Synchronization

Distributed Software Systems

Clock Synchronization

When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
Skew between computer clocks in a distributed system

Network

Clock Synchronization Algorithms

The relation between clock time and UTC when clocks tick at different rates.
Clock Synchronization

- Physical clocks drift, therefore need for clock synchronization algorithms
  - Many algorithms depend upon clock synchronization
  - Clock synch. Algorithms - Christian, NTP, Berkeley algorithm, etc.
- However, since we cannot perfectly synchronize clocks across computers, we cannot use physical time to order events

Clock synchronization using a time server

Diagram showing synchronization process between a process (p) and a time server (S).
**Cristian's Algorithm**

Getting the current time from a time server.

**Clock synchronization algorithms**

- **Cristian’s algorithm**
  - p should set its time to \( t + \frac{T_{\text{round}}}{2} \)
  - Earliest time at which S could have placed its time in \( m_t \) was \( \text{min} \) after \( p \) dispatched \( m_t \)
  - Latest point at which it could do so was \( \text{min} \) before \( m_t \) arrived at \( p \)
  - Time by S’s clock when message arrives at \( p \) is in range \([t + \text{min}, t + T_{\text{round}} - \text{min}]\)
    - Accuracy \( \pm (T_{\text{round}}/2 - \text{min}) \)
The Berkeley Algorithm

a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

An example synchronization subnet in an NTP implementation

Note: Arrows denote synchronization control, numbers denote strata.
Logical time & clocks

- Lamport proposed using logical clocks based upon the “happened before” relation
  - If two events occur at the same process, then they occurred in the order observed
  - Whenever a message is sent between processes, the event of sending occurred before the event of receiving
  - X happened before Y denoted by $X \rightarrow Y$

Events occurring at three processes

[Diagram showing physical time and logical events at three processes: $p_1$, $p_2$, and $p_3$.]
Lamport’s algorithm

- Each process has its own logical clock
- LC1: \( C_p \) is incremented before each event at process \( p \)
- LC2:
  1. When process \( p \) sends a message it piggybacks on it the value \( C_p \)
  2. On receiving a message \((m,t)\) a process \( q \) computes \( C_q = \max(C_q, t) \) and then applies LC1 before timestamping the receive event

Lamport timestamps for the events

A diagram showing the timestamps for events a, b, c, d, e, f, and messages \( m_1 \), \( m_2 \). The timeline is divided into physical time and Lamport time.
Vector Timestamps

- Shortcoming of Lamport’s clocks: if $L(e) < L(f)$, we cannot conclude that $e \rightarrow f$
- Vector clocks
  - A process keeps an array of clocks, one for each process
  - Like Lamport timestamps, processes piggyback vector timestamps on messages they send each other
    
    $V = W$ iff $V[j] = W[j]$ for $j = 1, 2, ..., N$
    
    $V \leq W$ iff $V[j] \leq W[j]$ for $j = 1, 2, ..., N$
    
    $V < W$ if $V \leq W$ and $V \neq W$

Vector timestamps for the events

![Diagram showing vector timestamps for events a, b, c, d, e, f, with vectors (1,0,0), (2,0,0), (2,1,0), (2,2,0), (0,0,1), and (2,2,2) and physical time on the x-axis.](image-url)
**Totally ordered logical clocks**

- Logical clocks only impose partial ordering.
- For total order, use \((T_a, P_a)\) where \(P_a\) is processor id.
- \((T_a, P_a) < (T_b, P_b)\) if and only if either \(T_a < T_b\) or \(T_a = T_b\) and \(P_a < P_b\).
- This ordering has no physical significance, but it is sometime useful, e.g. to break a tie between two processes trying to enter a critical section.

**Example: Totally-Ordered Multicasting**

Updating a replicated database and leaving it in an inconsistent state.

![Diagram showing multicasting with updates 1 and 2](image-url)
Distributed mutual exclusion

- Central server algorithm
- Ricart and Agrawal algorithm
  - A distributed algorithm that uses logical clocks
- Ring-based algorithms

NOTE: the above algorithms are not fault-tolerant and not very practical. However, they illustrate issues in the design of distributed algorithms

- Several other mutual exclusion algorithms have been proposed
  - Quorum consensus algorithms - Maekawa’s algorithm
  - We will discuss majority voting in the context of replicated data management

Server managing a mutual exclusion token for a set of processes
Ricart and Agrawala’s algorithm

*On initialization*

\[
\text{state} := \text{RELEASED};
\]

*To enter the section*

\[
\begin{align*}
\text{state} & := \text{WANTED}; \\
\text{Multicast request to all processes;} \\
T & := \text{request’s timestamp;} \\
\text{Wait until (number of replies received = } (N - 1)); \\
\text{state} & := \text{HELDED};
\end{align*}
\]

*On receipt of a request \(<T_i, p_i> \) at \(p_j \) \((i \neq j)\)*

\[
\begin{align*}
\text{if} \ (\text{state} & = \text{HELDED} \text{ or } (\text{state} = \text{WANTED} \text{ and } (T, p_j) < (T_i, p_i))) \\
\text{then} \\
\text{queue request from } p_i \text{ without replying;} \\
\text{else} \\
\text{reply immediately to } p_i;
\end{align*}
\]

*To exit the critical section*

\[
\begin{align*}
\text{state} & := \text{RELEASED}; \\
\text{reply to any queued requests;}
\end{align*}
\]
**Multicast synchronization**

