Consistency and Replication

Today
- Reasons for replication
- Consistency models
- Replica management
- Consistency protocols
Reliability, performance and scalability

- For reliability
  - Replicas can avoid issue 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
  - The choice may make replication, meant to improve scalability and performance, too costly in terms of performance!
Data-centric consistency

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

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

- No “best” solution, but application dependent
- Degrees of consistency
  - Replicas may differ in their numerical value (relative or absolute difference – your account balance)
  - Replicas may differ in their relative staleness
  - There may differences with respect to (number and order) of performed update operations

- Conit: consistency unit ⇒ specifies the data unit over which consistency is to be measured
  - A single stock in the stock exchange
  - An individual weather report

- Too fine-grained conits, more to manage; too coarse-grain conits, false sharing …
Continuous consistency

- Good to capture consistency requirement
- Some issues
  - There’s an overhead from managing many conits
  - You still need protocols to enforce consistency
  - Programmers must specify the requirements of their applications; complex models tend to be ignored

```java
DependsOnConit(ConitQ, 4, 0, 60);
read msg m from queue Q;
```

*Read depends on ConitQ; staleness should be limited to 4 unseen updates, 0 tentative local updates, last check should have been at most 60 seconds ago*
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

Notation

\( W_i(x)a \) – process \( P_i \) wrote value \( a \) to \( x \)
\( R_i(x)b \) – process \( P_i \) read value \( b \) from \( x \)

Absolute time does not matter
Causal consistency

- *W*rites 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.

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

Weaker than sequential, note that $W_1(x)c$ and $W_2(x)b$ are concurrent.
Grouping operations – entry consistency

- Sequential and causal consistency are defined at the level of reads and writes; initially developed for shared memory multiprocessors.
- At the application level, concurrent access is typically controlled through synchronization mechanisms:
  - Enter/leave critical section.
- Don’t care that reads and writes of a series of operations are immediately known to other processes; just want the effect of the series itself to be known:
  - 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.
### Grouping operations – entry consistency

- **Entry consistency**
  - When a process does an acquire, the acquire 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 in exclusive mode.
  - Before entering a critical section 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.

- **Weak consistency implies that we need to lock/unlock data (implicitly or not)**

<table>
<thead>
<tr>
<th>P1:</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>P2:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3:</td>
<td></td>
<td></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.
Client-centric consistency models

- For some distributed data stores with rare simultaneous updates, *eventual consistency* is enough
  - DNS, WWW, distributed email
- Problems appear if the same user access data from different replicas
- Consider a distributed database to which you have access through a notebook (that acts as a front end)
  - At location A you access the database doing reads/uploads.
  - At B you continue working, but …
Client-centric consistency models

– … but unless you access the same server as when at 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

• 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 – consistency for a single client, nothing about concurrent access by different clients
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

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(x₁) → L2: WS(x₁; x₂) → R(x₂)

L1: W(x₁) → L2: WS(x₂) → R(x₂)

 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)
Replica placement

- If support replication
  - Where to place replica servers and content?
- What are the best $K$ out of $N$ possible locations for a replica servers
  - Select one server at a time so to minimize the average distance between clients and replicas. Computationally expensive.
  - Position nodes in a $d$-dimensional geometric space, where distance reflects latency. Identify the $K$ regions with highest density and place a server in every one. To compute the region size use the average distance between two nodes and $K$. Computationally cheaper.
Content replication

- Different types of replicas
  - Permanent replicas: Process/machine always having a replica, think of it as the initial set
  - Server-initiated replica: Process that can dynamically host a replica on request of another server in the data store (remember you already have the replica servers placed)
  - Client-initiated replica: Process that can dynamically host a replica on request of a client (client cache)
Server-initiated replicas

- Keep track of access counts per file, aggregated by considering server closest to requesting clients
  - $C_1$ and $C_2$ share the same closest server $P$
- Number of accesses
  - $< \text{threshold } D \Rightarrow$ drop file
  - $> \text{threshold } R \Rightarrow$ replicate file
  - between $D$ and $R$ (and more requests at $P$ than at $Q$) $\Rightarrow$ migrate file to $P$

![Diagram showing server-initiated replicas](image-url)
Content distribution

- Consider only a client-server combination
  - Propagate only notification/invalidation of update
  - Transfer data from one copy to another
  - Propagate the update operation (aka active replication)
  - No single approach is best, but depends on available bandwidth and read-to-write ratio at replicas

- Pushing/pulling updates
  - Push - server-initiated, update is propagated regardless whether target asked for it
  - Pulling - client-initiated, client requests to be updated

- Tradeoffs
  - State at server
  - Messages sent
  - Response time at the client
Content distribution

- **Leases** to dynamically switch between pulling and pushing
  - A contract – server promises to push updates to client until the lease expires

- **Make lease expiration time dependent on system’s behavior (adaptive leases):**
  - Age-based: Long-lasting leases for object that hasn’t changed for a while
  - Renewal-freq based: The more often a client requests a specific object, the longer the expiration time for that client/object
  - State-based: Higher (state-space) load at servers, shorter expiration times

- **Unicasting or multicasting**
  - With push-based, multicasting may be a good idea
  - With pull-based, unicast is your only reasonable model
Consistency protocols – continuous

- Consistency protocol – an implementation of a specific consistency model
- Continuous consistency
  - Bounding numerical deviations
    • Replicas help to keep other replicas within bounds by pushing updates, looking at what they think everybody has seen (each keeps a log of writes with origins, $TW[i,j]$ – writes executed at $S_i$ originated from $S_j$)
  - Similar for staleness keeping track of what has been seen last from another replica
    • Replicas keep a real-time vector clock; starts pulling writes soon as time diff. is exceeding some limit
- Bounding ordering deviations
  • Because replicas tentatively apply updates submitted to them; each has a local queue of tentative writes, keep the length bounded
  • When reaching limit, stop accepting writes and try to commit the tentative writes by agreeing on some globally consistent order
    - In practice, primary-based or quorum-based protocols
Primary-based protocols

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

  ![Diagram of primary-backup protocol]

  - 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 trade fault tolerance for performance

Note
Primary-based 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

- Write operations can be done at multiple replicas
- Ensure each operation is carried out so that a majority vote (quorum) is established; distinguish read and write quorum
- File is replicated on N servers
  - \( N_r \) – 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)

This may not work:
- \( N_R = 3, N_W = 10 \)
- \( N_R = 7, N_W = 6 \)
- \( N_R = 1, N_W = 12 \)

This works:
- ROWA – Read from one, write to all
Summary

- Again, we use 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