Technical University of Cluj-Napoca
Computer Science Department

Distributed Systems

- Master in CS -

C6
Termination Detection (TD)

Fall 2017

Introduction

- Context
  - In DS, a problem is solved by the cooperation of a set of processes
  - In some applications, the problem to be solved is divided into many subproblems, and the execution of a subproblem cannot begin until the execution of the previous subproblem is complete
- Determining computing termination - a fundamental problem in DS

Introduction

- In a DS inferring if a DC has ended is essential
  - ⇒ the results produced by the computation can be used (may be in solving other problems)
- DC termination detection - a difficult problem
  - Problem complexity due to
    - » No process has complete knowledge of the global state, and
    - » Global time does not exist

Introduction (2)

- A DC is considered to be globally terminated if
  - every process is "**locally terminated"** and
  - there is no message in transit between any processes
- "locally terminated" state
  - a state in which a process has finished its computation and
  - will not restart any action unless it receives a message
- In the TD problem
  - a particular process (or all of the processes) must infer when the corresponding computation has terminated
- TD algorithms
  - infer if a certain DC has terminated

Introduction (3)

- Two distributed computations taking place in the distributed system
  - The *business logic computation* and
  - The TD *algorithm*
- Basic messages
  - Messages used in the business logic computation
- Control messages
  - Messages used by TD algorithms
- A TD algorithm must ensure
  - Execution of a TD algorithm cannot indefinitely delay the underlying computation (i.e. the execution of the TD algorithm must not freeze the business logic computation)
  - The TD algorithm must not require new communication channels between processes
- TD algorithms based on:
  - Distributed snapshot collection,
  - Weight throwing,
  - Spanning-tree

System Model (1)

- DC
  - Set of processes that communicate by message passing
  - All messages are received correctly after an arbitrary but finite delay
  - Communication is asynchronous
    - » i.e., a process never waits for the receiver to be ready before sending a message
  - Messages sent over the same communication channel may not obey the FIFO ordering

System Model (2)

- At any given time during execution of the DC, a process can be in only one of the two states:
  - Active (or busy)
    - » it is doing local computation
  - Idle (or passive)
    - » 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

System Model (3)

- Active processes becoming idle
  - It may happen at any time
  - Corresponds to the following situation
    - » the process has completed its local computation and
    - » has processed all received messages
- Idle processes becoming active
  - An idle process can become active only on the receipt of a message from another process

  => an idle process cannot spontaneously become active (except when the DC begins execution)
- Only active processes can send messages

System Model (4)

- Since we are not concerned with the initialization problem:
  - we assume that all processes are initially idle **and**
  - a message arrives from outside the system to start the computation
- 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

System Model (5)

- Constraint
  - We restrict our discussion to executions in which every process eventually becomes idle, although this property is in general undecidable
- If a TD algorithm is applied to a DC in which some processes remain in their active states forever

  => The TD algorithm itself will not terminate

System Model (6)

Definition of termination detection

- Notations
  - **$p_i(t)$** - the state (active or idle) of process **p_i** at instant **t** and
  - **$c_{i,j}(t)$** - 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_1 if:

  $(Vi, p_i(t_1) = idle) \wedge (Vi,j, c_{i,j}(t_1) = 0)$

Termination detection using distributed snapshots

- Fact (used by the algorithm):
  - A consistent snapshot of a distributed system captures stable properties
    - » Termination of a distributed computation is a stable property

  => If a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation
- Algorithm assumptions
  - 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

Termination detection using distributed snapshots
Informal description

- The main idea behind the algorithm
  - when a computation terminates, there must exist a unique process which became idle last
- The process that request termination detection test or any external agent may collect all the local snapshots of a request
- **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 said to be successful if
  - all processes have taken a local snapshot for it
- 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 in the recorded snapshot
  - In the recorded snapshot, all the processes are idle and there is no message in transit to any of the processes

Termination detection using distributed snapshots
Formal description (1)

- The algorithm needs logical time to order the requests
- Each process **i** maintains a logical clock denoted by **x**, which is 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
- Besides logical clock **x**, a process 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

Termination detection using distributed snapshots
Formal description (2)

- Logical time is compared as follows:

(x,k) > (x',k') if (x>x') or ((x=x') and (k>k'))

