Today

- Reasons for replication
- Consistency models
- Consistency protocols

Consistency and Replication
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
  - The choice may make replication 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 processes and a (distributed) data store
  - 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 [Yu and Vahdat 2002]
  - 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
Continuous consistency

- Good to capture consistency requirement
- Some issues
  - There’s overhead from managing many conits – too fine-grained conits, more to manage; too coarse-grain conits, false sharing …
  - We still need protocols to enforce consistency
  - Programmers must specify the requirements of their applications and, in gral, complex models tend to be ignored

```plaintext
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*
Classical consistency models

- 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 $x$

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

- Lineralizable – the interleaving is consistent with the real time at which operations occurred in the actual execution

<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)a R(x)b</td>
</tr>
</tbody>
</table>

Absolute time does not matter
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 applications concurrent access is typically controlled through synchronization mechanisms
  - Enter/leave critical section, grouping operations
  - Don’t care that a series of operations are immediately known to others, but the effect of the series itself to be known

- 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
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>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>Acq(Lx)</td>
<td>R(x)a</td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R(y)b</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.
Eventual consistency

- 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

- Bayou’s consistency model for mobile, disconnected use – eventually consistent data stores

- Eventual consistency – if no update takes place for a while, all replicas will eventually become consistent
  - Replication becomes no-transparent to applications
  - Cheaper to implement, harder to program for
Client-centric consistency models

- But, 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
  - *Consistency for a single client*, nothing about concurrent access by different clients
  - Bayou’s four models: monotonic reads, monotonic writes, reads your writes, writes follow reads
  - 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

- Example
  - Maintaining versions of replicated files in the correct order everywhere (CVS-like)
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($x_2$)

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)
Consistency protocols – continuous

- Consistency protocol – an implementation of a specific consistency model
- Continuous consistency
  - Bounding numerical deviations
    • Replicas keep other replicas within bounds by pushing updates, looking at what they think everybody has seen
  - Similar for staleness
    • Keep track of what have been seen last from another replica with a real-time vector clock; starts pulling writes when time difference > 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 at 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

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 trade fault tolerance for performance
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
- Active replication
  - Same updates, everywhere in the same order
  - Multicast them – totally ordered multicast
- Quorum based
  - 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)

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 – where to place replica servers? How to keep up up-to-date?