Chapter 7: Termination Detection

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Introduction

- A fundamental problem: To determine if a distributed computation has terminated.
- A non-trivial task since no process has complete knowledge of the global state, and global time does not exist.
- A distributed computation is globally terminated if every process is locally terminated and there is no message in transit between any processes.
- "Locally terminated" state is a state in which a process has finished its computation and will not restart any action unless it receives a message.
- In the termination detection problem, a particular process (or all of the processes) must infer when the underlying computation has terminated.

Introduction

- A termination detection algorithm is used for this purpose.
- Messages used in the underlying computation are called *basic* messages, and messages used for the purpose of termination detection are called *control* messages.
- A termination detection (TD) algorithm must ensure the following:
  1. Execution of a TD algorithm cannot indefinitely delay the underlying computation.
  2. The termination detection algorithm must not require addition of new communication channels between processes.

System Model

- At any given time, a process can be in only one of the two states: *active*, where it is doing local computation and *idle*, where the process has (temporarily) finished the execution of its local computation and will be reactivated only on the receipt of a message from another process.
- An active process can become idle at any time.
- An idle process can become active only on the receipt of a message from another process.
- Only active processes can send messages.
- A message can be received by a process when the process is in either of the two states, i.e., active or idle. On the receipt of a message, an idle process becomes active.
- The sending of a message and the receipt of a message occur as atomic actions.

Definition of Termination Detection

- Let $p_i(t)$ denote the state (active or idle) of process p_i at instant t.
- Let $c_{i,j}(t)$ denote the number of messages in transit in the channel at instant t from process p_i to process p_j.
- A distributed computation is said to be terminated at time instant t_0 iff: $(\forall i :: \ p_i(t_0) = \text{idle}) \wedge (\forall i, j :: \ c_{i,j}(t_0){=}0)$.
- Thus, a distributed computation has terminated iff all processes have become idle and there is no message in transit in any channel.

Termination detection Using Distributed Snapshots

- The algorithm assumes that there is a logical bidirectional communication channel between every pair of processes.
- Communication channels are reliable but non-FIFO. Message delay is arbitrary but finite.

Main idea:

- When a process goes from active to idle, it issues a request to all other processes to take a local snapshot, and also requests itself to take a local snapshot.
- When a process receives the request, if it agrees that the requester became idle before itself, it grants the request by taking a local snapshot for the request.
- A request is *successful* if all processes have taken a local snapshot for it.
- The requester or any external agent may collect all the local snapshots of a request.
- If a request is successful, a global snapshot of the request can thus be obtained and the recorded state will indicate termination of the computation,

Termination detection using distributed snapshots

Formal Description

- Each process *i* maintains a *logical clock* denoted by *x*, initialized to zero at the start of the computation.
- A process increments its *x* by one each time it becomes idle.
- A basic message sent by a process at its logical time *x* is of the form *B(x)*.
- A control message that requests processes to take local snapshot issued by process *i* at its logical time *x* is of the form *R(x, i)*.
- Each process synchronizes its logical clock *x* loosely with the logical clocks *x's* on other processes in such a way that it is the maximum of clock values ever received or sent in messages.
- A process also maintains a variable *k* such that when the process is idle, *(x,k)* is the maximum of the values *(x, k)* on all messages *R(x, k)* ever received or sent by the process.
- Logical time is compared as follows: $(x, k) > (x', k')$ iff $(x > x')$ or $((x=x')$ *and* $(k>k'))$, i.e., a tie between *x* and *x'* is broken by the process identification numbers *k* and *k'*.

Termination detection using distributed snapshots

The algorithm is defined by the following four rules.

- (R1): When process i is active, it may send a basic message to process j at any time by doing
  send a $B(x)$ *to* j.
- (R2): Upon receiving a $B(x')$, process i does
  let $x{:=}x'{+}1$;
  if (i is idle) $\rightarrow$ go active.
- (R3): When process i goes idle, it does
  let $x{:=}x{+}1$;
  let $k{:=}i$;
  send message $R(x, k)$ to all other processes;
  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 is idle)$\rightarrow$ let $(x,k){:=}(x', k')$;
  take a local snapshot for the request by $R(x', k')$;
  $\square$
  $((x', k') \leq (x,k)) \wedge (i \text{ is idle}) \rightarrow$ do nothing;
  $\square$
  (i is active)$\rightarrow$ let $x{:=}\max(x', x)]$.
- The last process to terminate will have the largest clock value. Therefore, every process will take a snapshot for it, however, it will not take a snapshot for any other process.

