Distributed Synchronization: outline

- Introduction

- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication

- Snapshots

This presentation is based on the book: “Distributed operating-systems & algorithms” by Randy Chow and Theodore Johnson
Distributed systems

A distributed system is a collection of independent computational nodes, communicating over a network, that is abstracted as a single coherent system.

- Grid computing
- Cloud computing (“infrastructure as a service”, “software as a service”)
- Peer-to-peer computing
- Sensor networks
- ...

A distributed operating system allows sharing of resources and coordination of distributed computation in a transparent manner.

(a), (b) – a distributed system
(c) – a multiprocessor
Distributed synchronization

- Underlies distributed operating systems and algorithms
- Processes communicate via message passing (no shared memory)
- Inter-node coordination in distributed systems challenged by
  - Lack of global state
  - Lack of global clock
  - Communication links may fail
  - Messages may be delayed or lost
  - Nodes may fail
- **Distributed synchronization** supports correct coordination in distributed systems
  - May no longer use shared-memory based locks and semaphores
Distributed Synchronization: outline

- Introduction

- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication

- Snapshots
Distributed computation model

- **Events**
  - Sending a message
  - Receiving a message
  - Timeout, internal interrupt

- **Processors send control messages to each other**
  - `send(destination, action; parameters)`

- **Processes may declare that they are waiting for events:**
  - `Wait for A_1, A_2, ..., A_n`
    - `A_1(source; parameters)`
      - code to handle `A_1`
    - .
    - .
    - `A_n(source; parameters)`
      - code to handle `A_n`
Causality and events ordering

- A distributed system has no global state nor global clock
  ➔ no global order on all events may be determined

- Each processor knows total order on events occurring in it

- There is a causal relation between the sending of a message and its receipt

Lamport's happened-before relation H

1. $e_1 <_p e_2 \Rightarrow e_1 <_H e_2$ (events within same processor are ordered)
2. $e_1 <_m e_2 \Rightarrow e_1 <_H e_2$ (each message m is sent before it is received)
3. $e_1 <_H e_2$ AND $e_2 <_H e_3 \Rightarrow e_1 <_H e_3$ (transitivity)

Leslie Lamport (1978): “Time, clocks, and the ordering of events in a distributed system”
Causality and events ordering (cont'd)

$P_1$  
$e_1$  
$e_4$  
$e_7$

$P_2$  
$e_2$  
$e_5$  
$e_6$

$P_3$  
$e_3$  
$e_8$

$e_1 \prec_H e_7$ ? Yes.  
$e_1 \prec_H e_3$ ? Yes.  
$e_5 \prec_H e_7$ ? No.  
$e_1 \prec_H e_8$ ? Yes.
Lamport's timestamp algorithm

1. Initially my_TS=0

2. Upon event e,

3. if e is the receipt of message m

4. my_TS=max(m.TS, my_TS)

5. my_TS++

6. e.TS=my_TS

7. If e is the sending of message m

8. m.TS=my_TS

To create a total order, ties are broken by process ID
Lamport's timestamps (cont'd)

\[\begin{array}{c}
P_1 \\
1.1 e_1 \\
2.1 e_4 \\
3.1 e_7 \\
P_2 \\
e_2 e_3 \\
1.2 2.2 \\
P_3 \\
e_6 e_8 \\
1.3 4.3 \\
\end{array}\]
Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
Vector timestamps - motivation

- Lamport's timestamps define a total order
  - $e_1 <_H e_2 \Rightarrow e_1.TS < e_2.TS$
  - However, $e_1.TS < e_2.TS \Rightarrow e_1 <_H e_2$ does not hold, in general. (concurrent events ordered arbitrarily)

**Definition:** Message $m_1$ **casually precedes** message $m_2$ (written as $m_1 <_c m_2$) if $s(m_1) <_H s(m_2)$ (sending $m_1$ happens before sending $m_2$)

**Definition:** causality violation occurs if $m_1 <_c m_2$ but $r(m_2) <_p r(m_1)$. In other words, $m_1$ is sent to processor $p$ before $m_2$ but is received after it.

