Synchronization

- Problems in synchronization in distributed systems.
- Synchronization vs. mutual exclusion
- Centralized synchronization mechanisms in distributed systems
- Distributed synchronization mechanisms

*Reading: Coulouris, Chapter 10*

---

Synchronization: Introduction

- A scary scenario:

- **Synchronization**: temporal ordering of sets of events produced by concurrent processes in time.
  - Synchronization between senders and receivers of messages.
  - Control of joint activity.
  - Serialization of concurrent access to shared objects/resources.
- Why not Semaphores ?!
  - centralized systems: shared memory, central clock
  - distributed system: message passing, no global clock
- Events cannot be totally ordered!
A Partial Event Ordering for Distributed Systems  
(Lamport 1978)

- Absence of central time means: no notion of happened-when (no total ordering of events)
- But can generate a happened-before notion (partial ordering of events)
- Happened-Before relation:

1. \( P_i \)  
   \[ a \quad \text{happened-before} \quad b \]  
   Event \( a \) happened-before Event \( b \). (\( a \rightarrow b \))

2. \( P_i \)  
   \[ a \quad \text{happened-before} \quad b \]  
   \[ P_j \]  
   Event \( a \) happened-before Event \( b \). (\( a \rightarrow b \))

3. \( P_i \)  
   \[ a \quad \text{happened-before} \quad b \]  
   \[ P_j \]  
   \[ b \quad \text{happened-before} \quad c \]  
   Event \( a \) happened-before Event \( c \). (\( a \rightarrow c \)) (transitivity)

**happened-before** Relation

- What when no happened-before relation exists between two events?

- Problem:
  - only approximate knowledge of state of other processes
- Need global time:
  - common clock
  - synchronized clocks
Synchronization Schemes

<table>
<thead>
<tr>
<th></th>
<th>centralized</th>
<th>distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutual</td>
<td>central process</td>
<td>circulating token</td>
</tr>
<tr>
<td>exclusion</td>
<td>physical clock</td>
<td>physical clocks</td>
</tr>
<tr>
<td>no mutual exclusion</td>
<td>eventcount</td>
<td>logical clocks</td>
</tr>
</tbody>
</table>

Centralized Synchronization Mechanisms

1. Physical Clocks
   provide a single clock

2. Central Process

   1. Send request message to coordinator to enter C.S.
   2. If C.S. is free, the coordinator sends a reply message. Otherwise it queues request and delays sending reply message until C.S. becomes free.
   3. When leaving C.S., send a release message to inform coordinator.

   - Characteristics:
     - ensures mutual exclusion
     - service is fair
     - small number of messages required
     - fully dependent on coordinator
Centralized Synch. Mechanisms:
3. Eventcounts

• Primitives:

  advance(E)
  • increase value of E by one. Indicates that particular event has happened.
  • Invoked by signaler.

  read(E)
  • return “current” value of E.
  • returns lower bound; why?

  await(E, v)
  • suspend calling process until value of E is at least v.

Eventcounts vs. Semaphores

• Example: Producer-Consumer Problem:

  Producer:  * FULL;  Eventcount  * EMPTY;
  int i = 0;
  while (TRUE) {
    i++;
    produce item;
    await(EMPTY, i-N);
    deposit item;
    advance(FULL);
  }

  Consumer:
  int i = 0;
  while (TRUE) {
    i++;
    await(FULL, i);
    remove item;
    advance(EMPTY)
    consume item;
  }
Eventcounts: Implementation

- **read**:
  1. send read message with seq#.
  2. reply current value with seq#.
  3. return from read call with value

- **advance**
  1. send advance message to owner

- **await**
  1. observer sends await(v) message to owner
  2. when value reaches v, owner sends “await confirm” message to observer.
  3. observer returns from await.

