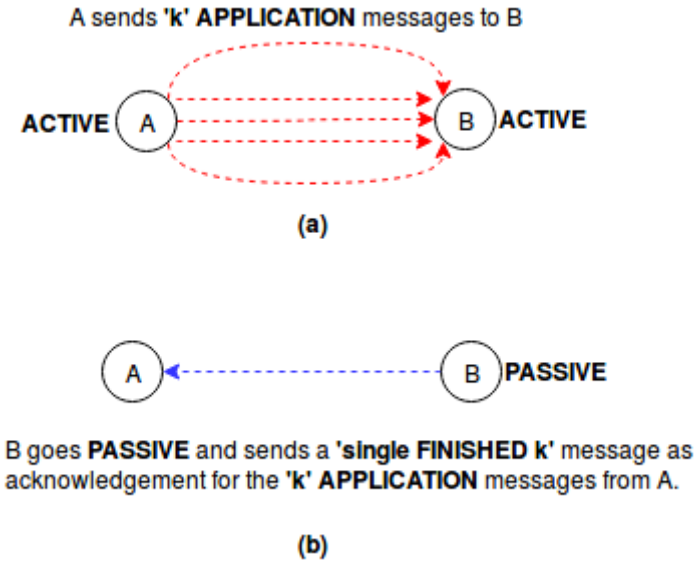


Figure 4: Example of a FINISHED(k) message for a set of 'k' APPLICATION messages.

- The FINISHED(k) message is sent by a process only if the process is PASSIVE and is in Termination Detection mode.

- The algorithm checks if a node is terminated by checking if its 'out' is "**FREE**" (all messages sent to other processes have been acknowledged) and "**PASSIVE**".

Data Structures:

- Each process Pi, 1<=i<=n, maintains four variables:
  - **in[1...n]:** An integer array, where $in_i[j]$ counts the number of messages received from Pj which either have not been finished (acknowledged), or have been finished but have not yet been so noticed to Pj. Initially, $in_i[j]$ is 1 if Pj is Pi's parent in T, and 0 otherwise. Note that the root 'P1' has no parent.

  - **out:** an integer that indicates the number of unfinished (or thought-to-be-unfinished) messages sent by Pi. Initially, 'out' is the number of children Pi has in T.

  - **mode**: a boolean variable indicating whether Pi is in DT (Detecting Termination) or NDT (Not Detecting Termination) mode. Initially, Pi is in NDT mode. Only after receiving a START message, it changes to DT mode.

  - **parent:** a pointer for a loaded(not free) Pi to indicate where the most recent major message came from. Initially, $parent_i$, i != 1, is initialized to Pi's parent in T, and Parent1 is NULL.

Algorithm:

A1: (Upon sending a basic message to Pj)

$out_i := out_i+1;$

A2: (Upon receiving a basic message from Pj)

$in_i[j]:= in_i[j]+1;$

if ($Parent_i$ == NULL) AND (i != 1) then $Parent_i := j;$

A3: (Upon deciding to switch to DT mode) **/* for p1 */**

or (Upon receiving a START message) **/* for Pi 2<=i<= n * /**

$mode_i := DT;$

for each child Pj of Pi do:

send a START message to Pj;

end for

if (Pi is idle) then:

call respond_minor(i);

call respond_major(i);

end if

A4: (Upon receiving a FINISHED(k) from Pj)

$out_i := out_i - k$;

if ($mode_i$=DT) AND (Pi is idle) then call respond_major(i);

A5: (Upon turning idle)

```
if (mode_i = DT) then
        call respond_minor(i);
        call respond_major(i);
end if;
```

A process calls **respond_minor()** when it turns **PASSIVE** to send out **FINISHED** messages to all its neighbours, except for it's parent, from which it received **APPLICATION** messages.

Procedure respond_minor(i: integer)

```
begin
        for each j!= Parent_i with in_i[j]!=0 do
                send a FINISHED (in_i[j]) to Pj;
                in_i[j]:=0;
        end for ;
end;
```

A process calls **respond_major()** when it turns **PASSIVE** and it has received **FINISHED** messages for all the **APPLICATION** messages it sent. It is used to sent a **FINISHED** message to the parent of the process.