- Lamport's timestamps do not allow to detect (hence nor prevent) causality violations.
Causality violation – an example

Causality violation between... $M_1$ and $M_3$. 
Vector timestamps

1. Initially my_VT = [0, ..., 0]

2. Upon event e,
   3. if e is the receipt of message m
   4. for i=1 to M
   5. my_VT[i] = max(m.VT[i], my_VT[i])

6. My_VT[self]++
7. e.VT = my_VT
8. if e is the sending of message m
9. m.VT = my_VT

For vector timestamps it does hold that: $e_1 <_V e_2 \iff e_1 <_H e_2$
An example of a vector timestamp

\[ e.VT = (3,6,4,2) \]
Comparison of vector timestamps

\[ e_1.VT = (5, 4, 1, 3) \]

\[ e_2.VT = (3, 6, 4, 2) \]

\[ e_3.VT = (0, 0, 1, 3) \]
VTs can be used to detect causality violations

Causality violation between... $M_1$ and $M_3$. 

Operating Systems, 2015, Meni Adler, Danny Hendler & Roie Zivan
Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
Preventing causality violations

A processor cannot control the order in which it receives messages... But it may control the order in which they are delivered to applications.

Protocol for FIFO message delivery (as in, e.g., TCP)
Preventing causality violations (cont'd)

- Senders attach a timestamp to each message

- **Destination delays the delivery of out-of-order messages**

- Hold back a message $m$ until we are assured that no message $m' <_H m$ will be delivered
  - For every other process $p$, maintain the earliest timestamp of a message $m$ that may be delivered from $p$
  - Do not deliver a message if an earlier message may still be delivered from another process

- **Algorithm assumes multicast communication**
## Algorithm for preventing causality violations

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textit{earliest}[1..M] initially [ &lt;1,0,...,0&gt;, &lt;0,1,...,0&gt;, ..., &lt;0,0,...,1&gt; ]</td>
</tr>
<tr>
<td>2</td>
<td>\textit{blocked}[1...M] initially [ {}, ..., {} ]</td>
</tr>
<tr>
<td>3</td>
<td>Upon the receipt of message ( m ) from processor ( p )</td>
</tr>
<tr>
<td>4</td>
<td>\textit{Delivery_list} = {}</td>
</tr>
<tr>
<td>5</td>
<td>If (\textit{blocked}[p] is empty)</td>
</tr>
<tr>
<td>6</td>
<td>\textit{earliest}[p] = ( m ).timestamp</td>
</tr>
<tr>
<td>7</td>
<td>Add ( m ) to the tail of \textit{blocked}[p]</td>
</tr>
</tbody>
</table>
| 8    | While (\exists k such that \textit{blocked}[k] is non-empty AND  
|      | \( \forall i \in \{1,...,M\} \) (except \( k \) and Self) not_earlier(earliest[i], earliest[k])  
|      | remove the message at the head of \textit{blocked}[k], put it in \textit{delivery\_list} |
| 9    | if \textit{blocked}[k] is non-empty  
| 10   | \textit{earliest}[k] \leftarrow \( m' \).timestamp, where \( m' \) is at the head of \textit{blocked}[k] |
| 11   | else |
| 12   | increment the \( k \)'th element of earliest[k] |
| 13   | End While |
| 14   | Deliver the messages \textit{in delivery\_list}, in causal order |

For each processor \( p \), the earliest timestamp with which a message from \( p \) may still be delivered
Algorithm for preventing causality violations

1. `earliest[1..M]` initially live in `[<1,0,...,0>, <0,1,...,0>, ..., <0,0,...,1>]`
2. `blocked[1..M]` initially live in `[,] ..., [ ]`

3. Upon the receipt of message `m` from processor `p`
4. `Delivery_list = {}`
5. If (`blocked[p]` is empty)
6. `earliest[p] = m.timestamp`
7. Add `m` to the tail of `blocked[p]`
8. While (∃ `k` such that `blocked[k]` is non-empty AND
7. `∀ i ∈ {1,...,M}` (except `k` and Self) not_earlier(`earliest[i], earliest[k]`)`
9. remove the message at the head of `blocked[k]`, put it in `delivery_list`
10. if `blocked[k]` is non-empty
11. `earliest[k] ← m'.timestamp`, where `m'` is at the head of `blocked[k]`
12. else
13. increment the k'th element of earliest[k]
14. End While
15. Deliver the messages in `delivery_list`, in causal order

