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 price 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
  - Specifies what the results of read/write operations are in the presence of concurrency
  - Must be intuitively clear or won’t be used (by app developers)
Classical consistency models

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

- 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 $x$

\[
\begin{align*}
\text{P1: } & W(x)a \\
\text{P2: } & R(x)\text{NIL} & R(x)a
\end{align*}
\]

Time to propagate
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
Sequentially consistent and
- ... 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 ...
Linearizability and sequential consistency

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

<table>
<thead>
<tr>
<th>Linearizable and sequentially consistent</th>
<th>Sequentially consistent but not linearizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: ( W(x)a )</td>
<td>P1: ( W(x)a )</td>
</tr>
<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>

- 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.**

To keep track of which process has seen which write, vector timestamps are used.

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

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

- But, in apps concurrent access is typically controlled through synchronization mechanisms and transactions
  - Enter/leave critical section for 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: \[\text{Acq}(L_x) \, W(x) \, a \, \text{Acq}(L_y) \, W(y) \, b \, \text{Rel}(L_x) \, \text{Rel}(L_y)\]
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.

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y)NIL
P3: Acq(Ly) R(y)b

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

- Remote write
  - No replicas

- Passive/primary-based
  - One replica in charge, keeping all others (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
Remote-write protocols

- Single server
  - All writes are executed at single server
  - No replication

![Diagram of remote-write protocols](image)

1. Write request
2. Forward write
3. Write ack
4. Ack write
Primary-backup protocols

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

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

- 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
  - All or a majority of 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

- In some cases, we can allow replicas to differ from each other, to different degrees – weak consistency

- 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
Eventual consistency – a missing model

- Practical system generally want eventual consistency

- Eventual – If no update takes place for a while, all replicas will eventually become consistent
  - Relies on total propagation
    - Every write is eventually received by each server
    - Writes propagated among servers by a process called anti-entropy (aka, rumor mongering, lazy replication)
  - Consistent ordering
    - All servers apply non-commutative writes to their databases in the same order
    - A variety of techniques to agree upon order
Pros/cons of eventual consistency

- What you get: 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

- 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:
  - Concurrent processes update a shared 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 work on Xerox PARC’s Bayou, a database to support collaborative applications in mobile computing environment
  - Bayou’s consistency model for mobile, disconnected use – eventually consistent data stores
Client-centric consistency

- Bayou’s four guarantees/consistency models
  - 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

The set of write operations at L2 include those done at L1

Order of access by process

Writes on copy L1 of $x$ have not been propagated
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 (e.g., if writes are commutative)

```
L1:  W(x_1)  
L2:   \hspace{1cm} WS(x_1) \hspace{1cm} W(x_2) 
```

- 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

\[
\begin{align*}
\text{L1:} & \quad W(x_1) \\
\text{L2:} & \quad \text{WS}(x_1;x_2) \quad \text{R}(x_2)
\end{align*}
\]
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)
Bayou’s guarantees

- Writes have a unique ID (WID)
  - Maybe granted by server that first accepts it
- Writes are
  - propagated among servers
  - all servers apply non-commutative writes to their DB in the same order
- Conflicting writes, different strategies to handle that
  - 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 writes that are relevant to the session reads
  - Write-set – Set of WIDs for 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
Bayou’s guarantees

- Better than read/write sets, use vector clocks
  - Each server \( i \) keeps a write vector, \( WVC_i \) – entry \( j \) is the timestamp of the most recent write originated from \( S_j \) that has been processed by \( i \)
  - A vector clock for each session, \( SVC_A \) – entry \( i \) the maximum timestamp of writes in \( A \) originated at server \( i \)

- When client logs in sever \( S_i \) as part of session \( A \)
  - Pass \( SVC_A \), if \( SVC_A[j] > WVC_i[j] \), \( S_i \) has not seen all writes originated from \( S_j \) than the client has seen
Summary

- Replication for performance and reliability
- But nothing is for free
  - Consistency, which we may pay in terms of performance
- A different view and different approaches focusing on data or clients
- Many “details” of placement and management – where to place replica servers? How to keep up up-to-date?