Synchronization and Consistency

Content taken from the following:

"Distributed Systems: Principles and Paradigms" by Andrew S. Tanenbaum and Maarten Van Steen (Chapters 6, 7 and 11)
Various online sources
Warm-up activity

- Break up into groups by table
- Suppose every person in your group is in a different country
  - The only communication mechanism is via email
- Devise a distributed approach to have everyone call a given phone number at the exact same instant in time.
Synchronization

• In a shared-memory system:
  – Core mechanism: mutual exclusion
  – Conditions, semaphores, and barriers

• In a distributed-memory system:
  – Core mechanism: message passing
  – Coordinated clocks
    • Absolute vs. logical
  – Election and consensus algorithms
  – Consistency models and protocols
Clocks / Timers

• Measuring time
  – Movements of sun, moon, and stars
  – Unwinding of wound spring
  – Quartz crystal oscillating under tension
  – Energy transitions of a caesium 133 atom

• Synchronizing absolute clocks
  – Calendars and leap year/second adjustments
    • Coordinated Universal Time (UTC)
  – Clock skew
    • Network Time Protocol (NTP)
Network Time Protocol

- Reference clocks (hardware-based)
- Strataums 1-15 and 16 (unsynced)
- 64-bit time values (<1 ns resolution)

Time offset:

$$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$$

Round-trip delay:

$$\delta = (t_3 - t_0) - (t_2 - t_1)$$

Logical clocks

- Lamport clocks / timestamps
  - Invented by Leslie Lamport in 1978
  - Core notion: "happens-before"
    - Imposes a partial ordering on messages
    - Increment local clock before sending
    - Include local clock when sending
  - Adjust local clock after communications
    - Must preserve “happens-before” ordering
    - Always forwards—never backwards!
  - If a happened before b, then $C(a) < C(b)$
    - Converse is not necessarily true!
    - Does not capture any notion of causality

For more info:
http://dl.acm.org/citation.cfm?id=359563
Vector clocks restore a notion of causality
- Keep a vector of clock values instead of only one
- \( VC_i[j] \) is the logical clock at process \( P_i \)
- \( VC_i[j] = k \) means that \( P_i \) knows that \( k \) events have occurred at \( P_j \) (i.e., \( P_i \)'s knowledge of \( P_j \)'s local time), any of which could have causality influence
Distributed mutual exclusion

- Clocks provide time-based synchronization
- What about task-based synchronization?
- How can we implement mutual exclusion in a distributed system?
Distributed mutual exclusion

- **Token-based** (often used in ring networks)
- **Permission-based**
  - **Centralized** (single coordinator)
  - **Decentralized** (multiple coordinators, need majority vote)
Distributed mutual exclusion

- **Token-based** (often used in ring networks)
  - Simple; slow; susceptible to lost tokens
- **Permission-based**
  - **Centralized** (single coordinator)
    - Easy to implement; single bottleneck and point of failure
  - **Decentralized** (multiple coordinators, need majority vote)
    - More resilient; can be slow; possibility of starvation
Election algorithms

- If a coordinator is needed, there are various election strategies available to choose one
  - **Bully algorithm**
    - Always defer to higher-numbered nodes
  - **Ring algorithm**
    - Enforce one-way election traffic (no token necessary)
  - **Wireless algorithms**
    - Choose the *best* coordinator (e.g., CPU speed, battery life, etc.)
Distributed consensus

- Elections (and related auctions) are a specialized form of the general problem of determining consensus in a distributed system
- **Paxos** protocol: two-phase rounds
  - **Prepare / promise**: A proposer creates a proposal with value N larger than any value it has previously used and sends it to a quorum of acceptors, who respond with a promise to ignore future proposals with a value less than N
  - **Accept / accepted**: If a proposer receives enough promises, it sets a final value M for its proposal and sends it to a quorum of acceptors, who accept it if M is greater than any other proposals it has promised to
  - Real protocol has multiple ways to handle failures and lack of consensus

<table>
<thead>
<tr>
<th>Client</th>
<th>Proposer</th>
<th>Acceptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X--------&gt;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>X---------</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>&lt;---------X--X--X</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>X---------</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>&lt;---------X--X--X</td>
</tr>
<tr>
<td>7</td>
<td>&lt;---------X</td>
<td></td>
</tr>
</tbody>
</table>
Distributed consensus

1. **Client**
   - Send [Request]

2. **Proposer**
   - Send [Prepare] (n)

3. **Acceptors**
   - [Prepare] (n/2+1)

4. **Client**
   - Send [Request]

5. **Proposer**
   - [Promise] (n)

6. **WinningProposer**
   - Send [Accept] (n)

7. **Acceptors**
   - [Accept] (n/2+1)