Procedure respond_major(i: integer)

```
begin
        if (out_i=0) then
                if (i=1) then report termination
                else
                        send a FINISHED(in_i[Parent_i]) to Parent_i;
                        in_i[Parent_i]:=0;
                        Parent_i := NULL;
                end if;
        end if;
end;
```

Spring 2017 CS5320: Distributed Computing Project Report

Complexity analysis:

- The number of START messages sent is |V|-1.
- In the worst case, each APPLICATION message may require a separate FINISHED message. So, number of FINISHED messages is |M|, where |M| is the number of APPLICATION messages exchanged.
- Also |V|-1 FINISHED messages have to be sent by each process to it's parent.
- In the best case, a single FINISHED message would be enough for all the APPLICATION messages.
- This will happen when a process receives APPLICATION messages only from it's parent. The only FINISHED message it will send is the one when it goes PASSIVE and is in Termination Detection mode.
- So worst case message complexity is |M|+2(|V|-1).
- Best case message complexity is 2(|V|-1).
- The response time to an APPLICATION message is high. The first APPLICATION message sent by a process won't be acknowledged until the receiver goes PASSIVE and is in Termination Detection mode.

Algorithm 2:

Title: An efficient Delay-Optimal Distributed Termination Detection algorithm.
Authors: Nihar R. Mahapatraa, Shantanu Dutt
Published in: Journal of Parallel and Distributed Computing, Volume 67 Issue 10, October, 2007

The paper has proposed a Termination Detection algorithm that has less delay than the Spanning Tree(Topor's) and Message-optimal termination detection algorithms studied in the literature.

Definitions:

1. A Node is ***busy*** if it has some primary computation to perform, otherwise it is ***idle***.
2. A Node is ***free*** if and only if it is idle and all primary messages sent have been acknowledged, otherwise it is ***loaded***.
3. A non-root node is ***active*** if it has either not sent any STOP messages to its parent, otherwise, it is ***inactive***.

Idea:

1. If no primary messages are sent, a non-root Node *'i'* sends a STOP message to its parent once it is free and has received STOP messages from all its child nodes, if any.

2. STOP messages are passed up the tree until the root node receives STOP's from all its children.
   Finally, when root node also becomes free, it broadcasts TERMINATION message to all nodes along the tree.

3. An APPLICATION message M(i,j) context is maintained at the source node *'i'* until an ACKNOWLEDGE is received for that message. If Node *'j'* has not sent STOP, (i.e. Its Active) it sents the ACKNOWLEDGE immediately.

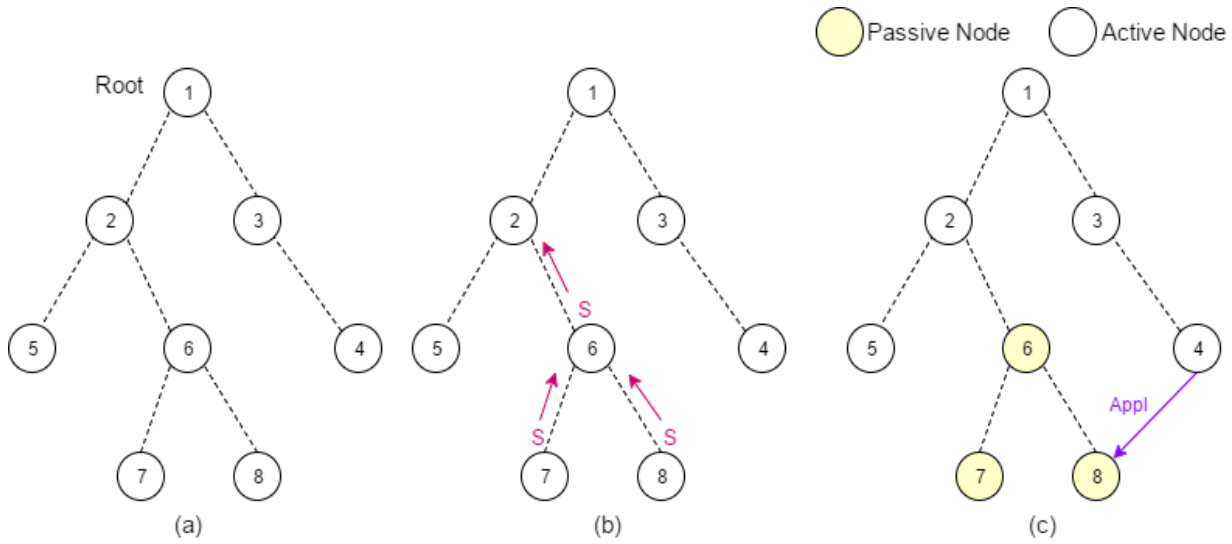
4. If Destination node *'j'* has reported STOP to its parent before receiving M(i,j), it "resumes," by sending a RESUME (i,j) upward in the tree. It is sent to Nullify the STOP sent before.

