Replication and Consistency

Today
- Replication
- Consistency models
- Consistency protocols

How to recognize the moods of an Irish setter
The value of replication

- For reliability and availability
  - Avoid problems with disconnection, data corruption, node crashes …

- For performance
  - Multiple workers on copies of the same data, data closer to process (geography)

- So what’s not to like?
  - Copies must be kept consistent – all changes have to be applied to all
  - How and in what order determines the prices of replication
  - Choice could make replication too costly in terms of performance!
Data-centric consistency

- Consistency is traditionally discussed in the context read/write operations on a shared data store

- Consistency model: a contract between processes and a (distributed) data store
  - Data store specifies precisely what the results of r/w operations are in the presence of concurrency

Distributed shared database, shared memory, file system
Classical consistency models

- Without a global clock, hard to tell what operation is first – alternative, a range of consistency models
  - Not a clear best solution – exactly what can be tolerated depends on the application

- Much past work from concurrent programming
  - Multiple processes accessing share resources

- Some basic notation
  - $W_i(x)a$ – process $P_i$ wrote value $a$ to $x$
  - $R_i(x)b$ – process $P_i$ read value $b$ from $x$

\[
\begin{array}{c}
\hline
P1: W(x)a \\
\hline
\end{array}
\]
\[
\begin{array}{ccc}
P2: & R(x)\text{NIL} & R(x)a \\
\hline
\end{array}
\]

\[\text{Time to propagate}\]
• Result of any execution is as if operations of all processes were executed in some order, and the operations of each process appear in this sequence in the order specified by its program

• … the interleaving is consistent with the real time at which operations occurred in the actual execution

But we can’t always synchronize clocks to the degree we want …
Sequential consistency

- Result of any execution is as if operations of all processes were executed in some sequential order, and the operations of each process appear in this sequence in the order specified by its program
  - i.e., Any valid interleaving of operations is OK, but all processes see the same interleaving

Absolute time does not matter
Linearizability and sequential consistency

- Every linearizable service is sequentially consistent
- The converse does not hold

Linearizable and sequentially consistent

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

Sequentially consistent but not linearizable

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

- BTW, both concepts conceived by Lamport
Causal consistency

- **Writes that are potentially causally related must be seen by all processes in the same order; concurrent writes may be seen in different order by different processes**

Weaker than sequential, note that $W_1(x)a$ and $W_2(x)b$ are concurrent.

To keep track of which process has seen which write, vector timestamps.
Grouping operations

- Sequential and causal consistency are defined at the level of reads and writes
  - Initially developed for shared memory multiprocessors

- But, in apps concurrent access is typically controlled through synchronization mechanisms and transactions
  - Enter/leave critical section, grouping operations
  - Bracketing turns a series of operations into an atomically executed unit (coarser granularity)
  - Don’t care that a series of operations are immediately known to others, but the effect of the series itself to be known
Grouping operations

- Synchronization variables to define consistency
  - Synchronization variables associated with data
  - Each synchronization variable has an owner – the process that last acquire it
  - Before entering a CS, acquire synch variable from owner; release it when leaving

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
# Grouping operations – Entry consistency

- **Entry consistency with synch variables**
  - When a process does an acquire, request can’t complete until all guarded shared data has been brought up to date.
  - Before updating a shared data item, a process must enter a critical section (CS) in exclusive mode.
  - Before entering a CS in nonexclusive mode, a process must first check with owner of a synch var guarding the section to fetch the most recent copies of the guarded shared data.

<table>
<thead>
<tr>
<th></th>
<th>Acq(Lx)</th>
<th>W(x)a</th>
<th>Acq(Ly)</th>
<th>W(y)b</th>
<th>Rel(Lx)</th>
<th>Rel(Ly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>Acq(Lx)</td>
<td>R(x)a</td>
<td></td>
<td></td>
<td>R(y)NIL</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>Acq(Ly)</td>
<td>R(y)b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P2 will get a for x, but may get nil when reading y.

Since P3 first does an acquire for y, it will read b when y is released by P1.
Consistency models and protocols

- Consistency protocol – implementation of a specific consistency model
- Passive/primary-based
  - One replica in charge, keeping all backups in sync
  - Remote-write (primary-backup) and local-write protocols
- Replicated-write
  - Active – replicas are state machines playing equivalent roles
  - Quorum-based – not all, just enough replicas
Primary-based remote-write protocols

- Primary-backup protocol – all writes are blocking, forwarded to primary server; reads are local

Note

- Process that does the write may block for a long while; but this is fault tolerant and easy to implement
- A straightforward implementation of sequential consistency
- A non-blocking variation would trades fault tolerance for performance
Primary-based local-write protocols

- Primary-backup protocol with local writes – migrate primary copy between processes that want to write
- Multiple successive writes can be done locally

Can be applied to mobile computing, for operation while being disconnected
  - Mobile node becomes primary of what it needs to update while disconnected
Replicated-write protocols – active

- Write operations can be done at multiple replicas
- Same updates, everywhere in the same order
- Need totally ordered multicast
  - For large-scale distributed systems – use a sequencer
  - Very close to primary-based
Replicated-write protocols - quorum