A comparison of three mutual exclusion algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>2 ( (n - 1) )</td>
<td>2 ( (n - 1) )</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to ( \infty )</td>
<td>0 to ( n - 1 )</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>
Maekawa’s algorithm

- Every node needs permission from other nodes in its quorum before it enters critical section
- Quorums are constructed in such a way that no two nodes can be in their critical section at the same time
- The size of each node’s quorum is $O(\sqrt{N})$, which can be shown to be optimal

Construction of quorum sets

Consider a system with 9 nodes
The quorum for any node includes the nodes in its row and column

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Quorum for Node 1 = {1, 2, 3, 4, 7}
Quorum for Node 2 = {2, 5, 8, 1, 3}
There is a non-null intersection for the quorums of any two nodes
Maekawa’s algorithm

On initialization
state := RELEASED;
voted := FALSE;

For pi to enter the critical section
state := WANTED;
Multicast request to all processes in \( V_i - \{p_i\} \);
Wait until (number of replies received = \((K - 1)\));
state := HELD;

On receipt of a request from pi at pj (\( i \neq j \))
if (state = HELD or voted = TRUE)
   queue request from pi without replying;
else
   send reply to pi;
   voted := TRUE;
end if

Maekawa’s algorithm – cont’d

For pi to exit the critical section
state := RELEASED;
Multicast release to all processes in \( V_i - \{p_i\} \);

On receipt of a release from pi at pj (\( i \neq j \))
if (queue of requests is non-empty)
   then
      remove head of queue – from \( p_k \), say;
      send reply to \( p_k \);
      voted := TRUE;
   else
      voted := FALSE;
end if
Election Algorithms

- An election is a procedure carried out to choose a process from a group, for example to take over the role of a process that has failed.
- Main requirement: elected process should be unique even if several processes start an election simultaneously.
- Algorithms:
  - Bully algorithm: assumes all processes know the identities and addresses of all the other processes.
  - Ring-based election: processes need to know only addresses of their immediate neighbors.

A ring-based election in progress

Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown darkened.
The bully algorithm

The election of coordinator \( p_2 \), after the failure of \( p_4 \) and then \( p_3 \)

Detecting global properties

a. Garbage collection

b. Deadlock

c. Termination
Global states and consistent cuts

- Capturing a global state would be straightforward if we had perfectly synchronized clocks.
- How to capture a meaningful global state from local states recorded at different real times?
- Each process records events that correspond to internal actions, e.g. updating a variable, and the sending or receipt of a message.
- A cut is a subset of the system’s global history that is a union of prefixes of process histories.
- A cut is consistent if for each event it contains, it also contains all events that happened before that event.

Consistent and Inconsistent Cuts

- Diagram showing events and their times on a timeline with consistent and inconsistent cuts.
**Chandy and Lamport’s snapshot algorithm**

- **Goal:** record a set of process and channel states for a set of processes such that even if the combination of recorded states may never have occurred at the same time, the recorded state is consistent.
- **State recorded locally at processes**
- **Assumptions**
  - neither channels nor processes fail; communication is reliable
  - channels are unidirectional and provide FIFO message delivery
  - the graph of processes and channels is strongly connected
  - any process may initiate a global snapshot at any time
  - processes may continue with their execution and send and receive normal messages while the snapshot takes place

**Chandy and Lamport algorithm**

- Organization of a process and channels for a distributed snapshot
**Chandy and Lamport’s ‘snapshot’ algorithm**

*Marker receiving rule for process \( p_i \)*

On \( p_i \)'s receipt of a *marker* message over channel \( c \):

- if \( p_i \) has not yet recorded its state:
  - records its process state now;
  - records the state of \( c \) as the empty set;
  - turns on recording of messages arriving over other incoming channels;
- else
  - \( p_i \) records the state of \( c \) as the set of messages it has received over \( c \) since it saved its state.

*Marker sending rule for process \( p_i \)*

After \( p_i \) has recorded its state, for each outgoing channel \( c \):

- \( p_i \) sends one marker message over \( c \) (before it sends any other message over \( c \)).

**Chandy & Lamport algorithm**

- (b) Process \( Q \) receives a marker for the first time and records its local state
- (c) \( Q \) records all incoming message
- (d) \( Q \) receives a marker for its incoming channel and finishes recording the state of the incoming channel
Two processes and their initial states

Assume that p₂ has already received an order for five widgets, which it will shortly dispatch to p₁.

Example: Chandy & Lamport's algorithm

1. Global state $S₀$
   
   $<$1000, 0$>$ $p₁$ $c₂$ (empty) $p₂$ $<$50, 2000$>$
   
   $c₁$ (empty)

2. Global state $S₁$
   
   $<$900, 0$>$ $p₁$ $c₂$ (Order 10, $100), M$ $p₂$ $<$50, 2000$>$
   
   $c₁$ (empty)

3. Global state $S₂$
   
   $<$900, 0$>$ $p₁$ $c₂$ (Order 10, $100), M$ (five widgets) $p₂$ $<$50, 1995$>$
   
   $c₁$ (empty)

4. Global state $S₃$
   
   $<$900, 5$>$ $p₁$ $c₂$ (Order 10, $100) M$ $p₂$ $<$50, 1995$>$
   
   $c₁$ (empty)

(M = marker message)