5. The RESUME(i,j) from Node *'j'* travels upward till an ancestor node *'k'* that has not reported STOP to its parent. Now, all the nodes along the path *'j'* to *'k'* are Active and would have to resend their STOP.

6. The ancestor node *'k'* receiving the RESUME, sends ACKNOWLEDGE for message (i,j) down the tree to node *'j'*. Finally, node 'j' sends it to the source node *'i'* of the message.

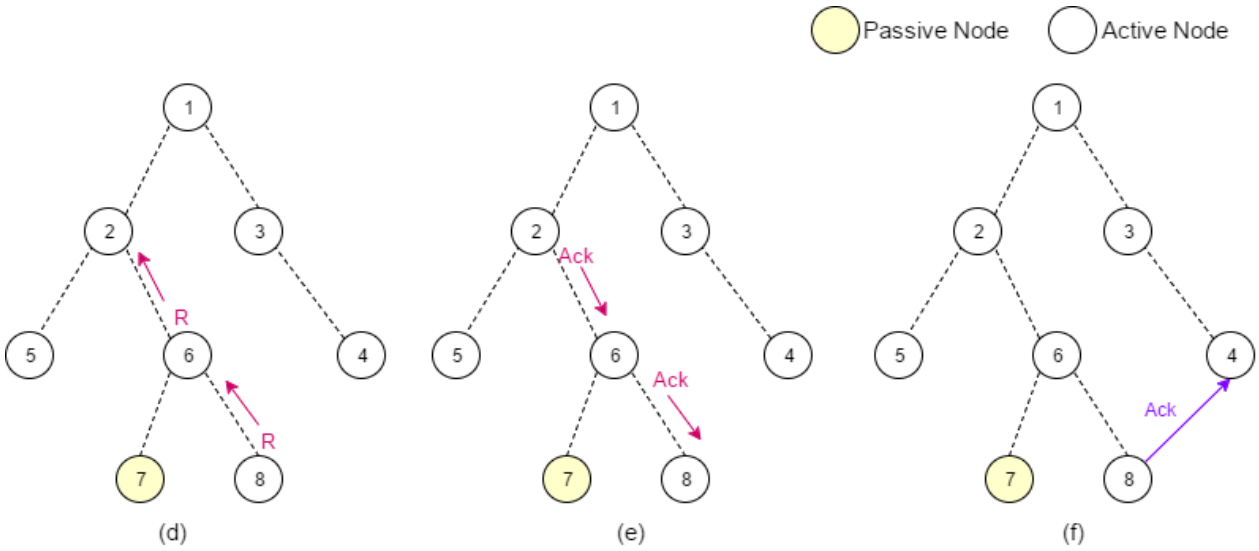
7. Source Node *'i'* on receiving ACKNOWLEDGE, deletes context for the APPLICATION message (i,j). It can now report STOP when it becomes free and has received STOPs from all its children.

Illustration -



In figure, initially there are 8 active nodes with node-1 as the root (a). An active node on receiving an APPLICATION message sends an ACKNOWLEDGEMENT immediately.

A STOP message is sent to the parent when a node gets PASSIVE and it has received STOP messages from all of its children (if they exist) (b). A PASSIVE node (e.g. Node-4) on receiving an APPLICATION message goes ACTIVE but does not send the acknowledgement immediately (c).



Instead, a RESUME message is sent to its parent indicating that the node is active again. If the parent node (Node-6) is also PASSIVE, it goes ACTIVE, marks the child as ACTIVE and forwards the RESUME message to its parent (Node-2). This process continues upwards until an active node receives the RESUME message (d).

The active parent (Node-2) marks the child as ACTIVE and sends an ACKNOWLEDGEMENT message to the child. If an intermediate node (Node-6) receives the ACKNOWLEDGEMENT message, it forwards the message to its child from whom it received the RESUME message. This continues until it reaches the destination of the APPLICATION message (Node-8) (e).

The destination of the APPLICATION message on receiving the ACKNOWLEDGEMENT message sends it to the sender (f). Now, the source (Node-4) removes the context of this APPLICATION message.