8. **WinningProposer**
   - Send [Accepted] (n/2+1)

   - Send [Response]
Replication

- All of these protocols require a lot of communication
  - Communication is expensive!
- Alternative: keep redundant data
  - *Replica*: a copy of data
    - In a distributed system, every process could be a separate replica
  - Goal: improved availability or performance
    - Related concepts: *mirroring* and *caching*
    - Relieve single-node access bottlenecks
- Issue: keeping replicas consistent
  - Propagating updates
  - Events (reads/writes) will arrive at different times
  - But maybe we’re ok with some inconsistency
Replication and consistency

- Theme: **loosen consistency constraints** to *decrease communication overhead*
  - Tradeoff: performance vs. consistency

Traditional databases:
  - **ACID** - Atomic, Consistent, Isolated, Durable

Distributed systems:
  - **BASE** - Basically Available, Soft-state, Eventually consistent
Replication

- **Consistency model**: contract between entities and data stores
  - If the entities follow the rules, the data store will be consistent

- **Data-centric models (global view)**
  - **Strict / continuous consistency** (absolute time)
  - **Sequential consistency** (logical time)
  - **Causal consistency** (logical causality)

- **Client-centric models (local view)**
  - Monotonic reads
  - Monotonic writes
  - Read-your-writes
  - Writes-follow-reads
Strict / continuous consistency

- All events are seen instantaneously by all nodes
  - Issue: speed of light (~$3 \times 10^8$ m/s) prevents instantaneous updates, especially in large-scale distributed systems
  - To be practical, designate an interval of allowable deviation

(interval)

$\text{speed of light} \cdot \text{distance traveled}$

1ms 1s 1min 24hr

(not to scale)
Sequential consistency

• Every node sees events in the same order
  – Events must have a total order (i.e., they must be linearizable)
  – Related to Lamport clocks (proposed by the same person)
  – Notation: "W(x)a" means "write value a to item x"
    • (corresponding notation for reads)

P0: W(x)a
P1: W(x)b
P2: R(x)b R(x)a
P3: R(x)b R(x)a

P0: W(x)a
P1: W(x)b
P2: R(x)b R(x)a
P3: R(x)a R(x)b
Sequential consistency

• Every node sees events in the same order
  – Events must have a total order (i.e., they must be linearizable)
  – Related to Lamport clocks (proposed by the same person)
  – Notation: "W(x)a" means "write value a to item x"
    • (corresponding notation for reads)

<table>
<thead>
<tr>
<th>P0:</th>
<th>W(x)a</th>
<th>P0:</th>
<th>W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1:</td>
<td>W(x)b</td>
<td>P1:</td>
<td>W(x)b</td>
</tr>
<tr>
<td>P2:</td>
<td>R(x)b</td>
<td>R(x)a</td>
<td>P2:</td>
</tr>
<tr>
<td>P3:</td>
<td>R(x)b</td>
<td>R(x)a</td>
<td>P3:</td>
</tr>
</tbody>
</table>

Sequentially-consistent

NOT sequentially-consistent
Causal consistency

- **Causally-related** events must be seen in order
  - Reads are causally-related to corresponding writes
  - Writes are causally-related to previous operations on the same node
  - Related to the notion of causality from vector clocks
  - To verify, build global causality chain and check each process’s view

| P0: W(x)a | P1: W(x)b | P2: R(x)b R(x)a | P3: R(x)a R(x)b |
| P0: W(x)a | P1: R(x)a W(x)b | P2: R(x)b R(x)a | P3: R(x)a R(x)b |
Causal consistency

- **Causally-related** events must be seen in order
  - Reads are causally-related to corresponding writes
  - Writes are causally-related to previous operations on the same node
  - Related to the notion of causality from vector clocks
  - To verify, build global causality chain and check each process’s view

<table>
<thead>
<tr>
<th>P0:</th>
<th>W(x)a</th>
<th>P0:</th>
<th>W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1:</td>
<td>W(x)b</td>
<td>P1:</td>
<td>R(x)a  W(x)b</td>
</tr>
<tr>
<td>P2:</td>
<td>R(x)b  R(x)a</td>
<td>P2:</td>
<td>R(x)b  R(x)a</td>
</tr>
<tr>
<td>P3:</td>
<td>R(x)a  R(x)b</td>
<td>P3:</td>
<td>R(x)a  R(x)b</td>
</tr>
</tbody>
</table>

Causally-consistent  NOT causally-consistent
**Causal consistency**

- **Causally-related** events must be seen in order
  - Reads are causally-related to corresponding writes
  - Writes are causally-related to previous operations on the same node
  - Related to the notion of causality from vector clocks
  - To verify, build global causality chain and check each process’s view

![Causally-consistent](image1)

![NOT causally-consistent](image2)
Sequential / causal consistency

- Is this sequence sequentially-consistent?
- Is it causally-consistent?

