Consistency and replication: Introduction

Reasons for replication

Performance and scalability

Main issue
To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the same order everywhere.

Conflicting operations: From the world of transactions
- **Read–write conflict**: a read operation and a write operation act concurrently
- **Write–write conflict**: two concurrent write operations

Issue
Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability. **Solution**: weaken consistency requirements so that hopefully global synchronization can be avoided.
Data-centric consistency models

Consistency model
A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

Essential
A data store is a distributed collection of storages:

![Diagram of distributed data store and processes with local copies](image-url)
Continuous Consistency

We can actually talk about a degree of consistency:

- replicas may differ in their numerical value
- replicas may differ in their relative staleness
- there may be 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.
**Example: Conit**

Each replica has a vector clock: ([known] time @ A, [known] time @ B)

- B sends A operation \( [\langle 5, B \rangle : g \leftarrow d + 45] \); A has made this operation permanent (cannot be rolled back)
**Example: Conit**

**Replica A**

- **Conit**
  - $d = 558$ // distance
  - $g = 95$ // gas
  - $p = 78$ // price

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;5, B&gt;$, $g \leftarrow g + 45$</td>
<td>$[g = 45]$</td>
</tr>
<tr>
<td>$&lt;8, A&gt;$, $g \leftarrow g + 50$</td>
<td>$[g = 95]$</td>
</tr>
<tr>
<td>$&lt;9, A&gt;$, $p \leftarrow p + 78$</td>
<td>$[p = 78]$</td>
</tr>
<tr>
<td>$&lt;10, A&gt;$, $d \leftarrow d + 558$</td>
<td>$[d = 558]$</td>
</tr>
</tbody>
</table>

- **Vector clock A** = $(11, 5)$
- **Order deviation** = 3
- **Numerical deviation** = $(2, 482)$

**Replica B**

- **Conit**
  - $d = 412$ // distance
  - $g = 45$ // gas
  - $p = 70$ // price

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;5, B&gt;$, $g \leftarrow g + 45$</td>
<td>$[g = 45]$</td>
</tr>
<tr>
<td>$&lt;6, B&gt;$, $p \leftarrow p + 70$</td>
<td>$[p = 70]$</td>
</tr>
<tr>
<td>$&lt;7, B&gt;$, $d \leftarrow d + 412$</td>
<td>$[d = 412]$</td>
</tr>
</tbody>
</table>

- **Vector clock B** = $(0, 8)$
- **Order deviation** = 1
- **Numerical deviation** = $(3, 686)$

**Conit (contains the variables $g$, $p$, and $d$)**

- **A** has three **pending** operations $\Rightarrow$ **order deviation** = 3
- **A** missed two operations from **B**; max diff is $70 + 412$ units $\Rightarrow$ $(2, 482)$
Sequential consistency

**Definition**

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

(a) A sequentially consistent data store. (b) A data store that is not sequentially consistent

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

**Notations**

- \( W(x)a \): write value \( a \) to variable \( x \)
- \( R(x)b \): read value \( b \) from variable \( x \)
Causal consistency

Definition

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 by different processes.

(a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store

Why?

P2 implies a causality: $a \Rightarrow b$
Grouping operations

Definition

- Accesses to **locks** are sequentially consistent.
- No access to a lock is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to locks have been performed.
Grouping operations

Definition

- Accesses to locks are sequentially consistent.
- No access to a lock is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to locks have been performed.

Basic idea

You don’t care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.
Grouping operations

A valid event sequence for entry consistency

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

Observation

Entry consistency implies that we need to lock and unlock data (implicitly or not).

Why P2: R(y) NIL?

It needs L(y) first; otherwise it reads a local unsynchronized value (NIL).

Consistency vs. Coherence

Consistency: a **set** of data items
Coherence: a **single** data item
Motivating Example: Consistency for mobile users

Example

Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.

- At location A you access the database doing reads and updates.
- At location B you continue your work, but unless you access the same server as the one at location 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

Note

The only thing you really 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 appear to be consistent to you.
Basic architecture

The principle of a mobile user accessing different replicas of a distributed database

- Client moves to other location and (transparently) connects to other replica
- Replicas need to maintain client-centric consistency
- Wide-area network
- Distributed and replicated database
- Portable computer
- Read and write operations
Monotonic reads

Definition

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.

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. (b) A data store that does not provide monotonic reads

<table>
<thead>
<tr>
<th>L1</th>
<th>W_1(x_1)</th>
<th>R_1(x_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2</td>
<td>W_2(x_1;x_2)</td>
<td>R_1(x_2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1</th>
<th>W_1(x_1)</th>
<th>R_1(x_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2</td>
<td>W_2(x_1</td>
<td>x_2)</td>
</tr>
</tbody>
</table>

Notations

$W_2(x_1;x_2)$: Process #2 produces version $x_2$ based on version $x_1$

$W_2(x_1|x_2)$: Process #2 produces version $x_2$ concurrently to version $x_1$
Monotonic reads

Example
Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

Example
Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.
Monotonic writes

Definition
A write operation by a process on a data item $x$ is completed before any successive write operation on $x$ by the same process.