For each process `p`, the messages from `p` that were received but were not delivered yet.
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially \[ \langle 1,0,...,0 \rangle, \langle 0,1,...,0 \rangle, ..., \langle 0,0,...,1 \rangle \]
2. $blocked[1...M]$ initially \[ \{\}, ..., \{\} \]

3. Upon the receipt of message $m$ from processor $p$

4. $Delivery\_list = \{\}$

5. If ($blocked[p]$ is empty)

6. \[earliest[p] = m.\text{timestamp}\]

7. Add $m$ to the tail of $blocked[p]$

8. While ($\exists k$ such that $blocked[k]$ is non-empty AND \[\forall i \in \{1,\ldots,M\} \text{ (except } k \text{ and Self)} \not \text{ not_earlier}(earliest[i], earliest[k])\])

9. remove the message at the head of $blocked[k]$, put it in $delivery\_list$

10. if $blocked[k]$ is non-empty

11. \[earliest[k] \leftarrow m'.\text{timestamp}, \text{ where } m' \text{ is at the head of } blocked[k]\]

12. else

13. increment the $k$'th element of $earliest[k]$

14. End While

15. Deliver the messages in $delivery\_list$, in causal order
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially $[<1,0,...,0>, <0,1,...,0>, ..., <0,0,...,1>]$
2. $blocked[1...M]$ initially $[{}, ..., {}]$

3. Upon the receipt of message $m$ from processor $p$

4. $Delivery\_list = {}$

5. If ($blocked[p]$ is empty)

6. $earliest[p]=m.timestamp$

7. Add $m$ to the tail of $blocked[p]$

8. While ($\exists k$ such that $blocked[k]$ is non-empty AND $\forall i \in \{1,...,M\}$ (except $k$ and Self) not_earlier($earliest[i], earliest[k]$))

9. remove the message at the head of $blocked[k]$, put it in $delivery\_list$

10. if $blocked[k]$ is non-empty

11. $earliest[k] \leftarrow m'.timestamp$, where $m'$ is at the head of $blocked[k]$

12. else

13. increment the $k$'th element of $earliest[k]$

14. End While

15. Deliver the messages in $delivery\_list$, in causal order
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially $[<1,0,...,0>, <0,1,...,0>, ..., <0,0,...,1>]$
2. $blocked[1..M]$ initially $[{}, ..., {}]$

3. Upon the receipt of message $m$ from processor $p$
4. $Delivery_list = {}$
5. If ($blocked[p]$ is empty)
6. \hspace{1em} $earliest[p] = m$.timestamp
7. \hspace{1em} Add $m$ to the tail of $blocked[p]$
8. While ($\exists k$ such that $blocked[k]$ is non-empty AND $\forall i \in \{1,...,M\}$ (except $k$ and Self) not_earlier($earliest[i], earliest[k]$))
9. \hspace{1em} remove the message at the head of $blocked[k]$, put it in $delivery_list$
10. if $blocked[k]$ is non-empty
11. \hspace{1em} $earliest[k] \leftarrow m'$.timestamp, where $m'$ is at the head of $blocked[k]$
12. else
13. \hspace{1em} increment the $k$'th element of $earliest[k]$
14. End While
15. Deliver the messages in $delivery_list$, in causal order

$m$ is now the most recent message from $p$ not yet delivered
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially \[ <1,0,...0>, <0,1,...,0>, ..., <0,0,...,1> \]
2. $blocked[1...M]$ initially \[ {}, ..., {} \]

