Consistency and Replication

- Introduction
- Data-centric consistency
- Client-centric consistency
- Distribution protocols
- Consistency protocols

Replication

- Goal:
  - Reliability
  - Performance
- Problem:
  - Consistency

- Examples: distributed object
Object Replication (1)

Organization of a distributed remote object shared by two different clients.

Object Replication (2)

(a) A remote object capable of handling concurrent invocations on its own.
(b) A remote object for which an object adapter is required to handle concurrent invocations
Object Replication (3)

a) A distributed system for replication-aware distributed objects.
b) A distributed system responsible for replica management

Performance and Scalability

- **Main issue:**
  - To keep replicas consistent, we generally need to ensure that all *conflicting* operations are done in the same order everywhere

- **Conflicting operations:**
  - From the world of transactions:
    - Read–write conflict: a read operation and a write operation act concurrently
    - Write–write conflicts: two concurrent write operations
  - Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability

- **Solution:**
  - weaken consistency requirements so that hopefully global synchronization can be avoided
Data-Centric Consistency Models

general organization of a logical data store, physically distributed and replicated across multiple processes

---

Strict Consistency

Any read to a shared data item X returns the value stored by the most recent write operation on X

Very hard to implement in distributed system
Sequential Consistency

The result of any execution is the same as if the operations of all processes were executed in some sequential order, the operations of each individual process appear in this sequence in the order specified by its program.

Linearizability and Sequential Consistency

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>z = 1;</td>
</tr>
<tr>
<td>print (y, z);</td>
<td>print (x, z);</td>
<td>print (x, y);</td>
</tr>
</tbody>
</table>

Linearizable:
Sequential, plus operations are ordered according to a global time
Linearizability and Sequential Consistency

Four valid execution sequences for the processes

Causal Consistency (1)

Necessary condition:
- Writes that are potentially causally related must be seen by all processes in the same order
- Concurrent writes may be seen in a different order on different machines
Causal Consistency (2)

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
<th>W(x)c</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2:</td>
<td>R(x)a</td>
</tr>
<tr>
<td>P3:</td>
<td>R(x)a</td>
</tr>
<tr>
<td>P4:</td>
<td>R(x)a</td>
</tr>
</tbody>
</table>

This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Causal Consistency (3)

(a) P1: W(x)a
P2: R(x)a W(x)b
P3: R(x)b R(x)a
P4: R(x)a R(x)b

(b) P1: W(x)a
P2: W(x)b
P3: R(x)b R(x)a
P4: R(x)a R(x)b
**FIFO Consistency (1)**

**Necessary Condition:**
- Writes done by a single process are seen by all other processes in the order in which they were issued.
- But writes from different processes may be seen in a different order by different processes.

---

**FIFO Consistency (2)**

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2:</td>
</tr>
<tr>
<td>R(x)a</td>
</tr>
<tr>
<td>W(x)b</td>
</tr>
<tr>
<td>W(x)c</td>
</tr>
<tr>
<td>P3:</td>
</tr>
<tr>
<td>R(x)b</td>
</tr>
<tr>
<td>R(x)a</td>
</tr>
<tr>
<td>R(x)c</td>
</tr>
<tr>
<td>P4:</td>
</tr>
<tr>
<td>R(x)a</td>
</tr>
<tr>
<td>R(x)b</td>
</tr>
<tr>
<td>R(x)c</td>
</tr>
</tbody>
</table>

A valid sequence of events of FIFO consistency.
Weak Consistency (1)

Properties:

• Accesses to synchronization variables associated with a data store are sequentially consistent
• No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
• No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed

Weak Consistency (2)
Release Consistency (1)

Synchronization variable access:

1. **Acquire**: requester to wait until the shared data can be accessed

2. **Release**: sends requester’s local value to other servers in data store

Release Consistency (2)

Rules:

- Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
- Before a release is allowed to be performed, all previous reads and writes by the process must have completed.
- Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).
Entry Consistency

With entry consistency, each shared data item is associated with a synchronization variable. When acquiring the synchronization variable, the most recent values of its associated shared data item are fetched.