Termination detection by Weight Throwing

System Model

- A process called *controlling agent* monitors the computation.
- A communication channel exists between each of the processes and the controlling agent and also between every pair of processes.
  Initially, all processes are in the idle state.
- The weight at each process is zero and the weight at the controlling agent is 1.
- The computation starts when the controlling agent sends a basic message to one of the processes.
- A non-zero weight W ($0 < W \leq 1$) is assigned to each process in the active state and to each message in transit in the following manner:

Termination detection by Weight Throwing

Basic Idea

- When a process sends a message, it sends a part of its weight in the message.
- When a processreceives a message, it adds the weight received in the message to it's weight.
- Thus, the sum of weights on all the processes and on all the messages in transit is always 1.
- When a process becomes passive, it sends its weight to the controlling agent in a control message, which the controlling agent adds to its weight.
- The controlling agent concludes termination if its weight becomes 1.

Notations

- The weight on the controlling agent and a process is in general represented by W.
- B(DW) - a basic message B sent as a part of the computation, where DW is the weight assigned to it.
- C(DW) - a control message C sent from a process to the controlling agent where DW is the weight assigned to it.

Algorithm

The algorithm is defined by the following four rules:

- **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 W=1, then it concludes that the computation has terminated.

Correctness of Algorithm

Notations

- A: set of weights on all active processes
- B: set of weights on all basic messages in transit
- C: set of weights on all control messages in transit
- W_c: weight on the controlling agent.
- Two invariants I_1 and I_2 are defined for the algorithm:
- I_1: $W_c + \sum_{W \in (A \cup B \cup C)} W = 1$
- I_2: $\forall\ W \in (A \cup B \cup C),\ W > 0$

Correctness of Algorithm

- Invariant I_1 states that sum of weights at the controlling process, at all active processes, on all basic messages in transit, and on all control messages in transit is always equal to 1.
- Invariant I_2 states that weight at each active process, on each basic message in transit, and on each control message in transit is non-zero.
- Hence,
  $W_c{=}1$
  $\Rightarrow \sum_{W \in (A \cup B \cup C)} W = 0$ (by I_1)
  $\Rightarrow (A \cup B \cup C) = \phi$ (by I_2)
  $\Rightarrow (A \cup B) = \phi$.
- $(A \cup B) = \phi$ implies the computation has terminated. Therefore, the algorithm never detects a false termination.
  Further,
  $(A \cup B) = \phi$
  $\Rightarrow W_c + \sum_{W \in C} W = 1$ (by I_1)
- Since the message delay is finite, after the computation has terminated, eventually $W_c{=}1$.
- Thus, the algorithm detects a termination in finite time.

Spanning-Tree-Based Termination Detection Algorithm

- There are N processes P_i, $0 \leq i \leq N$, which are modeled as the nodes i, $0 \leq i \leq N$, of a fixed connected undirected graph.
- The edges of the graph represent the communication channels.
- The algorithm uses a fixed spanning tree of the graph with process P_0 at its root which is responsible for termination detection.
- Process P_0 communicates with other processes to determine their states through signals.
- All leaf nodes report to their parents, if they have terminated.
- A parent node will similarly report to its parent when it has completed processing and all of its immediate children have terminated, and so on.
- The root concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated.

Spanning-Tree-Based Termination Detection Algorithm

- Two waves of signals generated one moving inward and other outward through the spanning tree.
- Initially, a contracting wave of signals, called *tokens*, moves inward from leaves to the root.
- If this token wave reaches the root without discovering that termination has occurred, the root initiates a second outward wave of *repeat* signals.
- As this repeat wave reaches leaves, the token wave gradually forms and starts moving inward again, this sequence of events is repeated until the termination is detected.

A spanning-tree-based termination detection algorithm

- Initially, each leaf process is given a token.
- Each leaf process, after it has terminated sends its token to its parent.
- When a parent process terminates and after it has received a token from each of its children, it sends a token to its parent.
- This way, each process indicates to its parent process that the subtree below it has become idle.
- In a similar manner, the tokens get propagated to the root.
- The root of the tree concludes that termination has occurred, after it has become idle and has received a token from each of its children.

Spanning-Tree-Based Termination Detection Algorithm

A Problem with the algorithm

- This simple algorithm fails under some circumstances, when a process after it has sent a token to its parent, receives a message from some other process, which could cause the process to again become active (See Figure 1).
- Hence the simple algorithm fails since the process that indicated to its parent that it has become idle, is now active because of the message it received from an active process.
- Hence, the root node just because it received a token from a child, can't conclude that all processes in the child's subtree have terminated.

