CSE535 Asynchronous Systems
Termination Detection

YoungMin kwon

Termination Detection

- Why do we need it
  - Decide when the result of a distributed computation can be used
  - When a program is divided into multiple phases, the execution of a phase needs to be delayed until the previous phase is completed

Termination Detection

- A distributed computation is considered to be globally terminated if
  - Every process is locally terminated
  - There is no message in transit

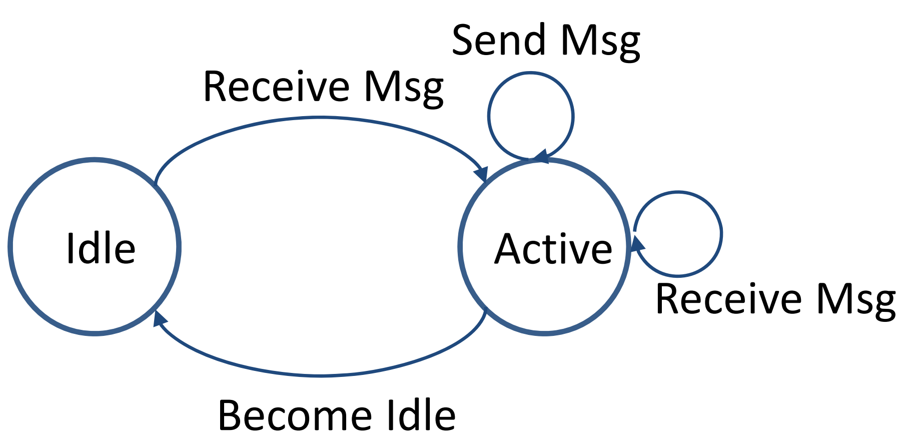
- Two distributed computations
  - The underlying computation (sending basic messages)
  - The termination detection algorithm (sending control messages)

System Model

- A process has two states
  - Active (busy): doing a local computation
  - Idle (passive): (temporarily) finished the execution and will be reactivated when receiving a message

System Model

- An active process can become idle at any time
- An idle process can become active only when receiving a message
- Only active processes can send messages
- Both active and idle processes can receive messages
- Sending a message and receiving a message are atomic actions.



System Model

- Definition of a termination detection

$$(\forall i :: p_i(t_0) = idle) \wedge (\forall i, j :: c_{i,j}(t_0) = 0)$$

  - $p_i(t)$: the state of process p_i at time t
  - $c_{i,j}(t)$: the number of messages in transit from p_i to p_j at time t

Termination Detection using Distributed Snapshot

- When a distributed computation terminates, there must be a unique process which became idle last.
  - When a process becomes idle, it takes a snap shot and issues a request to all processes to take a snapshot
  - If a process receives the request and agrees that the request is made after it became idle, it takes the snapshot.
  - A request is said to be successful if all processes have taken a snapshot for it.

Termination Detection using Distributed Snapshot

- Each process maintains a logical clock denoted by x
  - A process increments x by 1 each time it becomes idle
- Messages
  - B(x): a basic message sent at time x
  - R(x, i): a control message sent by i at time x
- Each process maintains a process id k such that
  - (x, k) is the maximum of the values (x', k') on all R(x', k') ever received or sent by the process
  - (x, k) > (x', k') iff (x > x') or (x = x' and k > k')

R1: When process i is active, it may send a basic message to process j at any time by doing

$$\text{send a } B(x) \textit{ to } j.$$

R2: Upon receiving a $B(x')$, process i does

$$\text{let } x := x' + 1;$$
$$\text{if}(i \text{ is } \textit{idle}) \rightarrow \text{ go } \textit{active}.$$

max(x,x')+1 ??