Distributed Synchronization: Physical Clocks

- Conditions for a physical clock $C_i$:
  - runs at approximately correct rate:
    \[
    \frac{dC_i}{dt} - 1 < k
    \]
  - should tell approximately the correct time:
    \[
    \forall i,j \left| C_i(t) - C_j(t) \right| < \varepsilon
    \]

- Synchronizing clocks by exchanging messages:
  - message delay $\delta_m = t - t'$
  - minimum delay $\mu_m > 0$
  - unpredictable delay $\rho_m = \delta_m - \mu_m$
Clock Synchronization: Basic Algorithm

1.) While no synchronization message arrives, clock $C_i$ increases monotonically.
2.) $P_i$ sends synchronization message $m$ at time $t$ with timestamp $T_m = C_i(t)$.
3.) $P_j$ receives synchronization message $m$ at time $t'$. Updates $C_j$ to be $C_i(t') := \max(C_i(t'), T_m + \mu_n)$.

Distributed Synchronization: 2. Logical Clocks

- **Absolute time?**
- **Is chronological ordering necessary?**
- **Logical clock**: assigns a number to each local event.

**Clock Condition**

\[ \forall \text{ Events } a, b : \text{ if } a \rightarrow b, \text{ then } C(a) < C(b) \]

- **In Other Words:**

```
  P_i  a   b  
        C_i(a) < C_i(b)

  P_j  b   c
        C_j(b) < C_j(c)
```
Total Ordering with Logical Clocks

• Rules:
  - Rule 1: increment $C_i$ after every local event.
  - Rule 2: timestamp outgoing messages with current local clock
  - Rule 3: Upon receiving message with timestamp $TS$, $P_j$ updates local clock $C_j$ to be $C_j = \max(C_j, TS+1)$

• Total ordering of events: assuming that clocks satisfy Clock Condition, define following relation:

\[
C_i(a) < C_j(b)
\]

\[
a \Rightarrow b \iff \begin{cases} C_i(a) = C_j(b) \text{ and } i < j \\ \text{or} \\ C_i(a) < C_j(b) \end{cases}
\]

for events $a$ on $P_i$ and $b$ on $P_j$.

Example: Distributed Checkpointing

• “At 5pm everybody writes its state to stable storage!”
• Centralized System:

• Distributed System:
Distributed Checkpointing and Logical Clocks

“At logical-clock time 5000 write state to stable storage!”

Logical Clocks and Distributed Mutual Exclusion

- Mutual Exclusion:
  - Process holding resource must release it before another process can acquire it.
  - Grant requests for resources in order in which they were made.
  - Requests are eventually granted, as long as holding processes return resources.
Lamport’s Algorithm

\[ \text{req}(T_m: \text{Pi}) \rightarrow \text{ack}(T_k) \rightarrow \text{req}(T_m: \text{Pi}) \rightarrow \text{ack}(T_k) \rightarrow \text{req}(T_m: \text{Pi}) \rightarrow \text{ack}(T_k) \rightarrow \ldots \]

Ricart and Agrawala

Reminder: Central Coordinator

Other participants form Distributed Coordinator

\[ \text{request, reply, release cycle} \]
Maekawa (1985)

- Ricart and Agrawala
  - fully symmetric algorithm: all processes run exactly the same algorithm.
  - improvements by fiddling with messages.
- Alternative
  - relax symmetry
  - allow arbitration requests to be exchanged by sets of nodes with pairwise non-null intersections.
- Choice of subsets (Coteries)
  - all pairwise intersections are non-null
  - every node $i$ contained in its own subset $S(i)$
  - all $S(i)$s should have the same size
  - every node $i$ should be contained in same number of subsets

Maekawa (cont)

- Example: Finite Projective Planes
  - $S(0) = \{0,5,6\}$
  - $S(1) = \{1,3,6\}$
  - $S(2) = \{2,1,0\}$
  - $S(3) = \{3,0,4\}$
  - $S(4) = \{4,1,5\}$
  - $S(5) = \{5,3,2\}$
  - $S(6) = \{6,4,2\}$

- Use request, reply, release cycle
  - need $3\sqrt{N}$ messages
Distributed Mutual Exclusion: Conclusion

<table>
<thead>
<tr>
<th>Centralized Scheme</th>
<th>Coteries (Maekawa)</th>
<th>Fully Distributed Scheme (Ricart and Agrawala)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_i$</td>
<td>$P_i$</td>
<td>$P_i$</td>
</tr>
<tr>
<td>3 messages</td>
<td>$3\sqrt{N}$ messages</td>
<td>$3N$ messages</td>
</tr>
</tbody>
</table>

CPSC-662 Distributed Computing  Synchronization