A spanning-tree-based termination detection algorithm

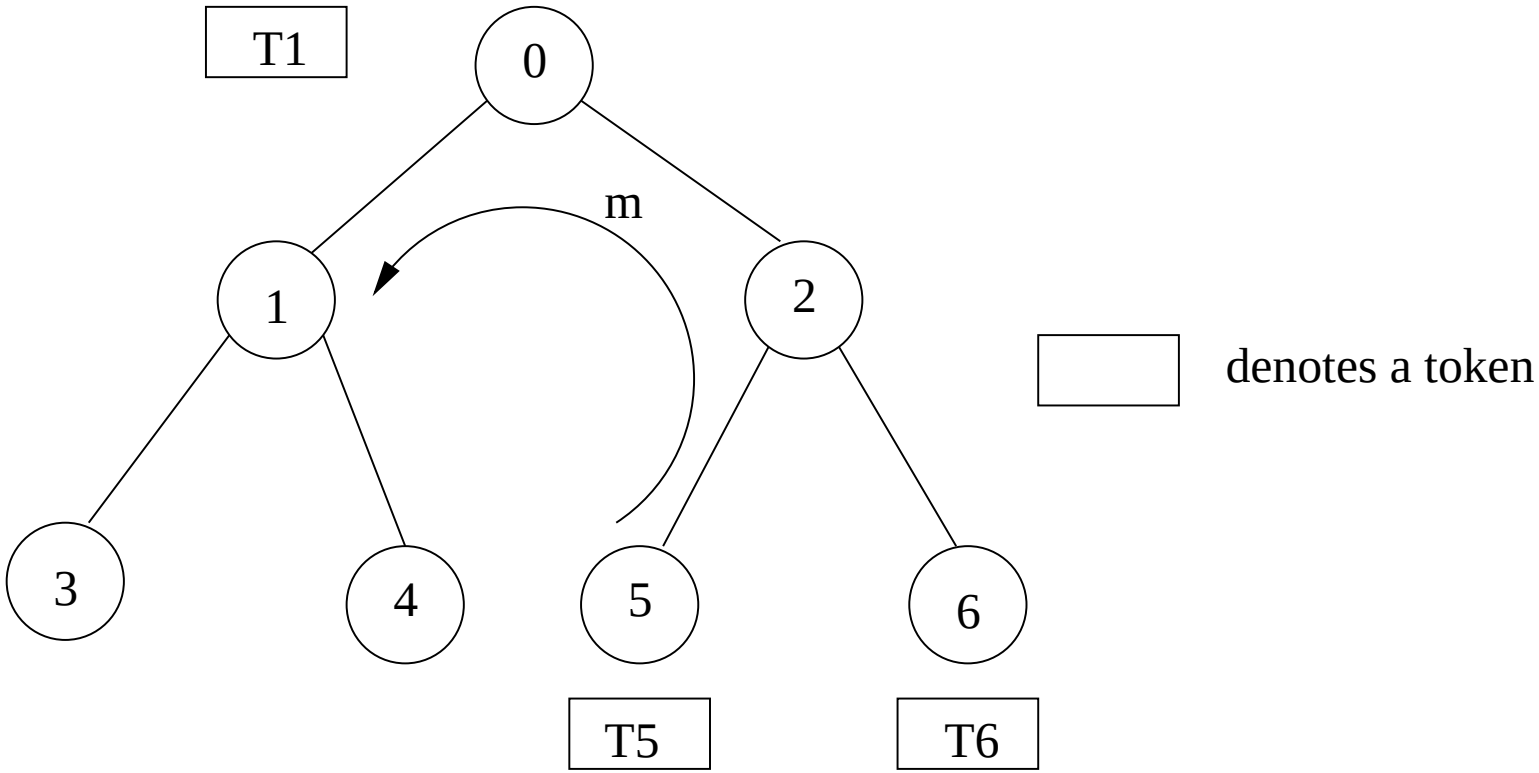

Figure 1: An Example of the Problem.

Spanning-Tree-Based Termination Detection Algorithm

- Main idea is to color the processes and tokens and change the color when messages such as in Figure 1 are involved.