- In other words
  - a tie between **x** and **x'** is broken by the process identification numbers **k** and **k'**
- The algorithm is defined by four rules
  - The guarded statements are used to express conditions and actions
  - Each process **i** applies one of the rules whenever it is applicable

Termination detection using distributed snapshots
Formal description (3)

- **Rule R1** (process active, sending B messages)
  - When a process **i** is active, it may send a basic message to process **j** at any time by doing

    send a B(x) to j

- Rule R1 states:
  - When a process sends a basic message to any other process, it sends its logical clock value in the message

Termination detection using distributed snapshots
Formal description (4)

- **Rule R2** (receiving B messages)
  - Upon receiving a B(x'), process i executes

    let x := x' + 1

    if (i is idle) -> go active

- Rule R2 states:
  - when a process receives a basic message, it updates its logical clock based on the clock value contained in the message

Termination detection using distributed snapshots
Formal description (5)

- **Rule R3** (process going idle, sending R messages)
  - when process i goes idle it executes:

    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)

- Rule R3 states:
  - when a process becomes idle:
    - » updates its local clock
    - » sends a request for snapshot R(x, k) to every other process, and
    - » takes a local snapshot for this request

Termination detection using distributed snapshots
Formal description (6)

- **Rule 4 (Receiving R messages)**

  Upon receiving message R(x', k'), process i executes :

  if ((x', k') > (x, k) **and** (i is idle))
  let (x, k) := (x', k');
  take a local **snapshot** for the request by R(x', k');
  if ((x', k') ≤ (x, k) **and** (i is idle))
  do nothing;
  if (i is active)
  let x := max(x, x')

Termination detection using distributed snapshots
Formal Description (7)

- Rule R4 states:
  - **(1)** On the receipt of a message R(x', k'), the process takes a local snapshot if it is idle and (x', k') > (x, k), i.e., timing in the message is later than the local time at the process
    => the sender of R(x', k') terminated after this process
  - In this case, it is likely that the sender is the last process to terminate and thus, the receiving process takes a snapshot for it
  - Because of this action, every process will eventually take a local snapshot for the last request when the computation has terminated, that is, the request by the latest process to terminate will become successful
  - **(2)** if (x', k') ≤ (x, k)
    => the sender of R(x', k') terminated before this process
  - Hence, the sender of R(x', k') cannot be the last process to terminate
  - Thus, the receiving process does not take a snapshot for it.
  - **(3)** In the third case, the receiving process has not even terminated => The sender of R(x', k') cannot be the last process to terminate and no snapshot is taken.
- 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

- In this technique, a process called controlling agent monitors the computation
  - The controlling agent can be one of the processes involved in computation
- A communication channel exists
  - between each of the processes and the controlling agent and also
  - between every pair of processes

Termination detection by weight throwing
Basic idea

- 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
  - The process becomes active and the computation starts
- 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 way:
  - When a process sends a message, it sends a part of its weight in the message
  - When a process receives a message, it add the weight received in the message to its weight.

  => 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

Termination detection by weight throwing
Notations

- W - the weight on the controlling agent and a process
- B(DW) - a basic message B having DW as its assigned weight is sent as a part of the computation,
- C(DW) - a control message C having DW as its assigned weight is sent from a process to the controlling agent

Termination detection by weight throwing
Formal Description

- The algorithm is defined by the following four rules [Huang]:
- 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, where 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

Termination detection by weight throwing
Formal Description (2)

- 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

Termination detection by weight throwing
Formal Description (3)

- Algorithm correctness
- To prove the correctness of the algorithm, the following sets are defined:
  - 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

Termination detection by weight throwing
Formal Description (4)

- Two invariants I1 and I2 are defined for the algorithm:

$$I_1\colon W_c + \sum_{W \in (A \cup B \cup C)} W = 1.$$
$$I_2\colon \forall W \in (A \cup B \cup C),\ W > 0.$$

- Invariant I1 states:
  - The 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 I2 states:
  - The weight at each active process, on each basic message in transit, and on each control message in transit is non-zero.