<table>
<thead>
<tr>
<th></th>
<th>P0: W(x)a</th>
<th></th>
<th>W(x)c</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1:</td>
<td>R(x)a</td>
<td>W(x)b</td>
<td></td>
</tr>
<tr>
<td>P2:</td>
<td>R(x)a</td>
<td>R(x)c R(x)b</td>
<td></td>
</tr>
<tr>
<td>P3:</td>
<td>R(x)a</td>
<td>R(x)b R(x)c</td>
<td></td>
</tr>
</tbody>
</table>
Sequential / causal consistency

- Is this sequence sequentially-consistent?
- Is it causally-consistent?

P0: \( W(x)a \)                      \( W(x)c \)
P1: \( R(x)a \) \( W(x)b \)
P2: \( R(x)a \) \( R(x)c \) \( R(x)b \)
P3: \( R(x)a \) \( R(x)b \) \( R(x)c \)
Sequential / causal consistency

- Is this sequence sequentially-consistent?
- Is it causally-consistent?

P0: $W(x)a$ $W(x)c$
P1: $R(x)a$ $W(x)b$
P2: $R(x)a$ $R(x)c$ $R(x)b$
P3: $R(x)a$ $R(x)b$ $R(x)c$

Causally-consistent, but
NOT sequentially-consistent

![Diagram showing the sequence and causal relationships]
Partial vs. total ordering

- **Ordering**: definition of “<” operator
  - Usually over pairs of entities (for us, messages)
  - **Total ordering**: definition of “<” for all pairs
    - Depicted graphically using a line
  - **Partial ordering**: definition of “<” for some pairs
    - Depicted graphically using a graph or lattice

“<” ≡ “happens-before” (partial ordering)

“<” ≡ “subset-of” (partial ordering)

“<” ≡ “less-than” (total ordering)

from: https://en.wikipedia.org/wiki/Lattice_(order)
Implication

• Sequential consistency implies causal consistency
  – There is no way for the partial ordering of causal consistency to contradict the total ordering implied by sequential consistency
    • Both properties (writes before reads on same data & strict ordering for events on single processes) used to build the partial ordering are already enforced by any valid total ordering
  – Thus, every sequentially-consistent sequence must also be causally-consistent
  – Colloquially: causal consistency is \textit{looser} than sequential consistency
Client-centric consistency

- Previous models focused on a **global** view of data
  - Sometimes called **data-centric** consistency models
- In a distributed system, we may only be interested in the **local** view at any given node
  - This motivates **client-centric** consistency models
Client-centric consistency

- Original application: **Bayou** database system for mobile computing
  - Developed in mid-1990s
  - Massive number of replicas
  - Multiple networks and unreliable connectivity
  - Data-centric, global consistency models are infeasible
  - Theme: loosen the constraints!
  - **Four** different consistency models (not mutually exclusive)

For more info:
http://dl.acm.org/citation.cfm?id=504497
Monotonic reads / writes

- **Monotonic reads**: if a process reads X, any successive read to X will see the same value or a more recent one
  - i.e., the process will never see an older version
  - e.g., distributed email database (messages shouldn’t disappear when migrating between replicas)

- **Monotonic writes**: if a process writes X, any successive write to X will see the effect of the first write
  - i.e., newer writes must wait for older ones to finish
  - e.g., wiki edits
• **Read-your-writes**: if a process writes X, any successive read to X will see the effect of the write
  
  – i.e., reads will never see old versions
  – Closely related to monotonic reads
  – Systems that often lack this consistency:
    • Retrieving websites
    • Updating passwords

• **Writes-follow-reads**: if a process reads X, any successive write to X will see the same value or a more recent one
  
  – i.e., writes will never see old versions
  – e.g., posts to an email list
Replicas

- Server-initiated (e.g., mirroring)
  - Updates are pushed to other replicas
- Client-initiated (e.g., caching)
  - Updates are pulled from other replicas
  - Write-through vs. write-back
- Peer-to-peer
  - Nodes have symmetric roles
  - Requires well-defined protocol for enforcing consistency
Consistency protocols

- Continuous consistency protocols
  - Bounding numerical deviation (# of updates)
  - Bounding staleness deviation (time of updates)

- Primary-based protocols
  - Primary: one replica that coordinates all writes for a data item
  - Remote-write: forward all writes to primary (similar to write-through)
  - Local-write: periodic updates sent to primary (similar to write-back)

- Replicated-write protocols
  - Active replication: multicast updates to all replicas
    - Need a reliable and efficient multicast protocol
  - Quorum-based voting: replicas vote on updates to replicas
    - Need a distributed voting/consensus protocol
Distributed version control