Why replication?

- **Performance**
  - Distributing the access demand to multiple near-by replicas improves performance

- **Reliability**
  - Multiple copies provide better protection against corrupted or unavailable data replicas
Data-Centric Consistency Models

- Data store is a persistent storage distributed across multiple machines on which replicas of data are stored.

- Consistency model = a contract between clients and the data store:
  - If you follow my rules then I will promise you “correct” data (latest data).
  - Absence of global clock makes the determination of “latest” data difficult.

- The general organization of a logical data store, physically distributed and replicated across multiple processes.
**Continuous Consistency**

- Introduced by Yu and Vadhat (2002)

- Applications specify tolerable inconsistencies
  - Deviation in numerical values (#updates applied)
  - Deviation in staleness (last update time) between replicas
  - Deviation with respect to the number and ordering of updates

- Conit = set of data items on which consistency is applied
  - Granularity of conit can affect the number and size of updates
Continuous Consistency

- Implementing bounds on numerical deviation
  - \( TW[i, j] = \# \text{writes originating at } j \text{ that are performed at } i \)
  - \( TW[k, i, j] = k\)'s best guess of \( TW[i, j] \)
  - Maintain \( TW[i,j] \) and \( TW[k, i, j] \)
  - Define \( v(t) = v(0) + \sum_{k=1..N} TW[k,k] \)
    - \( V(t) = \# \text{updates until time } t \)
  - Define \( v(i,t) = v(0) + \sum_{k=1..N} TW[I,k] \)
    - \( V(i,t) = \# \text{updates seen at } I \text{ until time } t \)
  - Bounding \( TW[k,k] - TW[k, i, k] \) by \( d/N \) ensures that \( v(t) - v(i,t) \leq d \)
Continuous Consistency

Implementing bounds on staleness deviations

- Let $T(i) =$ local real time at node $i$
- Maintain real time vector clocks $RVC(k,i) = T(i)$ means that $k$ thinks the last update at $i$ was at $T(i)$
- If $T(k) - RVC(k,i) >$ threshold then $k$ pulls updates that originated at $I$ after $RVC(k,i)$

Implementing bounding ordering deviations

- Each node maintains a queue of tentative local updates to be ordered and applied
- When a node’s queue exceeds a threshold, then tentative writes need to be committed via a distributed mutex protocol
Consistent ordering of operations

- Augment continuous consistency by providing models to order tentative update operations

- Notation

  - $R_i(x)a$ means the read of $x$ at node $I$ returned $a$
  - $W_i(x)b$ means the write of $x$ with $a$ at node $I$
Sequential consistency

The result of any execution is the same as if the read/write operations by all processes on the data store were executed in some sequential order, and all the operations of each individual process appear in this sequence in the order specified by its program.

i.e. all processes see the same valid interleaving of operations.
### Sequential Consistency

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

\(\text{valid}\)  

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

\(\text{invalid}\)
Sequential Consistency

- Three concurrently executing processes.

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = 1;)</td>
<td>(y = 1;)</td>
<td>(z = 1;)</td>
</tr>
<tr>
<td>print ((y, z));</td>
<td>print ((x, z));</td>
<td>print ((x, y));</td>
</tr>
</tbody>
</table>
Sequential Consistency

Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

- **x = 1;**
  - **print ((y, z);**
  - **y = 1;**
  - **print (x, z);**
  - **z = 1;**
  - **print (x, y);**

  **Prints:** 001011
  **Signature:** 001011 (a)

- **x = 1;**
  - **y = 1;**
  - **print (x,z);**
  - **z = 1;**
  - **print (x, y);**

  **Prints:** 101011
  **Signature:** 101011 (b)

- **y = 1;**
  - **z = 1;**
  - **print (x, y);**
  - **x = 1;**
  - **print (x, z);**

  **Prints:** 010111
  **Signature:** 1101011 (c)

- **y = 1;**
  - **x = 1;**
  - **z = 1;**
  - **print (y, z);**
  - **print (x, y);**

  **Prints:** 111111
  **Signature:** 111111 (d)
Strict Consistency

- Behavior of two processes, operating on the same data item.
- A strictly consistent store.
- A store that is not strictly consistent.
Causal consistency

- Relaxes sequential consistency
- The data store obeys the following rule
  - Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.
    - i.e. only causally related operations need to be sequentially consistent
Casual Consistency

This sequence is allowed with a casually-consistency, but not with sequentially consistency