3. Upon the receipt of message $m$ from processor $p$
4. $Delivery\_list = {}$
5. If ($blocked[p]$ is empty)
   6. $earliest[p]=m$.timestamp
   7. Add $m$ to the tail of $blocked[p]$
8. While ($\exists k$ such that $blocked[k]$ is non-empty AND
    \[ \forall i \in \{1,...,M\} (\text{except } k \text{ and Self}) \text{ not}_\text{earlier}(earliest[i], earliest[k]) \] )
   9. remove the message at the head of $blocked[k]$, put it in $delivery\_list$
10. if $blocked[k]$ is non-empty
    11. $earliest[k] \leftarrow m'$.timestamp, where $m'$ is at the head of $blocked[k]$
12. else
13. increment the $k$'th element of $earliest[k]$
14. End While
15. Deliver the messages in $delivery\_list$, in causal order
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially $[<1,0,...,0>, <0,1,...,0>, ..., <0,0,...,1>]$
2. $blocked[1..M]$ initially $[{}, ..., {}]$

Upon the receipt of message $m$ from processor $p$

3. $Delivery_list = {}$
4. If ($blocked[p]$ is empty)
5.   $earliest[p] = m$.timestamp
6.   Add $m$ to the tail of $blocked[p]$
8. While ($\exists k$ such that $blocked[k]$ is non-empty AND $\forall i \in \{1,...,M\}$ (except $k$ and Self) notEarlier($earliest[i]$, $earliest[k]$))
9.   remove the message at the head of $blocked[k]$, put it in $delivery_list$
10. if $blocked[k]$ is non-empty
11.   $earliest[k] \leftarrow m'$.timestamp, where $m'$ is at the head of $blocked[k]$
12. else
13.   increment the k'th element of $earliest[k]$
14. End While
15. Deliver the messages in $delivery_list$, in causal order
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially: $[<1,0,...,0>, <0,1,...,0>, ..., <0,0,...,1>]$

2. $blocked[1...M]$ initially: $[{}, ..., {}]$

3. Upon the receipt of message $m$ from processor $p$

4. $Delivery_list = {}$

5. If $(blocked[p]$ is empty)

6. $earliest[p]=m.timestamp$

7. Add $m$ to the tail of $blocked[p]$

8. While ($\exists k$ such that $blocked[k]$ is non-empty AND $\forall i \in \{1,...,M\}$ (except $k$ and Self) not_earlier($earliest[i], earliest[k]$))

9. remove the message at the head of $blocked[k]$, put it in $delivery_list$

10. if $blocked[k]$ is non-empty

11. $earliest[k] \leftarrow m'.timestamp$, where $m'$ is at the head of $blocked[k]$

12. else

13. increment the $k$'th element of $earliest[k]$

14. End While

15. Deliver the messages in $delivery_list$, in causal order

If there are additional blocked messages of $k$, update $earliest[k]$ to be the timestamp of the earliest such message.
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially $[<1,0,...,0>, <0,1,...,0>, ..., <0,0,...,1>]$
2. $blocked[1...M]$ initially $[\{\}, \ldots, \{\}]$
3. Upon the receipt of message $m$ from processor $p$
4. $Delivery\_list = \{\}$
5. If ($blocked[p]$ is empty)
6. $earliest[p] = m$.timestamp
7. Add $m$ to the tail of $blocked[p]$
8. While ($\exists k$ such that $blocked[k]$ is non-empty AND
   $\forall i \in \{1,...,M\}$ (except $k$ and Self) not_earlier($earliest[i], earliest[k]$)
9. remove the message at the head of $blocked[k]$, put it in $delivery\_list$
10. if $blocked[k]$ is non-empty
11. $earliest[k] \leftarrow m'.timestamp$, where $m'$ is at the head of $blocked[k]$
12. else
13. increment the k'th element of earliest[k]
14. End While
15. Deliver the messages in $delivery\_list$, in causal order
Algorithm for preventing causality violations

