Reasons for Replication

- Data is replicated to increase the reliability of a system.
- Replication for performance
  - Scaling in numbers
  - Scaling in geographical area
- Caveat
  - Gain in performance. E.g. Client caching in web browser
  - Cost of increased bandwidth for maintaining replication
Replication for Scaling

- **Placing copies of data close to client processes can help with scaling.** But keeping copies up to date requires more network bandwidth. Updating too often may be a waste. Not updating often enough is the flip side.

- **How to keep the replicas consistent?** Use global ordering using Lamport timestamps or use a coordinator. This may require a lot of communication for a large system.

- **In many cases, the real solution is to loosen consistency constraints.** E.g. The updates do not have to be atomic. To what extent we can loosen depends highly on the access and update patterns as well as the application.

- A range of consistency models are available.
Data-centric Consistency Models

Figure 7-1. The general organization of a logical data store, physically distributed and replicated across multiple processes.
Consistency Model

A **consistency model** is essentially a contract between processes and the data store. *It says that if processes agree to obey certain rules, the store promises to work correctly.*

Models with minor restrictions are easy to use while models with major restrictions are more difficult to use. But then easy models don’t perform as well. So we have to make trade-offs.
Inconsistencies

Three different axes of inconsistencies:

- Deviation in numerical values between replicas
- Deviation in staleness between replicas
- Deviation with respect to the ordering of update operations

A *conit* specifies the unit over which consistency is to be measured.
Continuous Consistency (1)

Figure 7-2. An example of keeping track of consistency deviations. Numerical deviation: (1, 5) A hasn’t seen one operation from B, and max difference is 0 for x and 5 for y, if all operations at B were done. So we use 5 as the weight.
Continuous Consistency (2)

Figure 7-3. Choosing the appropriate granularity for a conit. (a) Two updates lead to update propagation.
Continuous Consistency (3)

Figure 7-3. Choosing the appropriate granularity for a conit.
(b) No update propagation is needed (yet).
Consistency Models

- *Data-centric* consistency models
  - Sequential consistency
  - Causal consistency
  - Grouping operations

- *Client-centric* consistency models
  - Eventual consistency
  - Monotonic reads
  - Monotonic writes
  - Read your writes
  - Writes follow reads
Sequential Consistency (1)

Figure 7-4. Behavior of two processes operating on the same data item. The horizontal axis is time.
A data store is *sequentially consistent* when:

*The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*
Sequential Consistency (3)

(a)

\[
\begin{align*}
P1: & \quad W(x)a \\
P2: & \quad W(x)b \\
P3: & \quad R(x)b \quad R(x)a \\
P4: & \quad R(x)b \quad R(x)a
\end{align*}
\]

(b)

\[
\begin{align*}
P1: & \quad W(x)a \\
P2: & \quad W(x)b \\
P3: & \quad R(x)b \quad R(x)a \\
P4: & \quad R(x)a \quad R(x)b
\end{align*}
\]

Figure 7-5. (a) A sequentially consistent data store. (b) A data store that is not sequentially consistent.
Figure 7-6. Three concurrently-executing processes.
Figure 7-7. Four valid execution sequences for the processes of Fig. 7-6. The vertical axis is time. Signature is output of $P_1$, $P_2$, $P_3$ concatenated.
Causal Consistency (1)

For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

*W*rites *t*hat *a*r *p*otentially *c*ausally *r*elated *m*ust *b*e *s*een *b*y all *p*rocesses *i*n *t*he *s*ame *o*rder. *C*oncurrent *w*rites *m*ay *b*e *s*een *i*n a *d*ifferent *o*rder *o*n *d*ifferent *m*achines.

Weaker than sequential consistency.
### Causal Consistency (2)

<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>

**Figure 7-8.** This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.
Causal Consistency (3)

Figure 7-9. (a) A violation of a causally-consistent store.

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

(b)
Causal Consistency (4)

Figure 7-9. (b) A correct sequence of events in a causally-consistent store.
Causal Consistency (5)

Implementing causal consistency requires keeping track of which processes have seen which writes. It effectively means a dependency graph of which operation is dependent on which other operations must be constructed and maintained. This can be done using vector timestamps.
Entry Consistency with Grouping Operations (1)

Necessary criteria for correct synchronization:

- An acquire access of a synchronization variable is not allowed to perform until all updates to guarded shared data have been performed with respect to that process.
- Before exclusive mode access to synchronization variable by process is allowed to perform with respect to that process, no other process may hold synchronization variable, not even in nonexclusive mode.
- After exclusive mode access to synchronization variable has been performed, any other process’ 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 with Grouping Operations (2)**