Termination detection by weight throwing
Formal Description (5)

- As a result:

$$\begin{aligned} & W_c = 1 \\ & \Longrightarrow \textstyle\sum_{W \in (A \cup B \cup C)} W = 0 \text{ (by } I_1) \\ & \Longrightarrow (A \cup B \cup C) = \phi \text{ (by } I_2) \\ & \Longrightarrow (A \cup B) = \phi. \end{aligned}$$

- Note that (A U B) = 0 implies that the computation has terminated
- Therefore, the algorithm never detects a false termination.
- Further

$$\begin{aligned} & (A \cup B) = \phi \\ & \Longrightarrow W_c + \textstyle\sum_{W \in C} W = 1 \text{ (by } I_1). \end{aligned}$$

- Since the message delay is finite, after the computation has terminated, eventually $W_c = 1$. Thus, the algorithm detects a termination in finite time.

Termination detection based on spanning trees

- Assumptions
  - N processes P_i located in the nodes i, $0 \leq i \leq N$, of a fixed connected undirected graph
  - Graph edges represent the communication channels
- The algorithm uses a fixed spanning tree of the graph with process P_0 at the root
- Root node (process P_0)
  - Responsible for termination detection
  - Communicates with other processes to determine their states
  - The messages used for this purpose are called **signals**
  - Concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated
- Leaf nodes
  - All leaf nodes report to their parents, if they have terminated
- Interior node
  - Reports to its parent when
    - » it has completed processing and
    - » all of its immediate children have terminated

Termination detection based on spanning trees

- TD algorithm generates two waves of signals moving inward and outward through the spanning tree
- Initially, (as a request from the root) a contracting wave of **token** signals, 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

Termination detection based on spanning trees
Definitions

- Tokens
  - A contracting wave of signals that move inward from the leaves to the root
- Repeat signal
  - If a token wave fails to detect termination, node P0 initiates another round of termination detection by sending a signal called Repeat, to the leaves.
- Set S
  - The set of graph nodes having one or more tokens at any instant

Termination detection based on spanning trees
A simple (but incorrect) 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 way, 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

Termination detection based on spanning trees

A simple algorithm (incorrect)
Algorithm problem (1)

- After a process has sent its token to its parent, it should remain idle but this is not the case
- How the problem is generated:
  - A process p_i sent a token to its parent and then
  - Receives a message from some other process p_j
  - As a result, p_i becomes active again

  => The simple algorithm fails
- The algorithm has to be reworked to accommodate such message-passing scenarios

Termination detection based on spanning trees

A simple algorithm (incorrect)
Algorithm problem (2)

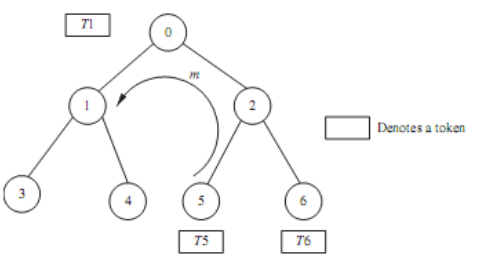


Figure 1 - Node 5 sends a message to node 1 (source [3])

Termination detection based on spanning trees

A simple algorithm (incorrect)
Algorithm problem (3)

- The problem is explained with the example shown in Figure 1
- Assume that process 1 has sent its token **T1** to its parent, process 0
- On receiving the token, process 0 concludes that process 1 and its children have terminated
- Process 0 if it is idle, can conclude that termination has occurred, whenever it receives a token from process 2. But now assume that just before process 5 terminates, it sends a message **m** to process 1
- On the reception of this message, process 1 becomes active again
- Thus, the information that process 0 has about process 1 (that it is idle) becomes void

Termination detection based on spanning trees

The correct algorithm

- The correct algorithm was developed by [Topor]
  - It works even when messages such as the one if Figure 1 are present
- The idea:
  - Color the processes and tokens and change the color when such messages are involved

Termination detection based on spanning trees

The correct algorithm (2)

- **Basic description**
- Use a coloring scheme
  - Enables the root node to know that a node in its children's subtree, that was assumed to be terminated, has become active due to a message
  - The root determines that an idle process has been activated by a message, based on the color of the token it receives from its children
- All tokens are initialized to white
- If a process had sent a message to some other process, it sends a black token to its parent on termination; otherwise, it sends a white token on termination
- The parent process on getting the black token knows that its child had sent a message to some other process