1. $earliest[1..M]$ initially $[<1,0,...,0>, <0,1,...,0>, ..., <0,0,...,1>]$
2. $blocked[1..M]$ initially $[{}, ..., {}]$
3. Upon the receipt of message $m$ from processor $p$
4. $Delivery_list = {}$
5. If ($blocked[p]$ is empty)
6.  $earliest[p] = m$.timestamp
7.  Add $m$ to the tail of $blocked[p]$
8. While ($\exists k$ such that $blocked[k]$ is non-empty AND 
   $\forall i \in \{1,...,M\}$ (except $k$ and Self) not_earlier($earliest[i], earliest[k]$)
   remove the message at the head of $blocked[k]$, put it in $delivery_list$
9. if $blocked[k]$ is non-empty
10.  $earliest[k] \leftarrow m'$.timestamp, where $m'$ is at the head of $blocked[k]$
11. else
12.  increment the $k$'th element of $earliest[k]$
13. End While
14. Deliver the messages in $delivery_list$, in causal order
15. Finally, deliver set of messages that will not cause causality violation (if there are any).
Execution of the algorithm as multicast

![Diagram showing the execution of the algorithm as multicast]

Since the algorithm is “interested” only in causal order of sending events, vector timestamp is incremented only upon send events.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1,0,0)</td>
<td>(0,1,0)</td>
<td>(0,0,1)</td>
</tr>
</tbody>
</table>

Upon receipt of $m_2$

- $(1,1,0)$ $m_2$  $(0,1,0)$  $(0,0,1)$

Upon receipt of $m_1$

- $(1,1,0)$ $m_2$  $(0,1,0)$  $m_1$  $(0,0,1)$
  - deliver $m_1$
- $(1,1,0)$ $m_2$  $(0,2,0)$  $(0,0,1)$
  - deliver $m_2$
- $(2,1,0)$  $(0,2,0)$  $(0,0,1)$
Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
Snapshots motivation – phantom deadlock

- Assume we would like to implement a distributed deadlock-detector
- We record process states to check if there is a waits-for-cycle

P1 \rightarrow r1 \rightarrow request \rightarrow OK \rightarrow request \rightarrow release \rightarrow OK

r1 \rightarrow r2 \rightarrow request \rightarrow OK \rightarrow request

P2 \rightarrow r2 \rightarrow request

Observed waits-for graph

Actual waits-for graph
What is a snapshot?

- **Global system state:**
  - $S=(s_1, s_2, \ldots, s_M)$ – local processor states
  - The contents $L_{i,j}=(m_1, m_2, \ldots, m_k)$ of each communication channel $C_{i,j}$ (channels assumed to be FIFO)
    These are messages sent but not yet received

- **Global state must be consistent**
  - If we observe in state $s_i$ that $p_i$ received message $m$ from $p_k$, then in observation $s_k$, $k$ must have sent $m$.
  - Each $L_{i,j}$ must contain exactly the set of messages sent by $p_i$ but not yet received by $p_j$, as reflected by $s_i$, $s_j$.

- **Snapshot state must be consistent**, one that might have existed during the computation.
- **Observations must be mutually concurrent** that is, no observation casually precedes another observation (a consistent cut)
Snapshot algorithm – informal description

- Upon joining the algorithm, a process records its local state

- The process that initiates the snapshot sends **snapshot tokens** to its neighbors (before sending any other messages)
  - Neighbors send them to their neighbors – broadcast

- Upon receiving a snapshot token:
  - a process records its state prior to sending/receiving additional messages
  - Must then send tokens to all its other neighbors

- How shall we record sets $L_{p,q}$?
  - $q$ receives token from $p$ and that is the first time $q$ receives token:
    $L_{p,q} = \{ \}$
  - $q$ receives token from $p$ but $q$ received token before:
    $L_{p,q} = \{ \text{all messages received by } q \text{ from } p \text{ since } q \text{ received token} \}$
## Snapshot algorithm – data structures

- The algorithm supports **multiple ongoing snapshots** – one per process

- Different snapshots from same process distinguished by version number

<table>
<thead>
<tr>
<th>Per process variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>integer my_version</strong> initially 0</td>
</tr>
<tr>
<td><strong>integer current_snap[1..M]</strong> initially [0,...,0]</td>
</tr>
<tr>
<td><strong>integer tokens_received[1..M]</strong></td>
</tr>
<tr>
<td><strong>processor_state S[1...M]</strong></td>
</tr>
<tr>
<td><strong>channel_state [1..M][1..M]</strong></td>
</tr>
</tbody>
</table>