Data Structures:

Each process Pi, 1<=i<=n, maintains these variables:

- ***idle*** {0|1} - Node is ***idle*** if it has no primary computation to perform, otherwise it is ***busy***.
- ***free*** {0|1} - Node is ***free*** if and only if it is idle and all primary messages sent have been acknowledged, otherwise it is ***loaded***.
- ***inactive*** {0|1} - Non-root node is ***active*** if it has either not sent any STOP messages to its parent, otherwise, it is ***inactive***.
- ***child_inactive[1..j]*** {0|1} - Marks which child nodes are active or inactive. 'j' is the number of child nodes.
- ***num_unack_msgs*** – Maintains the count of the unacknowledged APPLICATION messages sent.
- ***terminated*** {0|1} – Indicates whether the node has terminated or not.

Algorithm STATIC_TREE_DTD(i)

/* Detects termination of a parallel primary computation on P nodes */

Begin
Node i, 0 ≤ i < P, executes the following steps:

1. Initialization: *idle* := 0; *free* := 0; *inactive* := 0; for all children j of i,
   child_inactive[j] := 0; *num_unack_msgs* := 0; *terminated* := 0.
   /* Here *idle* = 1 (0) → Node is idle (busy);
   free = 1 (0) → Node is free(loaded);
   inactive = 1 (0) → Node is inactive (active)
   child_inactive[j] → 1 (0) if child 'j' of Node 'i' is Inactive (Active)
   num_unack_msgs → number of unacknowledged Appl messages
   terminated → 1 (0) Node has (has not) detected termination. */

Repeat

2. **On** (Node *'i'* becoming idle) *idle* := 1.
3. **If** (*idle* = 1) **and** (*num_unack_msgs* = 0) **then** *free* := 1.
4. **On** (receiving a STOP from Node j) *child_inactive*[j] = 1.
5. **If** (*child_inactive*[j] = 1 for all children *j* of *i*) **and** (*free* = 1) **and** (*inactive* = 0) **then begin**
   If (*i = root*) **then begin**
   Send TERMINATION to all child Pes; *terminated* := 1;
   Else Send STOP to parent Node;
   Endif
   inactive := 1;
   Endif
6. **On** (issuing a primary message *M(i,j) num_unack_msgs*++;
   /* *M(i,j)* indicates a message from Node *'i'* to Node *'j'*/
7. **On** (receiving a primary message *M(j,i)*) **begin**
   idle := 0; *free* := 0;
   If (*inactive* = 1) **then begin**
   Send RESUME (j,i) to parent Node; *inactive* := 0;
   Else Send ACKNOWLEDGE(j,i) to Node '*j*'.
   Endif
   Endon
8. **On** (receiving RESUME (j,k) from Node '*l*') **begin**
   child_inactive[l] = 0;
   If (*inactive* = 1) **then begin**
   Send RESUME (j,k) to parent Node; *inactive* := 0;
   Else Send ACKNOWLEDGE (j,k) to child Node on path to Node *'k'*
   Endif
   Endon

9. **On** (receiving ACKNOWLEDGE (j,k))
   If (*i = j*) **then** *num_unack_msgs*--;
   Else if (*i = k*) **then** Send ACKNOWLEDGE (j,k) to Node '*j*';
   Else Send ACKNOWLEDGE(j,k) to the child node on the path to Node *'k'*.

Endon

10. **On** (receiving TERMINATION) **begin**
    Send TERMINATION to all child Nodes; *terminated* := 1;
    Endon

Until (*terminated* = 1)
End /* Algorithm STATIC_TREE_DTD */

Message Complexity -

Each primary message can potentially cause O(D) RESUME messages before an acknowledgment for it is issued, hence the message complexity can be **O(MD + N)** where

M – Number of Application messages.
D – Diameter of the graph.
N – Number of nodes in the graph.

System Design -

The following are some assumptions for the development environment -

1. All channels in the system are bidirectional, reliable and satisfy the first-in-first-out (FIFO) property.

2. A reliable socket connection (TCP or SCTP) is used to implement a channel. All messages between neighboring nodes are exchanged over these connections.

3. Initially, each node in the network is either active or is made active (using a START message).

4. When a node Ni is active, it sends a certain number of messages randomly chosen between *minMsgLt* and *maxMsgLt* to its neighbors. Suppose Ni has k neighbors. Then it randomly chooses a neighbor out of k and then sends a message.