<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>
Casual Consistency

(a) Invalid

P1: \( W(x)a \)
P2: \( R(x)a \quad W(x)b \)
P3: \( R(x)b \quad R(x)a \)
P4: \( R(x)a \quad R(x)b \)

(b) Valid

P1: \( W(x)a \)
P2: \( W(x)b \)
P3: \( R(x)b \quad R(x)a \)
P4: \( R(x)a \quad R(x)b \)
Synchronization variables for grouping operations

- Access to replicas is done via critical sections
  - Processes execute enter/leave critical section
  - Synchronization variables guard set of replicated data
  - Each synchronization variable has a current owner, which can access the associated guarded data (critical section)

- Synchronization variables act as locks
  - Processes acquire/release locks
**Entry Consistency Rules**

- An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.

- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.

- After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.
Entry Consistency

A valid event sequence for entry consistency.

<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>Acq(Lx)</td>
<td>R(x)a</td>
<td>R(y)NIL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3:</td>
<td></td>
<td>Acq(Ly)</td>
<td>R(y)b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Client-centric consistency models

- Data-centric consistency models aim to provide systemwide consistent view of the data store.
- When there are concurrent updates by many processes data-centric models are used.
- To efficiently guarantee sequential consistency, transactions or locks are needed, which are rather expensive.
- Consider special case where there are no concurrent updates or if they occur they can easily be resolved.
- We introduce special models to hide many inconsistencies from the client in rather inexpensive ways.
- Client-centric consistency models provide consistency guarantees for a single client.
Eventual consistency

- There are no write-write conflicts but only read-write conflicts.
- If no updates happen for a long time then all replicas will eventually become consistent.
- Complications arise when a client accesses different replicas at different times.
Client-centric consistency models provide consistency guarantees for a single client

- These models originate from the Bayou mobile DBMS (1994-98)

Assumptions

- A client reads or writes a single replica (could differ over time) of data item
- Updates are eventually propagated to all replicas

Notation

- $X[i, t]$ is the version of the replica at node $i$ at time $t$
- $WS(x[i,t])$ is the set of writes to the replica at node $i$ until time $t$
- $WS(x[i,t], x[j,t'])$ the writes $WS(x[i,t])$ are reflected in the version $x[j,t']$ at the later time $t'$
Monotonic reads

Any process, upon reading version $x(i,t)$ of a data item, it will never read an older version of that same data item.

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

(b) 
\[
\begin{align*}
\text{L1: } & \quad \text{WS}(x_1) \quad \text{R}(x_1) \\
\text{L2: } & \quad \text{WS}(x_2) \quad \text{R}(x_2) \quad \text{WS}(x_1; x_2)
\end{align*}
\]
invalid
Monotonic writes

A write to a replica of a data item at a node is delayed until all previous writes to the same data element are executed at that node.

(a) 
L1: \( W(x_1) \)
L2: \( W(x_1) \quad W(x_2) \)

valid

(b) 
L1: \( W(x_1) \)
L2: \( \quad \) \( W(x_2) \)

invalid
Read-your-writes

A write to a replica of a data element will be reflected to any replica of the data element that is subsequently read.

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

valid

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

invalid
**Writes-follow-reads**

A write of a replica, that follows a read on version K of the replica, will only be done to a replica with version K or later.

- E.g. postings of msgs to newsgroups

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

(a)  **valid**

\[
\begin{array}{ll}
\text{L1: } & WS(x_1) \quad R(x_1) \\
\text{L2: } & WS(x_2) \quad W(x_2) \\
\end{array}
\]

(b)  **invalid**
Implementing Client-centric consistency

- Naïve method:
  - Each update has a global unique identifier (GID)
  - Each client has for each data element two sets of updates
    - The read-set = updates relevant to the value read by the client
    - The write-set=all updates performed by the client so far
  - In monotonic reads, the client presents to the replica server its read-set to verify it can perform the read or forward the read to another more up-to-date replica server
  - In monotonic-writes, the client presents its write-set to the replica set for verification
  - In read-your-writes, the client presents the replica server its write-set; server performs read if it did all the writes
  - In writes-follow-reads, the replica server performs all writes in the client’s read-set first, and then it performs the requested write
**Implementing Client-centric consistency**