R3: When process i goes *idle*, it does

$$\text{let } x := x + 1;$$
$$\text{let } k := i;$$
$$\text{send message } R(x, k) \text{ to all other processes;}$$
$$\text{take a local snapshot for the request by } R(x, k).$$

R4: Upon receiving message $R(x', k')$, process i does

$$[((x', k') > (x, k)) \wedge (i \text{ is } \textit{idle}) \rightarrow \text{let}(x, k) := (x', k');$$
$$\text{take a local snapshot for the request by} R(x', k');$$
$$\square$$
$$((x', k') \leq (x, k)) \wedge (i \text{ is } \textit{idle}) \rightarrow \text{ do nothing;}$$
$$\square$$
$$(i \text{ is } \textit{active}) \rightarrow \text{ let } x := max(x', x)].$$

Termination Detection by Weight Throwing

- A process called controlling agent monitors the computation
  - Initially the weight at each process is 0 and the weight at the controlling agent is 1

- Computation starts when the controlling agent sends a basic message to a process
  - When a process sends a message it sends a part of its weight in the message
  - When a process receives a message it adds the weight in the message to its weight
  - When a process becomes idle, it sends its weight to the controlling agent.
  - The controlling agent concludes the termination if its weight becomes 1.

Termination Detection by Weight Throwing

Rule 1: The controlling agent or an active process may send a basic message to one of the processes, say P, by splitting its weight W into $W1$ and $W2$ such that $W1 + W2 = W$, $W1 > 0$ and $W2 > 0$. It then assigns its weight $W := W1$ and sends a basic message $B(DW := W2)$ to P.

Rule 2: On the receipt of the message $B(DW)$, process P adds DW to its weight W ($W := W + DW$). If the receiving process is in the idle state, it becomes active.

Rule 3: A process switches from the active state to the idle state at any time by sending a control message $C(DW := W)$ to the controlling agent and making its weight $W := 0$.

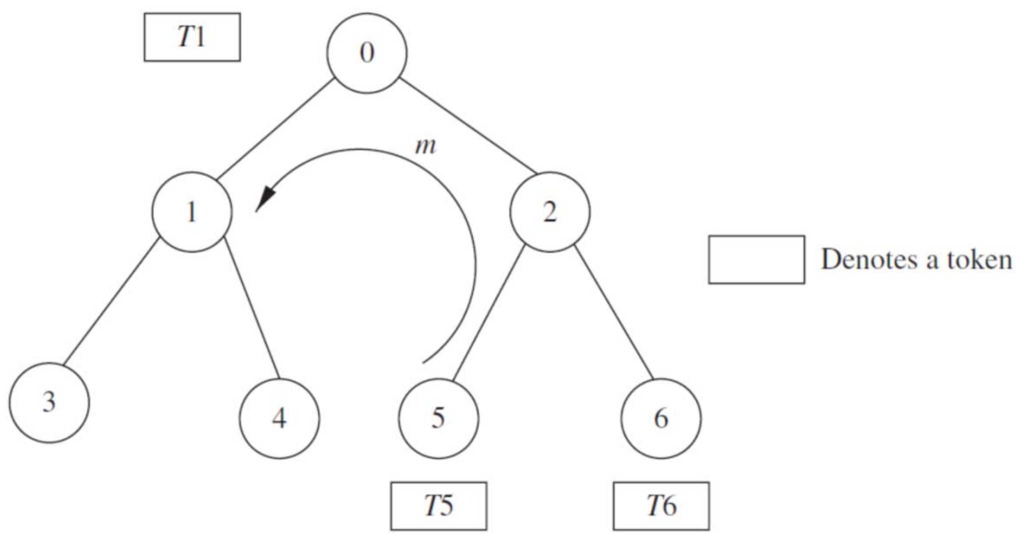
Rule 4: On the receipt of a message $C(DW)$, the controlling agent adds DW to its weight ($W := W + DW$). If $\mathrm{W} = 1$, then it concludes that the computation has terminated.

Spanning-Tree-Based Termination Detection Algorithm

- A simple (and incorrect) algorithm
  - Each leaf process is given a token and sends it to its parent when it becomes idle
  - Each intermediate process sends a token to its parent when it have received tokens from all of its children and when it becomes idle.
  - A root node concludes a termination when it receives tokens from all of its children.