Termination detection based on spanning trees

The correct algorithm (3)

- The parent, when sending its token (on terminating) to its parent, sends a black token only if it received a black token from one of its children
  - This way, the parent's parent knows that one of the processes in its child's subtree had sent a message to some other process
- This actions gets propagated and finally the root node knows that message-passing was involved when it receives a black token from one of its children
- In this case, the root asks all nodes in the system to restart the termination detection
  - For this, the root sends a repeat signal to all other process
  - After receiving the repeat signal, all leaves will restart the termination detection algorithm

Termination detection based on spanning trees

The correct algorithm (4)

- **Algorithm description**

1. Initially
   - Each leaf process is provided with a token
   - All processes and tokens are white

   The set S is used for book-keeping to know which processes have the token

   => S will be the set of all leaves in the tree
2. When a leaf node terminates, it sends the token it holds to its parent process
3. 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

Termination detection based on spanning trees

The correct algorithm (5)

4. A process turns black when it sends a message to some other process
   - This coloring scheme helps a process remember that it has sent a message
   - When a process terminates, if it is black, it sends a black token to its parent
5. A black process turns back to white after it has sent a black token to its parent
6. A parent process holding a black token (from one of its children), sends a black token to its parent, to indicate that a message-passing was involved in its subtree

Termination detection based on spanning trees

The correct algorithm (6)

7. Tokens are propagated to the root in this way
   - The root, upon receiving a black token, knows 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
8. Each child of the root propagates the Repeat signal to each of its children and so on, until the signal reaches the leaves
10. The leaf nodes restart the algorithm on receiving the Repeat signal
11. The root concludes that termination has occurred, if:

(a) it is white;

(b) it is idle; and

(c) it has received a white token from each of its children

Termination detection based on spanning trees

The correct algorithm (7)

- **An example**

1. Initially, all nodes 0 to 6 are white (Figure 2). Leaf nodes 3, 4, 5, and 6 are each given a token. Node 3 has token T3, node 4 has token T4, node 5 has token T5, and node 6 has token T6. Hence, S is { 3, 4, 5, 6 }.
2. When node 3 terminates, it transmits T3 to node 1. Now S = {1, 4, 5, 6}. When node 4 terminates, it transmits T4 to node 1 (Figure 3). S = { 1,5,6 }
3. Node 1 has received a token from each of its children and, when it terminates, it transmits a token T1 to its parent (Figure 4). S = { 0,5,6 }

Termination detection based on spanning trees

The correct algorithm (8)

4. After this, suppose node 5 sends a message to node 1, causing node 1 to again become active (Figure 5).
5. Node 5 is colored black, since it sent a message to node 1.
6. When node 5 terminates, it sends a black token T5 to node 2. Now S = { 0, 2 , 6}. After sending its token, node 5 turns white (Figure 6). When node 6 terminates, it sends the white token T6 to node 2. S = {0, 2}
7. When node 2 terminates, it sends a black token T2 to node 0, since it holds a black token T5 from node 5 (Figure 7).

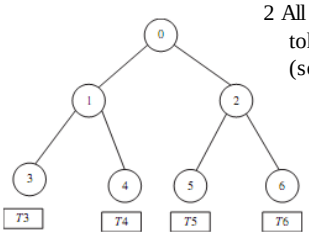
Termination detection based on spanning trees

The correct algorithm (9)

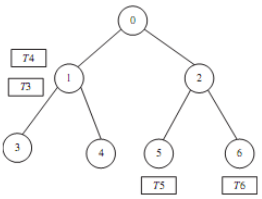
8. Since node 0 has received a black token T2 from node 2
   - It knows that there was a message sent by one or more of its children in the tree and hence
   - sends a repeat signal to each of its children
9. The repeat signal is propagated to the leaf nodes and the algorithm is repeated
10. Node 0 concludes that termination has occurred if it is white, it is idle, and it has received a white token from each of its children.