- For scalability – a majority (quorum) rather than all; distinguish read and write quorum
  - $N_r$ – read quorum; $N_w$ – write quorum
  - $N_r + N_w > N$ (to prevent read/write conflicts)
  - $N_w > N/2$ (to prevent write-write conflicts)

- An interesting case – Read-on, Write-All ROWA
  - Cheap read (any replica) but $$$ writes
Eventual consistency – a missing model

- We can allow replicas to differ from each other, to different degrees – weak consistency
- Eventual – If no update takes place for a while, all replicas will eventually become consistent
  - Relies on total propagation and consistent ordering
- If concurrent access is rare, most operations are read, and processes can handle inconsistency
  - DNS
    - Name space partitioned into domains with a naming authority
    - Only the authority can update, everyone can read/cache
    - Write-write conflict never occur, only read-write
  - WWW
    - Only webmaster can update pages
    - Everyone can read and caches improve scalability
Pros/cons of eventual consistency

- High-availability, good scalability, simplicity of design, but ...

- Limited use to applications where conflicts are rare, underlying data semantics simple, users can cope with tentative information – calendar?

- Cheaper to implement, harder to program for
  - For programmers, supply dependency checks and merge procedures – not trivial

- Replication becomes no-transparent to applications
  - For users, data they read is tentative
  - Changes made are tentative too (they may be altered later)
Eventual consistency and mobility

- Mobility and replication transparency
- As you change locations
  - Updates may not have yet been propagated
  - New updates may eventually conflict with old ones
- All you want is that the entries you updated and/or read at A, are in B the way you left them in A
  - In that case, the database will seem consistent to you
Client-centric consistency models

- Models discussed so far assume concurrent processes update a shared data store – data-centered
  - Mobility brings a new perspective – same user accessing data from different replicas

- Client-centric consistency – *Consistency for a single client*, nothing about concurrent access by different clients

- Originated from the work on Xerox PARC’s Bayou, a database to support collaborative application in mobile computing environment [Terry et al. 1994, 1998]
  - Bayou’s consistency model for mobile, disconnected use – eventually consistent data stores
Client-centric consistency

- Bayou’s four guarantees
  - Monotonic reads
  - Monotonic writes
  - Reads your writes
  - Writes follow reads

- Guarantees – either the storage system ensures them for each r/w belonging to a session or it informs the application that it can’t

- Some useful notation
  - \( x_i[t] \) – version of data item \( x \) at local copy \( L_i \) at time \( t \)
  - \( WS(x_i[t]) \) – Set of write operations at \( L_i \) on \( x \)
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.

Examples:
- Reading your personal calendar updates from different servers
- Reading (not modifying) incoming mail in the move
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
  - i.e. a write on \(x\) is performed only if that copy has been brought up to date

- The ordering of multiple previous writes may or not be important

<table>
<thead>
<tr>
<th>L1:</th>
<th>W((x_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>WS((x_1))</td>
</tr>
<tr>
<td></td>
<td>W((x_2))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1:</th>
<th>W((x_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2:</td>
<td>W((x_2))</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

- Maintaining versions of replicated files in the correct order everywhere (CVS-like)
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
  - i.e. a write is always completed before a successive read by the same process, no matter where the read is

Example:
- Changing your password in dylan and try to login into zappa too soon after

L1: \( W(x_1) \)

L2: \( WS(x_1;x_2) \) \( R(x_2) \)

L1: \( W(x_1) \)

L2: \( WS(x_2) \) \( R(x_2) \)

Writes on copy L1 of x have not been propagated
Writes follows 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.

**Example:**
- See reactions to posted articles only if you have seen the original posting (a read “pulls in” the corresponding write operation)

```
L1: WS(x₁)                   R(x₁)
    L2: WS(x₁; x₂)           W(x₂)
```

```
L1: WS(x₁)                   R(x₁)
    L2: WS(x₂)               W(x₂)
```

Writes on copy L1 of $x$ have not been propagated.
Bayou’s guarantees

- **Writes**
  - have a unique ID (WID), maybe granted by server that first accepts it
  - are propagated among servers by a process called anti-entropy (aka, rumor mongering, lazy replication)
  - All servers apply non-commutative writes to their DB in the same order
    - A variety of techniques to agree upon order (e.g., in Grapevine, originating timestamp)
  - To handle the allow conflict, different strategies
    - Write order determine what write “wins”
    - Humans resolve the conflict

- **Read from any server, guarantees provided by session manager**
Bayou’s guarantees

- Session manager maintains, per session
  - Read-set - set of WIDs for the writes that are relevant to the session reads
  - Write-set – set of WIDs for the writes performed in the session

- To provide read-your-write
  - Whenever a write is accepted by a server, it assigns a WID
  - Adds the WID to the session’s write-set
  - Before each Read to server S at time t, the session manager must check that the write-set is a subset of WS(S,t)
    - Check can be done at the server (by passing the write-set) or on the client by retrieving the servers’ list of WIDs in WS(S,t)
  - Keep trying with other servers if that’s not the case!
Summary

- Replication for performance and reliability
- Replication, however, introduces a few issues
  - The problem of consistency, which we may pay in terms of performance
- The “details” of placement and management – where to place replica servers? How to keep up up-to-date?