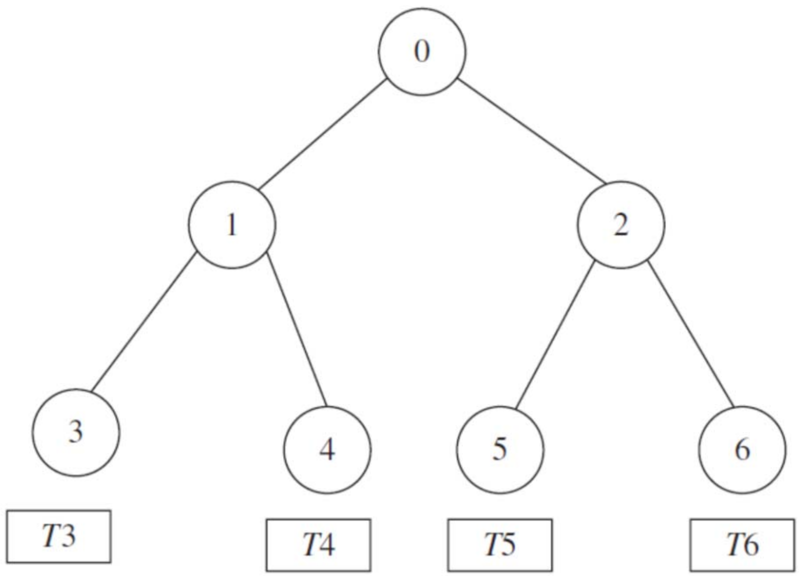
Spanning-Tree-Based Termination Detection Algorithm

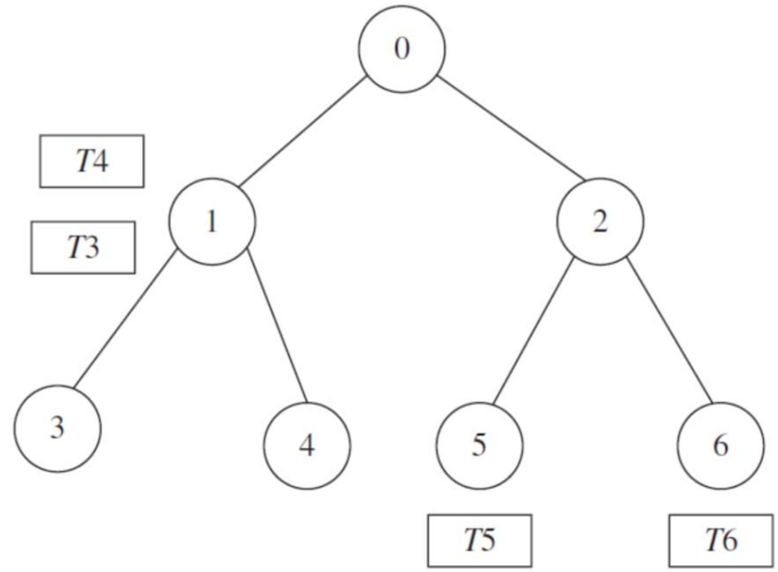
- A problem with the simple algorithm
  - After sending a token, a process can receive a message later and become active

