Distributed Termination Detection

- Determining termination of distributed programs using snapshots requires checking periodically which is inefficient.
- This can be improved by choosing an active node from the previous snapshot, and have this node start the next snapshot when it becomes inactive.
- But, we can still improve on this by designing a special purpose distributed algorithm for detecting termination.
- Determining termination for asynchronous distributed systems is difficult.
  - We can’t just check whether all threads have terminated, since the threads will continue to wait on queues.
  - We can’t just check whether all queues are empty, since there may be messages in transit.
  - Observing what is happening on multiple machines at exactly the same time is difficult or impossible.

Distributed Termination Detection

- We will study a few algorithms for termination detection, each of which is more general than the previous one.
- In our model, we ignore issues related to fault tolerance. In other words, the nodes of a concurrent computation and the communication channels between different nodes can never fail.
- Our model is the following:
  - A set of processes is executing a concurrent program.
  - A pair of processes may be connected by a directional communication channel. A bi-directional channel can be built by adding two channels in opposite directions.
  - A channel is error-free and delivers the messages in the order they are sent.
  - There is a unique process called source which initiates the computation. Also, every process is accessible from the source along some path.

Distributed Termination Detection

- The computation starts when the source process sends a message along at least one of its outgoing edges.
- A process starts computation when it has received the first message along one of its incoming edges. After starting computation, a process may send messages along its outgoing edges and receive additional messages along its incoming edges.
- A process may decide to terminate when its computation is finished. If a process terminates, it will not send any new message. However, it may restart again if some other process sends a message along one of its incoming edges.
- We distinguish between two kinds of messages. The first kind includes all messages related to the concurrent computation (whatever it is) and we simply call such a message as a message.
- The second kind of messages are sent and received by processes for detecting termination. We call such a message as a signal.

Distributed Termination Detection

- We will assume existence of special channels for sending the signals. For example, if there is a message channel from node $n_1$ to node $n_2$, we will assume that there is a channel from $n_2$ to $n_1$ for sending a signal.
- Another assumption is that a node is always able to send and receive signals irrespective of its state i.e., even if the computation is terminated or the message channel is blocked.
- We assume a node-task for each of the nodes participating in the computation. Each node-task has two subtasks. The first subtask is responsible for communicating with the outside world (i.e., with other nodes). The second subtask is responsible for the initialization of different variables and for the underlying computation.
- Each node contains two array Incoming and Outgoing. For example for node I, Outgoing(J).Exists is true if there is a channel from node I to node J.
Distributed Termination Detection

```haskell
task type Nodes is
  entry Message(Data : Integer; ID : Integer);
  entry Signal(ID : Integer);
end Nodes;

Node : array (1..N) of Nodes;

task body Nodes is
  ---- Global variables.
task Main_Process; --- this subtask does the computation
task body Main_Process is ....;
  --- here starts the communication subtask
begin
  loop
    select
      accept Message(Data : Integer; ID : Integer) do
        Received_ID := ID;
        Received_Data := Data;
        end Message;
      end --- process the message here
    or
      accept Signal(ID : Integer) do
        Received_ID := ID;
        end Signal;
      end --- process the signal here
    end select;
  end loop;
end Nodes;
```

Dijkstra-Scholten Algorithm for a Tree

- First, let us consider the case of a simple process graph which is a tree.
- A distributed computation which is tree-structured is not uncommon. (Such a process graph may arise when the computation is strictly divide-and-conquer type.)
- For a tree, it is easy to detect termination. When a leaf process determines that it has terminated, it sends a signal to its parent. In general, a process waits for all its children to send signals and then it sends a signal to its parent.
- The program terminates when the root receives signals from all its children.

Dijkstra-Scholten Algorithm for Acyclic Directed Graphs

- The algorithm for a tree can be extended to acyclic directed graphs. We add an additional edge `Deficit` to each edge.
- On an incoming edge, `Deficit` will denote the difference between the number of messages received and the number of signals sent in reply.
- When a node wishes to terminate, it waits until it has received signals from outgoing edges reducing their deficits to zero.
- Then it sends enough signals to ensure that the deficit is zero on each incoming edge.
- Since the graph is acyclic, some nodes will have no outgoing edges and these nodes will be the first to terminate after sending enough signals to their incoming edges. After that the nodes at higher levels will terminate level by level.
• If cycles are allowed, the previous algorithm does not work. This is because, there may not be any node with zero outgoing edges. So, potentially there is no node which can terminate without consulting other nodes.

• The Dijkstra-Scholten algorithm solves this problem by implicitly creating a spanning-tree of the graph.

• A spanning-tree is a tree which includes each node of the underlying graph once and the edge-set is a subset of the original set of edges.

• The tree will be directed (i.e., the channels will be directed) with the source node (which initiates the computation) as the root.

• The spanning-tree is created in the following way. A variable \texttt{First Edge} is added to each node. When a node receives a message for the first time, it initializes \texttt{First Edge} with the edge through which it received the message.

• Note that the spanning tree depends on the order of messages in the system.