The version number of my current snapshot
Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

### Per process variables

- **integer** `my_version` initially 0
- **integer** `current_snap[1..M]` initially [0,...,0]
- **integer** `tokens_received[1..M]`
- **processor_state** `S[1...M]`
- **channel_state** `[1..M][1..M]`

`current_snap[r]` contains version number of snapshot initiated by processor `r`
Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

**Per process variables**

integer *my_version* initially 0

integer *current_snap*[1..M] initially [0,...,0]

integer *tokens_received*[1..M]

*processor_state* S[1...M]

*channel_state* [1..M][1..M]

*tokens_received*[r] contains the number of tokens received for the snapshot initiated by processor r
Snapshot algorithm – data structures

☐ The algorithm supports multiple ongoing snapshots – one per process

☐ Different snapshots from same process distinguished by version number

Per process variables

integer my_version initially 0

integer current_snap[1..M] initially [0,...,0]

integer tokens_received[1..M]

processor_state S[1...M]

channel_state [1..M][1..M]

processor_state[r] contains the state recorded for the snapshot of processor r
## Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

<table>
<thead>
<tr>
<th><strong>Per process variables</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>integer <em>my_version</em> initially 0</td>
</tr>
<tr>
<td>integer <em>current_snap</em>[1..M] initially [0,...,0]</td>
</tr>
<tr>
<td>integer <em>tokens_received</em>[1..M]</td>
</tr>
<tr>
<td><em>processor_state</em> S[1...M]</td>
</tr>
</tbody>
</table>

*channel_state*[r][q] records the state of channel from q to current processor for the snapshot initiated by processor r
Snapshot algorithm – pseudo-code

execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
1. \( my\_version++ , \) current_snap[self]=my_version
2. \( S[self] \leftarrow my\_state \)
3. for each outgoing channel q, send(q, TOEKEN, my_version)
4. tokens_received[self]=0

TOKEN(q; r,version):
5. If current_snap[r] < version
6. \( S[r] \leftarrow my\_state \)
7. current_snap[r]=version
8. L[r][q] \leftarrow empty, send token(r, version) on each outgoing channel
9. tokens_received[r] \leftarrow 1
10. else
11. tokens_received[r]++
12. L[r][q] \leftarrow all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for (r,version) is finished
Snapshot algorithm – pseudo-code

execute_snapshot()

\textit{Wait for a snapshot request or a token}

\textbf{Snapshot request:}
\begin{enumerate}
\item \texttt{my\_version++}, current\_snap[self]=my\_version
\item \texttt{S[self]} $\leftarrow$ my\_state
\item \texttt{for each outgoing channel q, send(q, TOEKEN, my\_version)}
\item \texttt{tokens\_received[self]=0}
\end{enumerate}

\texttt{TOKEN(q; r,version):}
\begin{enumerate}
\item If current\_snap[r] < version
\item \texttt{S[r]} $\leftarrow$ my state
\item \texttt{current\_snap[r]=version}
\item \texttt{L[r][q]} $\leftarrow$ empty, send token(r, version) on each outgoing channel
\item \texttt{tokens\_received[r]} $\leftarrow$ 1
\item else
\item \texttt{tokens\_received[r]++}
\item \texttt{L[r][q]} $\leftarrow$ all messages received from q since first receiving token(r,version)
\item if tokens\_received = #incoming channels, local snapshot for (r,version) is finished
\end{enumerate}
Snapshot algorithm – pseudo-code

execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:

1. my_version++, current_snap[self]=my_version
2. S[self] ← my_state
3. for each outgoing channel q, send(q, TOEKEN, my_version)
4. tokens_received[self]=0

TOKEN(q; r,version):
5. If current_snap[r] < version
6. S[r] ← my state
7. current_snap[r]=version
8. L[r][q] ← empty, send token(r, version) on each outgoing channel
9. tokens_received[r] ← 1
10. else
11. tokens_received[r]++
12. L[r][q] ← all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for (r,version) is finished
Snapshot algorithm – pseudo-code

execute_snapshot()