<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>

Figure 7-10. A valid event sequence for entry consistency.
Eventual Consistency

Figure 7-11. The principle of a mobile user accessing different replicas of a distributed database.
Monotonic Reads (1)

A data store is said to provide monotonic-read consistency if the following condition holds:

*If a process reads the value of a data item x, then any successive read operation on x by that process will always return that same value or a more recent value.*

But no guarantees on concurrent access by different clients.
Figure 7-12. The read operations performed by a single process \( P \) at two different local copies of the same data store. 
(a) A monotonic-read consistent data store.
Monotonic Reads (3)

Figure 7-12. The read operations performed by a single process P at two different local copies of the same data store. (b) A data store that does not provide monotonic reads.
Monotonic Writes (1)

In a monotonic-write consistent store, the following condition holds:

*A write operation by a process on a data item \( x \) is completed before any successive write operation on \( x \) by the same process.*
Figure 7-13. The write operations performed by a single process P at two different local copies of the same data store. (a) A monotonic-write consistent data store.
Monotonic Writes (3)

Figure 7-13. The write operations performed by a single process $P$ at two different local copies of the same data store. (b) A data store that does not provide monotonic-write consistency.
A data store is said to provide read-your-writes consistency, if the following condition holds:

*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.*
Figure 7-14. (a) A data store that provides read-your-writes consistency. $WS(x_1)$ is the series of write operations that took place since initialization at a local copy. $WS(x_1;x_2)$ denotes that operations in $WS(x_1)$ also been performed at another local copy that has its set of operations in $WS(x_2)$. 
Read Your Writes (3)

Figure 7-14. (b) A data store that does not.
A data store is said to provide writes-follow-reads consistency, if the following holds:

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.
Writes Follow Reads (2)

Figure 7-15. (a) A writes-follow-reads consistent data store.
Figure 7-15. (b) A data store that does not provide writes-follow-reads consistency.
Replica Management

A key issue for any distributed system that supports replication is to decide where, when and by whom replicas should be placed, and subsequently the mechanisms to use for keeping the replicas consistent. Two separate problems:

- placing *replica servers*
- placing *content*
Replica-Server Placement

Figure 7-16. Choosing a proper cell size for server placement.

Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall, Inc. All rights reserved. 0-13-239227-5
Figure 7-17. The logical organization of different kinds of copies of a data store into three concentric rings.
Server-Initiated Replicas

Figure 7-18. Counting access requests from different clients.
Client-Initiated Replicas

- Same as *client caches*
- Used to improve access times
- Sharing a cache between clients may or may not improve the hit rate. E.g. Web data, file servers
- Shared caches can be at departmental or organization level
State versus Operations

Possibilities for what is to be propagated:

1. Propagate only a notification of an update.
2. Transfer data from one copy to another.
3. Propagate the update operation to other copies.
Pull versus Push Protocols

<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>

Figure 7-19. A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems. *Use leases to dynamically switch between push and pull.*
Leases for Pull versus Push

A lease is a promise by the server that it will push updates to the client for a specified time. When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary. Use leases to dynamically switch between push and pull. Types of leases:

- **Aged-based** leases. Grant long lasting leases to data that is expected to remain unmodified
- **Renewal-frequency based** leases. Give longer leases to clients that where its data is popular
- **State-space overhead based** leases. Start lowering expiration times as space runs low
- **Using multicasting** can be much more efficient than unicasting for the updates
Implementations: Primary-based Protocols

- Implementations tend to prefer simpler consistency models
- Primary-based protocols are used for implementing sequential consistency
- In a primary-based protocol, each data item $x$ in the data store has an associated primary, which is responsible for write operations on $x$
- Primary can be fixed at a remote server or write operations can be carried out locally after moving the primary to the process where the write operation was initiated
Remote-Write Protocols

Figure 7-20. The principle of a primary-backup protocol. Implements sequential consistency. Non-blocking version the primary acknowledges after updating its copy and informs backup servers afterwards.
Local-Write Protocols

Figure 7-21. Primary-backup protocol in which the primary migrates to the process wanting to perform an update. Updates have to be propagated back to other replicas.
Replicated-Write Protocols

- Write operations can be carried out at multiple replicas instead of just one. In *active replication*, the operation is forwarded to all replicas and in *quorum-based* protocols it is done by majority voting.

- Active replication requires operations to be carried out in the same order everywhere. So it requires totally ordered multicasts using either Lamport timestamps (e.g.) or a central coordinator.
Quorum-Based Protocols

Figure 7-22. Three examples of the voting algorithm. (a) A correct choice of read and write set. (b) A choice that may lead to write-write conflicts. (c) A correct choice, known as ROWA (read one, write all). $N_R + N_W > N$, $N_W > N/2$