Termination detection based on spanning trees

The correct algorithm (10)



2 All leaf nodes have tokens S = { 3,4,5,6} (source [3])

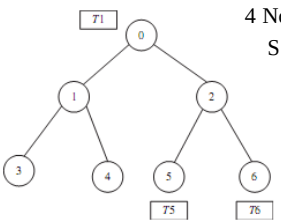


3 Nodes 3 and 4 become idle

S = { 1,5,6}

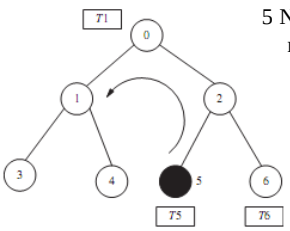
Termination detection based on spanning trees

The correct algorithm (11)



4 Node 1 becomes idle

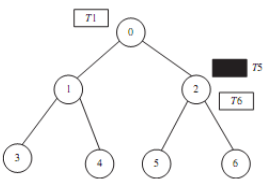
S = {0,5,6}



5 Node 5 sends a message to node 1

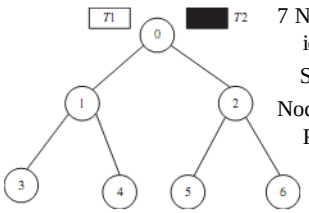
Termination detection based on spanning trees

The correct algorithm (12)



6 Nodes 5 and 6 become idle

S = {0,2}



7 Node 2 becomes idle

S = {0}

Node 0 initiates a Repeat signal

Termination detection based on spanning trees

The correct algorithm (13)

- **Performance**
- Best case message complexity of the algorithm is O(N), N is the number of processes in the computation
  - The best case occurs when all nodes send all computation messages in the first round

  ⇒In this case, the algorithm executes only twice and the message complexity depends only on the number of nodes
- Worst case complexity of the algorithm is O(N * M), M is the number of computation messages exchanged.
  - Worst case occurs when only computation message is exchanged every time the algorithm is executed

Termination detection in a general DC model

- Until now, we have assumed that the reception of a single message is enough to activate a passive process
- In the general model of distributed computing, a passive process does not necessarily become active on the receipt of a message
  - Instead, the **activation condition** of a passive process is more general and a passive process requires a set of messages to become active
  - This requirement is expressed by an activation condition defined over the set DS_i of processes from which a passive process P_i is expecting messages
  - The set DS_i associated with a passive process P_i is called the **dependent set** of P_i
- A passive process becomes active only when its activation condition is fulfilled

Termination detection in a general DC model
Model and Assumptions

- The distributed computation consists of a finite set of processes P_i (i = 1 .. n) interconnected by unidirectional communication channels
- Communication channels are reliable, but they do not obey FIFO property
- Message transfer delay is finite but unpredictable
- A passive process that has terminated its computation by executing for example an end or stop statement is said to be individually terminated
  - Its dependent set is empty and therefore, it can never be activated

Termination detection in a general DC model
Model and Assumptions

- **AND, OR, and AND-OR models**
- There are several request models, such as AND, OR, AND-OR models
- In the **AND** model, a passive process P_i can be activated only after a message from every process belonging to DS_i has arrived
- In the **OR** model, a passive process P_i can be activated when a message from any process belonging to DS_i has arrived
- In the **AND-OR** model, the requirement of a passive process P_i is defined by a set R_i of sets DS_i^1, DS_i^2 , ... DS_i^{qi} such that for all r, $1 \leq r \leq q_i$, $DS_i^r \subseteq P$
- The dependent set of P_i is

  $DS_i = DS_i^1 \cup DS_i^2 \cup ... DS_i^{qi}$
- Process P_i waits:
  - for messages from all processes belonging to DS_i^1 or
  - for messages from all processes belonging to DS_i^2 or
  - ...
  - for messages from all processes belonging to DS_i^{qi}

Termination detection in a general DC model
Model and Assumptions

- **The k out of n model**
- In the **k out of n** model, the requirement of a passive process P_i is defined by the set DS_i and an integer k_i , $1 \leq k_i \leq |DSi| = n_i$ and process P_i becomes active when it has received messages from k_i distinct processes in DS_i
- Note that a more general **k out of n** model can be constructed as disjunctions of several k out of n requests
- **Predicate fulfilled**
- To abstract the activation condition of a passive process P_i, a predicate $fulfilled_i(A)$ is introduced, where A is a subset of P
- Predicate $fulfilled_i(A)$ is true if and only if messages arrived (and not yet consumed) from all processes belonging to set A are sufficient to activate process P_i