The algorithm works as follows:

- Initially, each leaf process is provided with a token. The set S is used for book-keeping to know which processes have the token Hence S will be the set of all leaves in the tree.
- Initially, all processes and tokens are colored white.
- When a leaf node terminates, it sends the token it holds to its parent process.
- A parent process will collect the token sent by each of its children. After it has received a token from all of its children and after it has terminated, the parent process sends a token to its parent.
- A process turns black when it sends a message to some other process. When a process terminates, if its color is black, it sends a black token to its parent.
- A black process turns back to white, after it has sent a black token to its parent.

Spanning-Tree-Based Termination Detection Algorithm

- A parent process holding a black token (from one of its children), sends only a black token to its parent, to indicate that a message-passing was involved in its subtree.
- Tokens are propagated to the root in this fashion. The root, upon receiving a black token, will know that a process in the tree had sent a message to some other process. Hence, it restarts the algorithm by sending a Repeat signal to all its children.
- Each child of the root propagates the Repeat signal to each of its children and so on, until the signal reaches the leaves.
- The leaf nodes restart the algorithm on receiving the Repeat signal.
- The root concludes that termination has occurred, if
  1. it is white,
  2. it is idle, and
  3. it received a white token from each of its children.

Spanning-Tree-Based Termination Detection Algorithm

Performance

- The best case message complexity of the algorithm is O(N), where N is the number of processes in the computation, which occurs when all nodes send all computation messages in the first round.
- The worst case complexity of the algorithm is O(N*M), where M is the number of computation messages exchanged, which occurs when only computation message is exchanged every time the algorithm is executed.

Message-Optimal Termination Detection

- The network is represented by a graph $G = (V, E)$, where V is the set of nodes, and $E \subseteq V \times V$ is the set of edges or communication links.
- The communication links are bidirectional and exhibit FIFO property.
- The algorithm assumes the existence of a leader and a spanning tree in the network.
- If a leader is not available, the minimum spanning tree algorithm of Gallager et al. can be used to elect a leader and find a spanning tree using $O(|E| + |V| \log|V|)$ messages.
- Spanning-tree-based termination detection algorithm is inefficient in terms of message complexity because every message of the underlying computation can potentially cause the execution of one more round of the termination detection algorithm, resulting in significant message traffic.
- This is explained next.

Message-Optimal Termination Detection

- Consider the example shown in Figure 2.
- Suppose before node q receives message m, it has already sent a white token to its parent.
- Node p can not send a white token to its patent until node q becomes idle.
- To insure this, node p changes its color to black and sends a black token to its parent so that termination detection is performed again.