Ni sends messages with a delay that is exponentially distributed with an average μ-delay. After sending M messages to its neighbors where M is randomly chosen to be such that minMsgLt ≤ M ≤ maxMsgLt, Ni becomes passive.

5. Only an active node can send a message.

6. A passive node, on receiving a message, becomes active and follows the steps mentioned in Step-3. But to ensure that the computation eventually terminates, we add one extra rule: *Once a node Ni has sent maxSent, it can't become active on the receipt of any new message.*

7. Each node logs onto his local buffer along with the time-stamp in the format:
"Time: Message sent by Ni to Nj".

Input - The input to the program will be a file, named "*in-params.txt"*, consisting of all the parameters described above and the graph topology. The first line of the input file will contain the parameters:

n, minMsgLt, maxMsgLt, μ-delay, maxSent

In addition, another file describing the topology of the input graph named as "*topology.txt"*.
The format of the file is as follows:

The first line will be number of nodes in the system which is n. The next line will show the binding of nodes to IP addresses and their ports.

For instance if n is 3, then the following is a binding:

1 – 192.168.1.1:3001
2 – 192.168.1.2:3002
3 – 192.168.1.3:3003

The next few lines contains the graph topology in the form of an adjacency list.
For n as 3, a sample topology showing a complete graph is as follows:

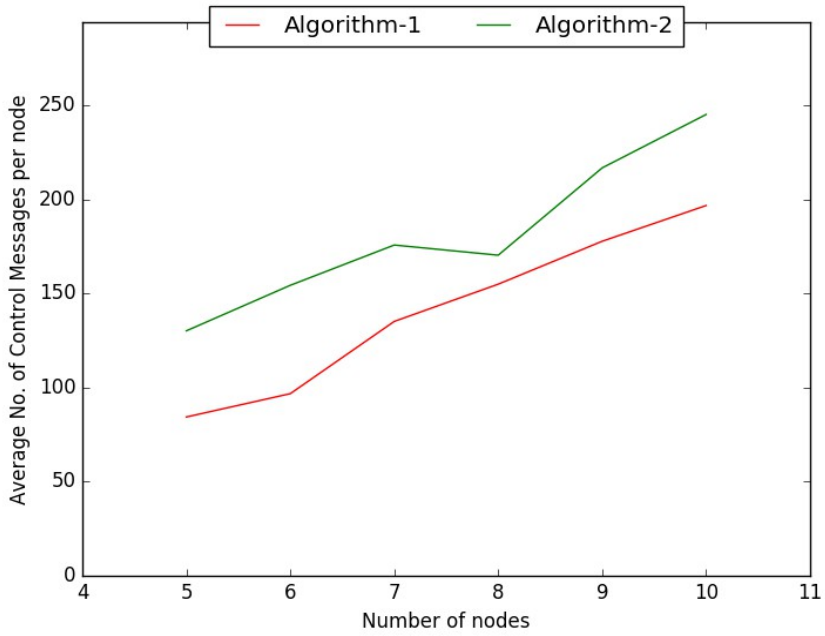
1 2 3
2 1 3
3 1 2

Next, the spanning tree of the above topology is mentioned in the subsequent lines.
Each line will show the parent and its children with the first line showing the children of the root.
For instance a spanning tree with n being 7 as follows:

1 2 3
2 4 5
3 6 7

Performance Comparision -

1. Average Control Messages Per Node



- *Algorithm 1* - The number of control messages is less because a node will send FINISHED (or ACKNOWLEDGE) to a sender node only when it goes PASSIVE and is in Termination Detection mode. A single FINISHED message will be sent for multiple received APPLICATION messages. This results in the reduction of control messages in the system.