Termination is handled by each node in three steps:

1. Send signals on all incoming edges except the first edge.
2. Wait for signals from all outgoing edges.
3. Send signals on \texttt{First Edge}.

• In the first step, each node will send signals which reduces the deficit on each incoming edge to zero.

• Similarly, the number of signals received on each outgoing edge should reduce each of their deficits to 0.

• Once steps 1 and 2 are complete, a node informs its parent in the spanning tree about its intention of terminating.

type Edge is
  record
    Exists : Boolean := False;
    Deficit: Integer := 0;
  end record;

N_Signals : Integer := 0;
First_Edge: Integer := 0;

--Message processing is done here
if First_Edge = 0 then
  First_Edge := Received_ID;
end if;
Wait(S);
Incoming(Received_ID).Deficit :=
  Incoming(Received_ID).Deficit + 1;
Signal(S);

--Signal processing is done here
Wait(S);
N_Signals := N_Signals - 1;
Signal(S);

procedure Send_Message(Data : Integer; ID : Integer) is
begin
  Wait(S);
  N_Signals := N_Signals + 1;
  Signal(S);
  Node(ID).Message(Data, I);
end Send_Message;

function Decide_to_Terminate return Boolean is
procedure Send_Signals(ID : Integer) is
begin
  while Incoming(ID).Deficit > 0 loop
    Incoming(ID).Deficit := Incoming(ID).Deficit -1;
    Signal(S);
    Node(ID).Signal(I);
    Wait(S);
  end loop;
end Send_Signals;

begin
Dijkstra-Scholten Algorithm for Cyclic Directed Graphs

Safety property: If the source decides that termination has occurred, then all processes are quiescent.

Liveness property: If all processes become quiescent, then eventually the program decides to terminate.

For each node, let $D = \sum \text{Deficit}$ where the sum is taken over all incoming edges.

Dijkstra-Scholten Algorithm for Cyclic Directed Graphs

```plaintext
begin
for J in 1..N loop
  if J /= First_Edge then
    Wait(S);
    Send_Signals(J);
    Signal(S);
  end if;
end loop;
Wait(S);
if N_Signals = 0 then
  if I /= 1 and First_Edge /= 0 then
    Send_Signals(First_Edge);
    First_Edge := 0;
  end if;
  Signal(S);
  return TRUE;
else
  return FALSE;
end if;
end Decide_to_Terminate;
```

Theorem. The following formulas are invariant in each node:

\[ D \geq 0 \]  \hspace{1cm} (1)
\[ N_{\text{Signals}} \geq 0 \]  \hspace{1cm} (2)
\[ D > 0 \vee N_{\text{Signals}} = 0 \]  \hspace{1cm} (3)

Proof: To prove (1), note that, the deficit is incremented when a new message is received and decremented when a signal is sent in response to a message. Hence, no signal is sent without a corresponding message and so $D \geq 0$.

$N_{\text{Signals}}$ keeps track of signals expected from the nodes to which the current node has sent messages. It is incremented whenever a new message is sent and decremented whenever a signal is received in reply to a message. Hence, it cannot be less than 0. This proves (2).

At any point of time, if $D > 0$, the node has sent more messages than received signals. In this case, $N_{\text{Signals}} > 0$ since the node is expecting more signals.

In the other case, if $D = 0$, the deficit is 0 and that means, either the node has not sent any messages (so it does not expect any signals), or all the outstanding signals have been received. In either case, $N_{\text{Signals}} = 0$. This proves (3).
Dijkstra-Scholten Algorithm for Cyclic Directed Graphs

An engaged process is one which has received a message and has not yet become quiescent. Processes may become engaged several times and processes that are disengaged don’t send messages, however, they continue to send and receive signals.

An engagement edge is a First Edge for some node.

Theorem. The following statements are invariant:

1. An engagement edge has non-zero deficit.
2. The engagement edges form a tree.
3. Every engaged node is reachable from the source node by a directed path formed from engagement edges.

Proof: An engagement edge is an edge of the underlying spanning-tree. To prove 1, note that a node sends signals on the First Edge just before becoming quiescent. So, if a node is still engaged, its engagement edge will have non-zero deficit.

The proof of 2 and 3, follows from the fact that the engagement edges form the spanning-tree.

Dijkstra-Scholten Algorithm for Cyclic Directed Graphs

Theorem. The safety and liveness properties are preserved when the program decides to terminate.

Proof: Since the source process decides the final termination, we have to prove that all other nodes except the source are quiescent at the moment the source decides to terminate. Since the root decides on termination after getting signals from all its children, it follows that all its children have become quiescent earlier. Arguing in this way, it can be proved that all the nodes are quiescent.

The liveness can be proved inductively. The basis of the induction is a leaf node of the spanning tree. If a leaf node terminates, it will signal along its First Edge to its parent.

Suppose, all the nodes at level $i$ have terminated. Then all of them send signals on their respective First Edges which will cause termination of the nodes at level $i-1$. So, ultimately the root (or the source node) will terminate.