*Wait for a snapshot request or a token*

Snapshot request:
1. \( my\_version++ \), current_snap[self]=my_version
2. \( S[self] \leftarrow\) my_state
3. *for each outgoing channel q, send(q, TOEKEN; self, my_version)*
4. \( \text{tokens\_received}[self]=0 \)

\( \text{TOKEN}(q; r,\text{version}): \)
5. If current_snap[r] < version
6. \( S[r] \leftarrow\) my state
7. current_snap[r]=version
8. \( L[r][q] \leftarrow\) empty, send token(r, version) on each outgoing channel
9. tokens_received[r] \( \leftarrow\) 1
10. else
11. tokens_received[r]++
12. \( L[r][q] \leftarrow\) all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for \((r,\text{version})\) is finished

Send snapshot-token on all outgoing channels, initialize number of received tokens for my snapshot to 0
**Snapshot algorithm – pseudo-code**

```plaintext
execute_snapshot()

*Wait for a snapshot request or a token*

**Snapshot request:**
1. `my_version++`, `current_snap[self]=my_version`
2. `S[self] ← my_state`
3. *for each outgoing channel q, send(q, TOEKEN; self, my_version)*
4. `tokens_received[self]=0`

**TOKEN(q; r,version):**
5. If `current_snap[r] < version`
6. `S[r] ← my_state`
7. `current_snap[r]=version`
8. `L[r][q] ← empty, send token(r, version) on each outgoing channel`
9. `tokens_received[r] ← 1`
10. else
11. `tokens_received[r]++`
12. `L[r][q] ← all messages received from q since first receiving token(r,version)`
13. if `tokens_received = #incoming channels, local snapshot for (r,version) is finished`
```

Upon receipt from q of TOKEN for snapshot (r,version)
execute_snapshot()

*Wait for a snapshot request or a token*

**Snapshot request:**
1. $my\_version++$, current_snap[self]=my\_version
2. $S[self] \leftarrow my\_state$
3. *for each outgoing channel q, send(q, TOEKEN; self, my\_version)*
4. tokens_received[self]=0

**TOKEN(q; r,version):**
5. *If current_snap[r] < version*
6. $S[r] \leftarrow my\_state$
7. current_snap[r]=version
8. $L[r][q] \leftarrow empty$, send token(r, version) on each outgoing channel
9. tokens_received[r] $\leftarrow 1$
10. else
11. tokens_received[r]++
12. $L[r][q] \leftarrow all\ messages\ received\ from\ q\ since\ first\ receiving\ token(r,version)$
13. *if tokens_received = #incoming channels, local snapshot for (r,version) is finished*
execute_snapshot()

*Wait for a snapshot request or a token*

**Snapshot request:**
1. $my\_version++,$ current\_snap[self]=my\_version
2. $S[self] \triangleq my\_state$
3. for each outgoing channel $q,$ send($q,$ TOEKEN; self, $my\_version$
4. $tokens\_received[self]=0$

**TOKEN(q; r,version):**
5. If current\_snap[r] < version
6. $S[r] \triangleq my\_state$
7. current\_snap[r]=version
8. $L[r][q] \triangleq empty,$ send token($r,$ version) on each outgoing channel
9. $tokens\_received[r] \triangleq 1$
10. else
11. tokens\_received[r]++
12. $L[r][q] \triangleq all$ messages received from $q$ since first receiving token($r,$version)
13. if tokens\_received = #incoming channels, local snapshot for ($r,$version) is finished

Record local state for r'th snapshot
Update version number of r'th snapshot

Record local state for r'th snapshot
Update version number of r'th snapshot
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
1. my_version++, current_snap[self]=my_version
2. S[self] ← my_state
3. for each outgoing channel q, send(q, TOKEN; self, my_version)
4. tokens_received[self]=0

TOKEN(q; r,version):
5. If current_snap[r] < version
6. S[r] ← my state
7. current_snap[r]=version
8. L[r][q] ← empty, send token(r, version) on each outgoing channel
9. tokens_received[r] ← 1
10. else
11. tokens_received[r]++
12. L[r][q] ← all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for (r,version) is finished