(a) A monotonic-write consistent data store. (b) A data store that does not provide monotonic-write consistency. (c) Again, no consistency as $WS(x_1|x_2)$ and thus also $WS(x_1|x_3)$. (d) Consistent as $WS(x_1;x_3)$ although $x_1$ has, seemingly, overwritten $x_2$. 

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1:</td>
<td>$W_1(x_1)$</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td>$W_2(x_1;x_2)$</td>
<td>$W_1(x_2;x_3)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(a)</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1:</td>
<td>$W_1(x_1)$</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td>$W_2(x_1</td>
<td>x_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(b)</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1:</td>
<td>$W_1(x_1)$</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td>$W_2(x_1</td>
<td>x_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(c)</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1:</td>
<td>$W_1(x_1)$</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td>$W_2(x_1</td>
<td>x_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(d)</td>
</tr>
</tbody>
</table>
Monotonic writes

Example
Updating a program at server $S_2$, and ensuring that all components on which compilation and linking depends, are also placed at $S_2$.

Example
Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).
Read your writes

Definition

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.

(a) A data store that provides read-your-writes consistency. (b) A data store that does not.

<table>
<thead>
<tr>
<th>L1: $W_1(x_1)$</th>
<th>L1: $W_1(x_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2: $W_2(x_1;x_2)$</td>
<td>L2: $W_2(x_1;x_2)$</td>
</tr>
<tr>
<td>$R_1(x_2)$</td>
<td>$R_1(x_2)$</td>
</tr>
</tbody>
</table>

(a) (b)
Read your writes

Definition
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.

(a) A data store that provides read-your-writes consistency. (b) A data store that does not.

Example
Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.
Writes follow reads

Definition
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.

(a) A writes-follow-reads consistent data store. (b) A data store that does not provide writes-follow-reads consistency

Example
See reactions to posted articles only if you have the original posting (a read “pulls in” the corresponding write operation).
Content replication

Distinguish different processes

A process is capable of hosting a replica of an object or data:

- **Permanent replicas**: Process/machine always having a replica
- **Server-initiated replica**: Process that can dynamically host a replica on request of another server in the data store
- **Client-initiated replica**: Process that can dynamically host a replica on request of a client (client cache)
The logical organization of different kinds of copies of a data store into three concentric rings

- Permanent replicas
- Server-initiated replicas
- Client-initiated replicas
- Clients

→ Server-initiated replication
→ Client-initiated replication
Server-initiated replicas

Counting access requests from different clients

- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold $D \Rightarrow$ drop file
- Number of accesses exceeds threshold $R \Rightarrow$ replicate file
- Number of access between $D$ and $R \Rightarrow$ migrate file
Content distribution

Consider only a client-server combination

- Propagate only notification/invalidation of update (often used for caches)
- Transfer data from one copy to another (distributed databases): passive replication
- Propagate the update operation to other copies: active replication

Note
No single approach is the best, but depends highly on available bandwidth and read-to-write ratio at replicas.
Content distribution: client/server system

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

- **Pushing updates**: server-initiated approach, in which update is propagated regardless whether target asked for it.
- **Pulling updates**: client-initiated approach, in which client requests to be updated.
Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system’s behavior (adaptive leases)
Content distribution

Observation
We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system’s behavior (adaptive leases)

- **Age-based leases**: An object that hasn’t changed for a long time, will not change in the near future, so provide a long-lasting lease
Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system’s behavior (adaptive leases)

- **State-based leases**: The more loaded a server is, the shorter the expiration times become.
Content distribution

Observation
We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system’s behavior (adaptive leases)

- **Age-based leases**: An object that hasn’t changed for a long time, will not change in the near future, so provide a long-lasting lease
- **State-based leases**: The more loaded a server is, the shorter the expiration times become

Why
Trying to reduce the server’s state as much as possible while providing strong consistency.
Primary-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**

Example primary-backup protocol

Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on the same LAN.
Primary-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

**Example primary-backup protocol**

Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on same LAN.
Primary-based protocols

Primary-backup protocol with local writes

1. Write request
2. Move item x to new primary
3. Acknowledge write completed
4. Tell backups to update
5. Acknowledge update

R1. Read request
R2. Response to read
Primary-based protocols

Primary-backup protocol with local writes

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

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

Example primary-backup protocol with local writes

Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).
Replicated-write protocols

Quorum-based protocols

Ensure that each operation is carried out in such a way that a majority vote is established: distinguish read quorum and write quorum.

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).

- For the first example, $N_R = 3$, $N_W = 10$.
- For the second example, $N_R = 7$, $N_W = 6$.
- For the third example, $N_R = 1$, $N_W = 12$.

- $N_R + N_W > N$
- $N_W > \frac{N}{2}$