Summary of Consistency Models

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order</td>
</tr>
<tr>
<td>FIFO</td>
<td>All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order</td>
</tr>
</tbody>
</table>

Models with synchronization operations

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Shared data can be counted on to be consistent only after a synchronization is done</td>
</tr>
<tr>
<td>Release</td>
<td>Shared data are made consistent when a critical region is exited</td>
</tr>
<tr>
<td>Entry</td>
<td>Shared data pertaining to a critical region are made consistent when a critical region is entered.</td>
</tr>
</tbody>
</table>
Client-centric consistency models

- System model
- Monotonic reads
- Monotonic writes
- Read-your-writes
- Write-follows-reads

Goal: avoid system-wide consistency:
concentrating on what specific clients want, instead of what should be maintained by servers

Examples:
- DNS: Updates are propagated slowly, and inserts may not be immediately visible
- NEWS: Articles and reactions are pushed and pulled throughout the Internet, such that reactions can be seen before postings
- Lotus Notes: Geographically dispersed servers replicate documents, but make no attempt to keep (concurrent) updates mutually consistent
- WWW: Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page
Consistency for Mobile Users

At location $A$ you access the database doing reads and updates.

At location $B$ you continue your work, but unless you access the same server as the one at location $A$, you may detect inconsistencies:
- your updates at $A$ may not have yet been propagated to $B$
- you may be reading newer entries than the ones available at $A$
- your updates at $B$ may eventually conflict with those at $A$

Need: Eventual Consistency
**Monotonic Reads**

If a process reads the value of a data item $x$, any successive read operation on $x$ by that process will always return that same or a more recent value.

**Monotonic Writes**

A write operation by a process on a data item $x$ is completed before any successive write operation on $x$ by the same process.
Read Your Writes

The effect of a write operation by a process on data item \( x \), will always be seen by a successive read operation on \( x \) by the same process.

Writes Follow Reads

A write operation by a process on a data item \( x \) following a previous read operation on \( x \) by the same process, is guaranteed to take place on the same or a more recent value of \( x \) that was read.
Distribution Protocols

• Replica Placement
• Update Propagation
• Epidemic Protocols

Replica Placement

The logical organization of different kinds of copies of a data store into three concentric rings.
Server-Initiated Replicas

Counting access requests from different clients

Update Propagation

- Propagate only notification/invalidation of update (often used for caches)
- Transfer data from one copy to another (distributed databases)
- Propagate the update operation to other copies (also called active replication)
Pull versus Push Protocols

- Pushing updates: server-initiated approach, in which update is propagated regardless whether target asked for it.
- Pulling updates: client-initiated approach, in which client requests to be updated.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Push-based</th>
<th>Pull-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>State of server</td>
<td>List of client replicas and caches</td>
<td>None</td>
</tr>
<tr>
<td>Messages sent</td>
<td>Update (and possibly fetch update later)</td>
<td>Poll and update</td>
</tr>
<tr>
<td>Response time at client</td>
<td>Immediate (or fetch-update time)</td>
<td>Fetch-update time</td>
</tr>
</tbody>
</table>

Lease

A contract in which the server promises to push updates to the client until the lease expires:

- **Age-based leases**: An object that hasn’t changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases**: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases**: The more loaded a server is, the shorter the expiration times become
Epidemic Algorithms (1/2)

Basic Idea
Assume there are no write–write conflicts:
• Update operations are initially performed at one or only a few replicas
• A replica passes its updated state to a limited number of neighbors
• Update propagation is lazy, i.e., not immediate
• Eventually, each update should reach every replica

Epidemic Algorithms (2/2)

Anti-entropy:
Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards

Gossiping:
A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).
Consistency Protocols

- Primary-based protocols
  - Local or remote
- Replicated-write protocols
- Cache-coherence protocols

Remote-Write Protocols (1)

Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.
Remote-Write Protocols (2)

The principle of primary-backup protocol.

Local-Write Protocols (1)

Primary-based local-write protocol in which a single copy is migrated between processes.
Local-Write Protocols (2)

Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

Active Replication (1)

The problem of replicated invocations.
Active Replication (2)

(a) Forwarding an invocation request from a replicated object.
(b) Returning a reply to a replicated object.

Quorum-Based Protocols

Three examples of the voting algorithm:

a) A correct choice of read and write set
b) A choice that may lead to write-write conflicts
c) A correct choice, known as ROWA (read one, write all)