Today

- Reasons for replication
- Consistency models
- Replica management
- Consistency protocols
Performance and reliability

- Replications for reliability and performance
  - Reliability – you always have a valid copy
  - Performance – enables scaling in
    • Number – to handle more users/processes
    • Geographic spread – to bring it closer to the user

- Problem – now you need to keep them consistent!
  - Ensure all conflicting operations are done in the same order everywhere
    • Read–write conflict: a read & write operation act concurrently
    • Write–write conflict: two concurrent write operations
  - Remedy worst than the disease – potentially downgrade scalability
  - *Can you weaken consistency requirements to avoid global synchronization?*
Data-centric consistency models

- A data store is a distributed collection of storages accessible to clients
  - Shared memory, database or file system

Consistency model

- Contract between a (distributed) data store and processes
- Data store specifies precisely what the results of read/write operations are in the presence of concurrency
Continuous consistency

- We can actually talk about a degree of consistency
  - Replicas may differ in their numerical value (relative or absolute difference – your account balance)
  - Replicas may differ in their relative staleness (when was it last updated)
  - 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 [Yu & Vahdat 02]
  - A single stock in the stock exchange example
  - The account balance

- Too fine-grained conits, more to manage; too coarse-grain conits, false sharing …
Consistent ordering of operations

- Problem – how to consistently order operations on shared data?

- Operations
  - \( W_i(x)a \) – Process \( P_i \) write value \( a \) to data item \( x \)
  - \( R_i(x)b \) – Process \( P_i \) read value \( b \) from data item \( x \)

- Assume each data item is initially NIL

- Operations occur over a time line

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

- There may be some propagation delay (note the first read of \( x \) by \( P_2 \)
Sequential consistency

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

- A program should work with all valid sequences according to its contract with the data store.

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

Note: nothing to do with time; W(x)a happens before W(x)b in absolute time.

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

Here, however, different processes see the operations in different order.
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**
  - Causally related – they *may* have influenced each other

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

Causally but not sequentially consistent

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

Causally consistent

Not any more - \( R_2(x)a \) introduces causality
Grouping operations – entry consistency

- Instead of single operations, groups of them bracketed by enter_cs/leave_cs (critical section)
- We need clear semantics for enter/leave cs; think in terms of synchronization variables
  - To enter/leaves its CS, process should acquire/release the relevant synch variables
    - Can be shared if all it is needed is to read – non-exclusive
  - Each synch variable has an owner (process holding it)
- Operation conditions
  - Acquires do not success until process has seen all updates to guarded variables
  - Before exclusive mode access is granted, everybody should release the synch variable (even in non-exclusive mode)
  - If a process wants non-exclusive access to a guarded variable it should first get the updates from the owner
Grouping operations – entry consistency

How to associate data with synch variables? Explicitly or implicitly (e.g. the whole object)

<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>Acq(Ly)</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
Client-centric consistency models

- For some distributed stores with mostly reads and rare simultaneous updates, eventual consistency is enough
  - DNS, WWW, distributed email

- Consider a distributed database to which you have access through a notebook (that acts as a front end)
  - Potential problems if the same user access data from different replicas

- Client-centric consistency – consistency for one client, *nothing about concurrent access by different clients*
Client-centric consistency models

- Originated from the work on Bayou
- Bayou – a database system for mobile computing
  - Network connectivity is unreliable
- Four different consistency models – process access a
  the data store through a local copy; updates are
  eventually propagated to other copies
  - Monotonic-read
  - Monotonic-write
  - Read-your-writes
  - Writes-follows-reads
- Some notation
  - $x_i[t]$ – Version of data item $x$ at local copy $L_i$ at time $t$
  - $WS(x_i[t])$ – Series of write operations at $L_i$ that resulted in $x_i[t]$
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
  - i.e. If you’ve seen a value of \(x\) at time \(t\), you’ll never see anything older at a later time

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

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

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

\[\begin{align*}
L1: & \quad W(x_1) \\
L2: & \quad WS(x_2) \quad R(x_2)
\end{align*}\]

W(x_1) is not part of WS(x_2)
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
  - i.e. Any successive write by a process on a data item \( x \) will be performed on a copy of \( x \) that is up to date with the value most recently ready by that process

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

- If the system supports replication, new questions:
  - Who does it?
  - When does it do it?
  - Where does it place them?

- Placing has two parts
  - Placing replica servers? Optimization problem: the best $K$ out of $N$ possible locations for a replica
    - Pick one server at a time to minimize the average distance between clients and replicas
    - Select the $K$-th largest ASes and place a server at the best-connected host
    - Split space into regions based on density (distance between nodes) and put a server in the $K$ regions with highest density
  - Where to place the replicas themselves
Content replication – types of replicas

- Permanent replicas – think of it as the initial set
  - E.g. Replication in a cluster, mirrors
- Server-initiated replica – dynamically on request of another server in the data store
  - Track access counts per file, aggregated by considering server closest to requesting clients
  - Accesses < threshold D → drop file
  - Accesses > threshold R → replicate file
  - Access between D and R (and more at P than at Q) → migrate file to server P
- Client-initiated replica – dynamically per client request (client cache); at the client itself or a shared cache
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 the best, but depends on available bandwidth and read-to-write ratio at replicas

<table>
<thead>
<tr>
<th>Issue</th>
<th>Push-based</th>
<th>Pull-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>State at 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>

- Pushing/pulling updates:
  - Push - server-initiated, update is propagated regardless whether target asked for it
  - Pulling - client-initiated, client requests to be updated
Leases to dynamically switch between pulling and pushing
- A contract – server promises to push updates to client until lease expires

Make lease expiration time dependent on system’s behavior (adaptive leases):
- Age-based – If object hasn’t changed for a while, it will not change in the near future, provide a long-lasting lease
- Renewal-frequency based – The more often a client requests an object, the longer the expiration time for that client/object
- State-based – Higher load at servers, shorter expiration times (keep track of fewer clients)

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

- **Bounding numerical deviations**
  - Replicas help to keep other replicas within bounds by *pushing* updates, looking at what they think everybody has seen
    - Replicas help each other to fill in matrix of “writes executed by \( S_i \) that originated at \( S_j \)”
    - Propagation of updates can be done through gossip

- **Similar for staleness**, having servers keep track of what has been seen last from others (vector clock)
  - Replica starts *pulling* writes soon as time diff. is exceeding some limit

- **Bounding ordering deviations**
  - As servers tentatively apply updates submitted to them; each one has a local queue of tentative writes, keep the length bounded
  - When at limit, stop accepting writes and try to commit tentative writes by agreeing on some globally consistent order
Primary-based protocols

- A simple implementation of sequential consistency
- Primary-backup protocol – all writes are blocking, forwarded to primary server; reads are local

The process that does the write may block for a long while; but this is fault tolerant and easy to implement

A non-blocking approach trades 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
Replicated-write protocols

- Write operations can be done at multiple replicas
- Active replication
  - Each replica has an associated process to carry out updates
  - Propagate the update operation
- Replicas need to agree on order of updates
  - Use totally-ordered multicast using Lamport clocks, or
  - Use a sequencer to decide on ordering
Replicated-write protocols

- Gather quorum before reading/writing
- Ensure each operation is carried out in a way that a majority vote (quorum) is established
- 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 works

With ROWA – you can read from one but must 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.