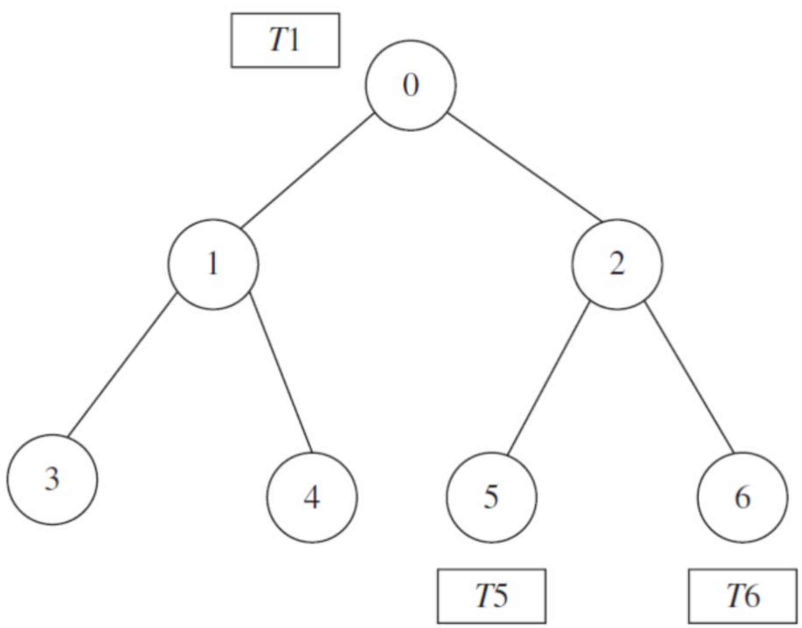


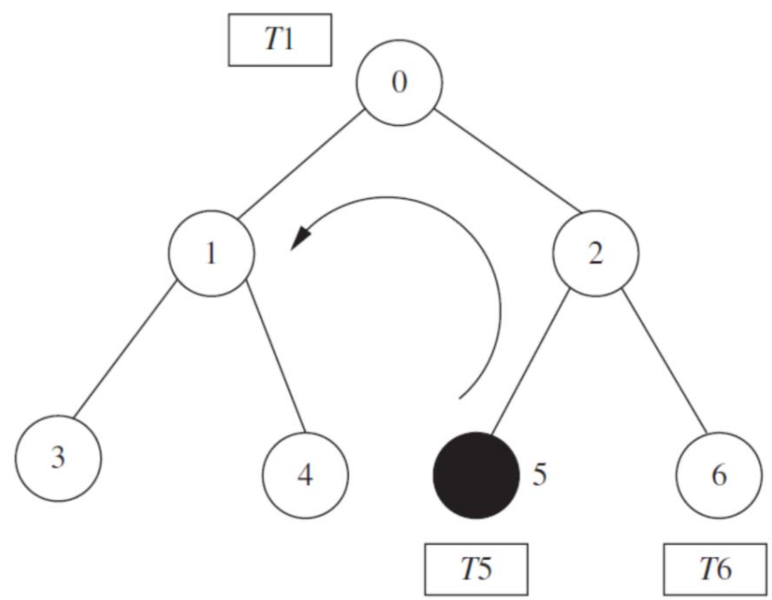
Spanning-Tree-Based Termination Detection Algorithm

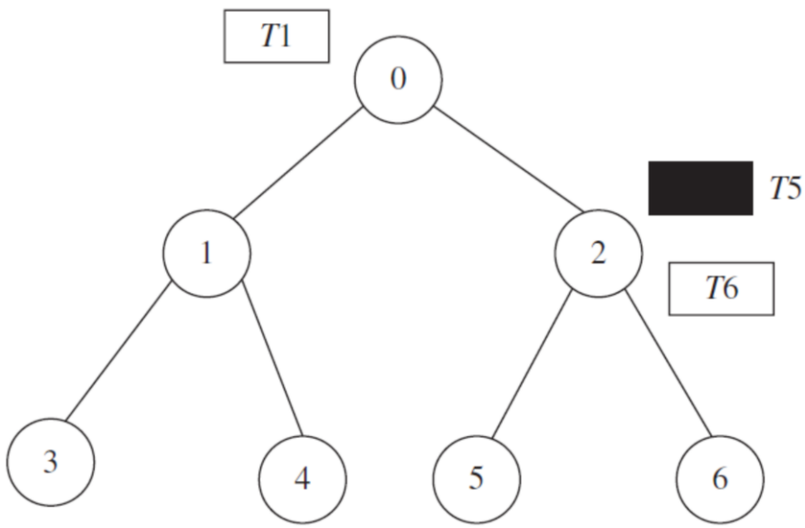
- All tokens and processes are initialized to white
- If a process send a message to other processes, it becomes black
  - After receiving tokens from all of its children and when it becomes idle, a process sends a token to its parent
  - White processes send a white token and black processes send a black token to its parent.
  - After sending a token, a process become white
- An intermediate node becomes black after sending a message or when it receives a black token.
- If a root node had received a black token from any of its children, restart the termination detection