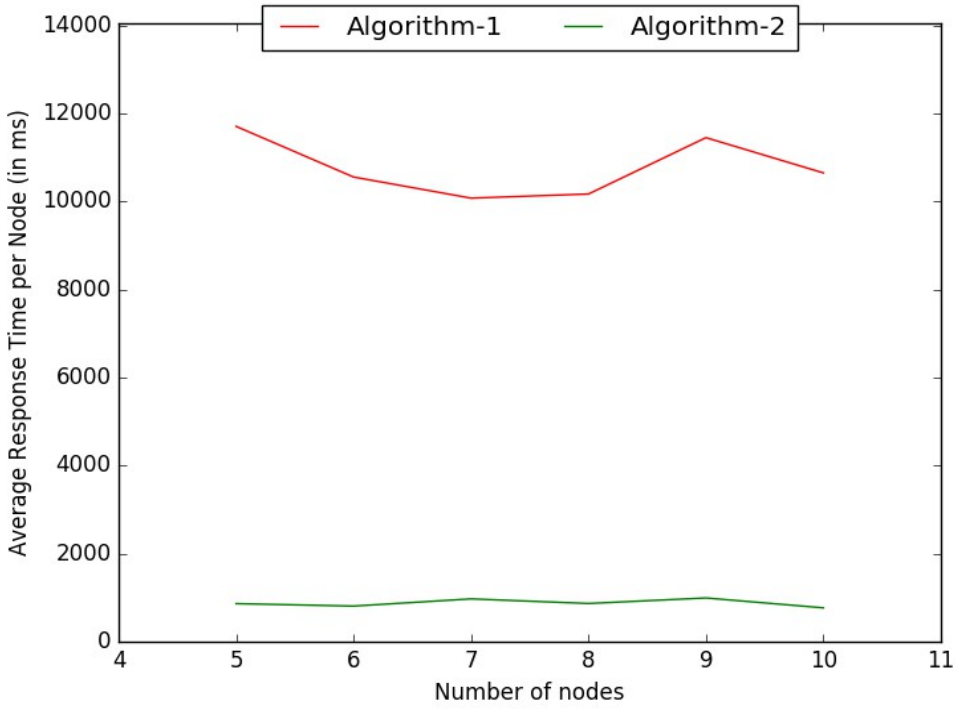
- *Algorithm 2* - Each primary message may cause a lot of RESUME-ACKNOWLEDGE messages along the tree which increases the number of control messages. In the worst case, a single APPLICATION message may trigger an RESUME to go up till the root node and come down with an ACKNOWLEDGE message.

- *Comparison* -

  In algorithm-1, the number of control messages is very low compared to the number of APPLICATION messages. A single FINISHED message is enough for multiple APPLICATION messages. This reduces the number of control messages.

  In algorithm-2, the number of control messages is high because an APPLICATION message can cause a lot of RESUME-ACKNOWLEDGE messages along the tree before the actual acknowledge is sent. This increases the number of control messages. Also, a STOP message is sent every time when a node gets Passive.

2. Average Response Time (Delay) per node



- *Algorithm 1* - The response time to an APPLICATION message depends on when the node gets PASSIVE, irrespective of when it was received. This holding back of FINISH (ACKNOWLEDGE) message increases the response time (delay) of the APPLICATION message.

- *Algorithm 2* – The response time to an APPLICATION message depends on the destination node being Active or Passive. An immediate ACKNOWLEDGE is returned if the destination is Active.
  Otherwise, a RESUME message is sent upwards to an Active Ancestor to Nullify the STOP message previously sent. Destination node on receiving the ACKNOWLEDGE from its ancestor along the downward path, sends it the source node of the APPLICATION message.

- *Comparision* -
  In algorithm-1, the response time to an APPLICATION message is very high. The first APPLICATION message sent by a process won't be acknowledged until the receiver goes PASSIVE and is in Termination Detection mode. This holding back of the ACKNOWLEDGE increases the response time.

  In case of algorithm-2, the response time is very low as
  - The response time overhead depends on the RESUME-ACKNOELEDGE cycle along the tree (upward-downward). *The impact would be more as the height of the tree increases.*
  - As the number of nodes increases and all are initially Active, the response time of the APPLICATION messages is very low thus reducing the response time.

References -

[1] Jeff Matocha, Tracy Camp, Nov. 1998. “A taxonomy of distributed termination detection algorithms.”, Journal of Systems and Software, Volume 43 Issue 3, Nov. 1998, Pages 207 - 221

[2] Chandrasekaran, S., Venkatesan, S., 1990. “A message-optimal algorithm for distributed termination detection.”, Journal of Parallel and Distributed Computing 8, 245-252.

[3] Lai, T., Tseng, Y., and Dong, X., 1992. “A more efficient message optimal algorithm for distributed termination detection.”, Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, pp. 274-281.

[4] Nihar R. Mahapatra, Shantanu Dutt, October 2007, “An efficient delay optimal distributed termination detection algorithm”, Journal of Parallel and Distributed Computing Volume 67, Issue 10, Pages 1047-1066.

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

[6] Lai, T., 1986. “Termination detection for dynamically distributed systems with non-first-in-first-out communication.” Journal of Parallel and Distributed Computing 3, 577-599.