- We can improve on the naïve method by a compact representation of read and write sets that relies on vector clocks.
- Each write at a node is timestamped by \#writes performed.
- Let \( WCV[i,j] = \) timestamp of most recent write that originated at node \( j \) and has been performed by node \( i \).
  - Writes from node \( j \) are performed in FIFO order at node \( i \).
- A read/write-set is compactly represented via vector clocks:
  - Write-set at node \( i \) is \( WCV[i] \).
  - Read-set at client whose most recent read was from node \( i \) is \( WCV[i] \).
  - What is the write-set at the client?
Replica Management

- Decide where to place replica servers
- Decide which replica servers to place replicas of a data element
Placement of Replica servers

- Optimization problem of placing K servers in a network with N nodes to optimize a certain optimization criterion
  - E.g. average distance of each node from closest server
  - Problem is often NP-hard
  - Kalpakis et al (2001) describe a dynamic programming algorithm for optimal placement of servers in a tree network with minimum total read, write, and storage costs
- Plethora of heuristics
  - Qiu et al provide a simple heuristic of placing one server at a time
  - Various clustering heuristics based on embeddings of the nodes in a high dimensional normed vector space
Replica Placement

- The logical organization of different kinds of copies of a data store into three concentric rings.
Server-Initiated Replicas

Rabinovich et al (1999) describe a dynamic replica placement scheme

Counting access requests from different clients.
Client-initiated replicas

- Caching
  - Client caches improve access times
  - Sharing of caches among many clients may be useful
    - Contrary to traditional file system usage
Content distribution

- What should be propagated to the replicas?
  - Notifications of updates only
    - Leading to invalidation protocols
  - Updated value of replica
  - Description of the operation to perform the update
    - Active replication – useful when it has succinct description
    - Hybrid: propagate updates for a while then just update notifications
- Should updates be pushed or pulled?
Pull versus Push Protocols

A comparison between push-based and pull-based protocols in the case of multiple client, single server systems.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Push-based</th>
<th>Pull-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>State of 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>
Lease-based content distribution

Lease = promise by a server to push updates to a client for a certain period
- Provide a dynamic method to switch between push and pull

Three types of leases
- Age-based
  - Grant long-lasting leases to data items that are expected to remain unmodified for a long time
- Refresh-based
  - Grant long lasting leases to data items that are frequently refreshed by the clients
- State/load-based
  - As server becomes overloaded it decreases the duration of new leases
Unicast vs multicast push?

Multicasting generally more efficient if supported by the underlying network stack or hardware (e.g., broadcast medium)
Consistency Protocols

- Consistency protocols implement consistency models
  - Remote-write protocol
    - Primary-backup protocol
  - Local-write protocol
    - Primary copy migrates among writers, and then follow the primary-backup protocol
  - Replicated-write protocols
    - Active replication & the notion of sequencer
      - Need total-ordering of multicast messages (see Vector clocks)
    - Quorum-based protocols
Primary-backup protocol

W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read
Local-Write Protocol

1. Read or write request
2. Forward request to current server for \( x \)
3. Move item \( x \) to client's server
4. Return result of operation on client's server
Local-Write Protocol

**W1.** Write request
**W2.** Move item x to new primary
**W3.** Acknowledge write completed
**W4.** Tell backups to update
**W5.** Acknowledge update

**R1.** Read request
**R2.** Response to read
Active Replication

The problem of replicated invocations.

Client replicates invocation request

All replicas see the same invocation

Replicated object

Object receives the same invocation three times
Active Replication

(a) Forwarding an invocation request to a replicated object.

(b) Returning a reply to a replicated object.

Client replicates invocation request.

Coordinator of object B.

Result.
Quorum-based protocols

- Use read and write quorums
  - Read most recent replica from read quorum
  - Update all replicas in write quorum
- Example: Thomas/Gifford voting protocol

![Diagram showing read and write quorums with nodes A, B, C, D, E, F, G, H, I, J, K, L, with read quorum sizes NR = 3, NW = 10, write quorum sizes NR = 7, NW = 6, and modified write quorum sizes NR = 1, NW = 12.](image-url)
Cache Coherence Protocols

- Cache-coherence protocols ensure that a client’s cache is consistent with the server-initiated replicas

- Coherence detection strategy
  - When are any inconsistencies detected?
    - Verify before the first access
    - Access while verifying
    - Verify before commit
Cache Coherence Protocols

Coherence enforcement strategy

How are caches kept consistent?

- Invalidate/propagate updates from server to cache
- Write-through cache= client updates cache and the forward each update to server
- Write-back cache=similar to write-through, except that client can perform multiple updates before updating the server