Message-Optimal Termination Detection

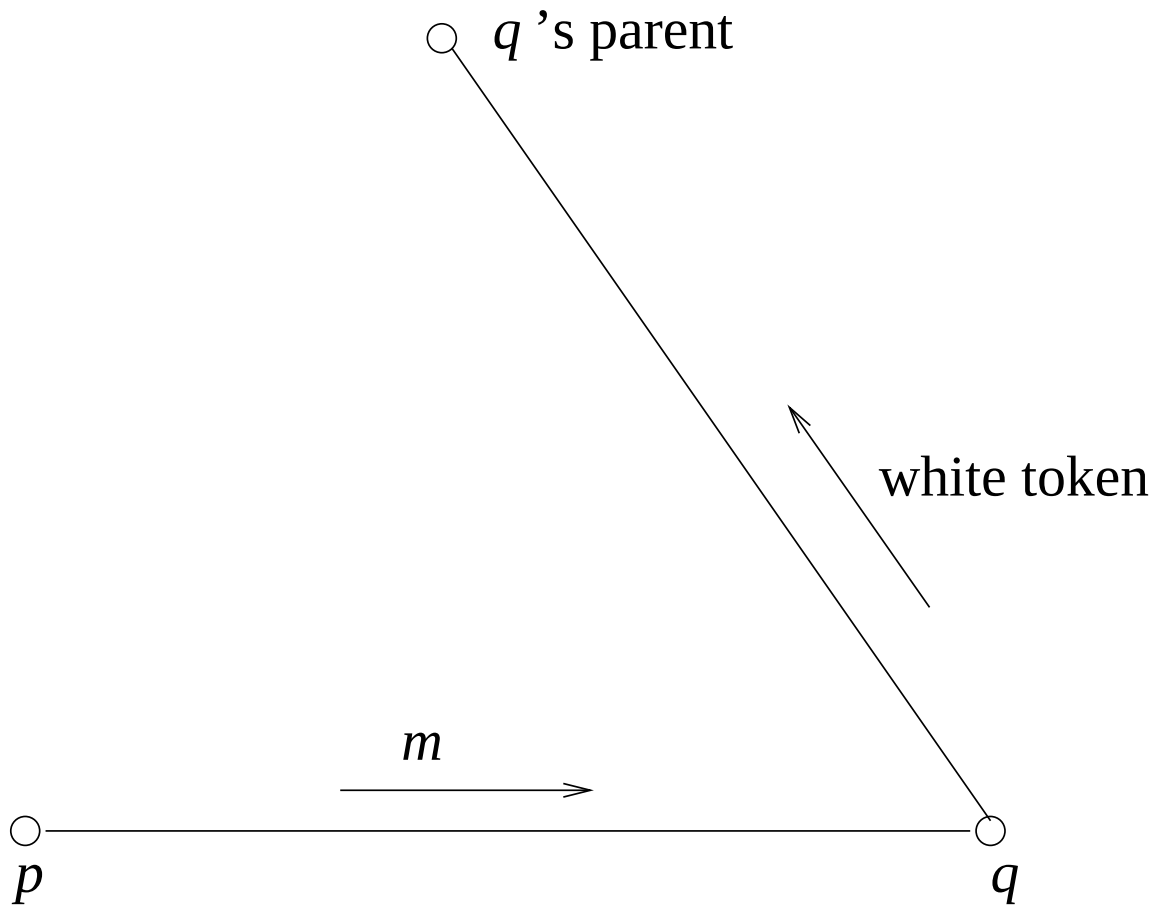

Figure 2: Node p sends a message m to node q that has already sent a white token to its parent.

Message-Optimal Termination Detection

. . . Formal Description of the Algorithm

- Initially, all nodes in the network are in state NDT (not detecting termination) and all links are uncolored.
- For termination detection,the root node changes its state to DT (detecting termination) and sends a warning message on each of its outgoing edges.
- When a node p receives a warning message from its neighbor, say q, it colors the incoming link (q, p) and if it is in state NTD, it changes its state to DT, colors each of its outgoing edges, and sends a warning message on each of its outgoing edges.
- When a node p in state DT sends a basic message to its neighbor q, it keeps track of this information by pushing the entry TO(q) on its local stack.
- When a node x receives a basic message from node y on the link (y, x) that is colored by x, node x knows that the sender node y will need an acknowledgement for this message from it.
- The receiver node x keeps track of this information by pushing the entry FROM(y) on its local stack.

Message-Optimal Termination Detection

. . . Formal Description of the Algorithm

- Procedure receive_message is given below:

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;

- When a node p becomes idle, it calls procedure stack_cleanup, which examines its stack from the top, and for every entry of the form FROM(q), it deletes the entry and sends the *remove_entry* message to node q.

Message-Optimal Termination Detection

Formal Description of the Algorithm

- Node p repeats this until it encounters an entry of the form TO(x) on the stack.
- The idea behind this step is to inform those nodes that sent a message to p that the actions triggered by their messages to p are complete.
- **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

- Node x on receipt of the control message *remove_entry* from node y, examines its stack from the top and deletes the first entry of the form TO(y) from the stack.
- If node x is idle, it also performs the *stack_cleanup* operation.
- The procedure *receive_remove_entry* is defined as follows:

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 when it satisfies all the following conditions:
  1. It is idle.
  2. Each of its incoming links is colored (it has received a warning message on each of its incoming links).
  3. Its stack is empty.
  4. It has received a *terminate* message from each of its children (this rule does not apply to leaf nodes).
- When the root node satisfies all of the above conditions, it concludes that the underlying computation has terminated.

Performance

- In the worst case, each node in the network sends one warning message on each outgoing link. Thus, each link carries two warning messages, one in each direction.
- Since there are |E| links, the total number of warning messages generated by the algorithm is 2*|E|.
- For every message generated by the underlying computation,exactly one *remove_message* is sent on the network.
- If M is the number of messages sent by the underlying computation, then at most M *remove_entry* messages are used.
- Finally, each node sends exactly one *terminate* message to its parent and since there are only |V| nodes and |V|−1 tree edges, only $|V| - 1$ *terminate* messages are sent.
- Hence, the total number of messages generated by the algorithm is 2* $|E| + |V| - 1 + M$.
- Thus, the message complexity of the algorithm is $O(|E| + M)$ as $|E| > |V| - 1$ for any connected network.
- The algorithm is asymptotically optimal in the number of messages.