Time and Coordination in Distributed Systems

Operating Systems

Clock Synchronization

- Physical clocks drift, therefore need for clock synchronization algorithms
  - Many algorithms depend upon clock synchronization
  - Often we need to know the order in which two events occurred on two different computers
  - 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
Skew between computer clocks in a distributed system

Network

Clock synchronization using a time server

\[ p \rightarrow m_t \rightarrow m_r \rightarrow \text{Time server, S} \]
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 $m_l$ was $\min$ after p dispatched $m_r$
  - Latest point at which it could do so was $\min$ before $m_l$ arrived at p
  - Time by S’s clock when message arrives at p is in range $[t + \min, t + T_{\text{round}} - \min]$

- Accuracy $\pm (\frac{T_{\text{round}}}{2} - \min)$

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 events occurring at three processes with physical time and logical clocks.](image-url)
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 \((m_t, t)\) a process \( q \) computes \( C_q = \max(C_q, t) \) and then applies LC1 before timestamping the receive event

Lamport timestamps for the events

```
p1 1 2  m1 4
 a b m1

p2 3
 c
d
m2

p3 1 5
 e
 f
```

Physical time
Vector timestamps for the events

![Diagram showing vector timestamps]

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 \((T_a = T_b \text{ and } P_a < P_b)\)
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

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

On initialization
state := RELEASED;

To enter the section
state := WANTED;
Multicast request to all processes;
T := request’s timestamp;
Wait until (number of replies received = (N – 1));
state := HELD;

On receipt of a request <Ti, pi> at pj (i ≠ j)
if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
then
  queue request from pi without replying;
else
  reply immediately to pi;
end if

To exit the critical section
state := RELEASED;
reply to any queued requests;
**Multicast synchronization**

Maekawa’s algorithm

- Every node needs permission from the other nodes in its quorum before it enters the 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 proved to be optimal.
Construction of coteries

Consider a system with 9 nodes

<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>

The quorum for any node includes the other nodes in the same row and column

Node 1’s quorum = \{1,2,3,4,7\}
Node 2’s quorum = \{1,2,3,5,8\}
Node 6’s quorum = \{4,5,6,3,9\}

The quorum of any two nodes have a non-null intersection. This ensures that two nodes cannot get permission to enter their critical section at the same time

Maekawa’s algorithm

On initialization

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

For \(p_i\) to enter the critical section

\[\text{state} := \text{WANTED};\]
Multicast request to all processes in \(V_i - \{p_i\}\);
Wait until (number of replies received = \((K - 1)\));
\[\text{state} := \text{HELD};\]

On receipt of a request from \(p_j\) at \(p_i\) (\(i \neq j\))
if (\(\text{state} = \text{HELD}\) or \(\text{voted} = \text{TRUE}\))
then
queue request from \(p_i\) without replying;
else
send reply to \(p_i\);
\[\text{voted} := \text{TRUE};\]
end if
Maekawa’s algorithm – cont’d

For $p_i$ to exit the critical section
state := RELEASED;
Multicast release to all processes in $V_i \setminus \{p_i\}$;

On receipt of a release from $p_i$ at $p_j$ ($i \neq j$)
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 chose 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
P₂,
after the failure of P₄ and then
P₃