Set of messages on channel from q is empty
Send token on all outgoing channels
Initialize number of received tokens to 1
Snapshot algorithm – pseudo-code

```plaintext
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
1. my_version++, current_snap[self]=my_version
2. S[self] <-- my_state
3. for each outgoing channel q, send(q, TOEKEN; self, my_version)
4. tokens_received[self]=0

TOKEN(q; r,version):
5. If current_snap[r] < version
6. S[r] <-- my state
7. current_snap[r]=version
8. L[r][q] <-- empty, send token(r, version) on each outgoing channel
9. tokens_received[r] <-- 1
10. else
11. tokens_received[r]++
12. L[r][q] <-- all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

Operating Systems, 2015, Meni Adler, Danny Hendler & Roie Zivan
Snapshot algorithm – pseudo-code

execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
1. my_version++, current_snap[self]=my_version
2. S[self] ← my_state
3. for each outgoing channel q, send(q, TOEKEN; self, my_version)
4. tokens_received[self]=0

TOKEN(q; r,version):
5. If current_snap[r] < version
6. S[r] ← my state
7. current_snap[r]=version
8. L[r][q] ← empty, send token(r, version) on each outgoing channel
9. tokens_received[r] ← 1
10. else
11. tokens_received[r]++
12. L[r][q] ← all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for (r,version) is finished

Yet another token for snapshot (r,version)
Snapshot algorithm – pseudo-code

execute_snapshot()

*Wait for a snapshot request or a token*

Snapshot request:
1. `my_version++`, `current_snap[self]=my_version`
2. `S[self] ← my_state`
3. `for each outgoing channel q, send(q, TOEKEN; self, my_version)`
4. `tokens_received[self]=0`

TOKEN(q; r,version):
5. If `current_snap[r] < version`
6. `S[r] ← my state`
7. `current_snap[r]=version`
8. `L[r][q] ← empty, send token(r, version) on each outgoing channel`
9. `tokens_received[r] ← 1`
10. else
11. `tokens_received[r]++`
12. `L[r][q] ← all messages received from q since first receiving token(r,version)`
13. If `tokens_received = #incoming channels, local snapshot for (r,version) is finished`

These messages are the state of the channel from q for snapshot (r,version)
Snapshot algorithm – pseudo-code

execute_snapshot()

*Wait for a snapshot request or a token*

**Snapshot request:**
1. `my_version++`, `current_snap[self]=my_version`
2. `S[self] ← my_state`
3. `for each outgoing channel q, send(q, TOEKEN; self, my_version)`
4. `tokens_received[self]=0`

**TOKEN(q; r, version):**

5. If `current_snap[r] < version`
6. `S[r] ← my state`
7. `current_snap[r]=version`
8. `L[r][q] ← empty, send token(r, version) on each outgoing channel`
9. `tokens_received[r] ← 1`
10. else
11. `tokens_received[r]++`
12. `L[r][q] ← all messages received from q since first receiving token(r,version)`

13. **If all tokens of snapshot (r,version) arrived, snapshot computation is over**

14. **If all tokens of snapshot (r,version) arrived, local snapshot for (r,version) is finished**
Sample execution of the snapshot algorithm

- A token passing system
- Each processor receives, in its turn, a privilege token, uses it to perform privileged operations and then passes it on
- Assume processor p did not receive the token for a long duration so it invokes the snapshot algorithm to find out why

```
  p ————> q
    ^
    v
```
Sample execution of the snapshot algorithm

1. p requests a snapshot

   ![Diagram](p requests a snapshot)

   State(p)={}

2. q sends privilege token

   ![Diagram](q sends privilege token)

   State(p)={}

3. Snapshot token arrives at q

   ![Diagram](Snapshot token arrives at q)

   State(p)={}
   State(q)={}
   \( L_{p,q}={} \)

4. Snapshot token arrives at p

   ![Diagram](Snapshot token arrives at p)

   State(p)={}
   State(q)={}
   \( L_{p,q}={} \)
   \( L_{q,p}={} \)

Observed state:

![Diagram](Observed state)