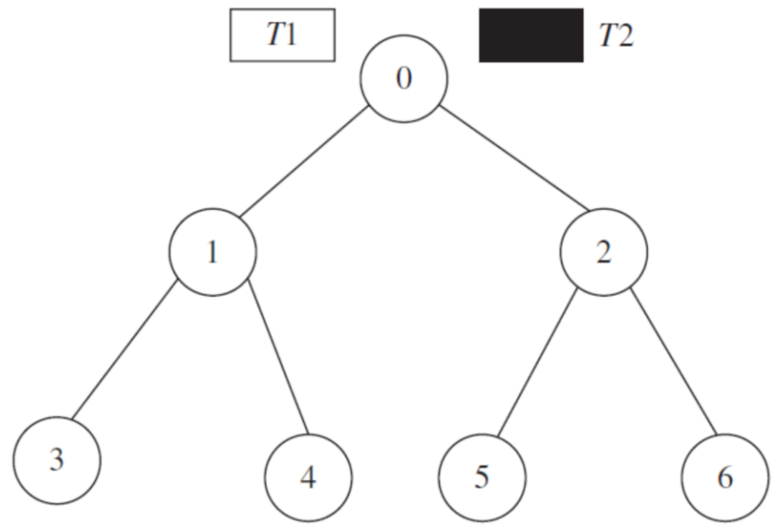

0
1
2
3
4
5
6
T3
T4
T5
T6


0
T4
T3
1
2
3
4
5
6
T5
T6


T1
0
1
2
3
4
5
6
T5
T6


T1
0
1
2
3
4
5
6
T5
T6


T1
T5
T6
0
1
2
3
4
5
6


T1
T2
0
1
2
3
4
5
6

Message-Optimal Termination Detection

- The spanning-tree-based termination detection algorithm is not efficient in terms of the message usage.
  - Sending any message can result in a restart of the termination detection process.
- Message-optimal termination detection
  - After receiving a message send an ack to the sender when the process becomes idle
  - A sender sends a token if it had received all ack messages expected and it becomes idle.

Message-Optimal Termination Detection

- Initially all nodes are in NDT (Not Detecting Termination) state and all links are uncolored
- Root node changes its state to DT (Detecting Termination) state and sends a warning message to all of its outgoing edges.
- When a node p receives a warning message from q,
  - It colors the incoming link (q, p)
  - If it is in state NDT, changes its state to DT, colors each of its outgoing edges, sends a warning message on each of its outgoing edges

Message-Optimal Termination Detection

- When p in DT state sends a message to q, push TO(q) on its local stack
- When x receives a message from y on the edge (y, x) colored by x, push FROM(y) on its local stack

Procedure *receive_message*(y: neighbor);
(* performed when a node x receives a message from its neighbor y on the link (y,x) that was colored by x *)

```
begin
    receive message from y on the link (y,x)
    if (link (y,x) has been colored by x) then
        push FROM(y) on the stack
end;
```

Message-Optimal Termination Detection

- When p becomes idle, it calls stack_cleanup

```
Procedure stack_cleanup;
begin
        while (top entry on stack is not of the form "TO()") do
            begin
                pop the entry on the top of the stack;
                let the entry be FROM(q);
                send a remove_entry message to q
            end
  end;
```

Message-Optimal Termination Detection

- When x receives remove_entry message from y, it calls receive_remove_entry

Procedure *receive_remove_entry*(*y*: neighbor);
(* performed when a node x receives a *remove_entry* message from its neighbor *y* *)
begin
scan the stack and delete the first entry of the form *TO*(*y*);
if idle **then**
stack_cleanup
end;

Message-Optimal Termination Detection

- A node sends a terminate message to its parent if
  - The process is in idle state
  - Each of its incoming link is colored (it has received a warning message on each of its incoming links)
  - Its stack is empty
  - For an intermediate node, it has received a terminate messages from each of its child