Consistency and Replication

Topics to be covered

- Introduction
- Consistency Models
- Distribution Protocols
- Consistency Protocols

Introduction

Performance
- Reliability

Availability: proportion of time for which a service is accessible
- Delays due to pessimistic cc
- Server failures
- Network partitions and disconnected operation

\[ P(\text{all failed or unreachable}) = 1 - p^n \]

\( p = 5\%, n = 2 \), availability = 99.75%

Fault tolerance: guarantees strictly correct behavior despite a certain type and number of faults
- Correct (e.g., freshness, timeliness of response)
  - Up to \( f \) of \( f+1 \) crash
  - Up to \( f \) byzantine failures, \( 2f + 1 \)

Requirements

Replication Transparency
One logical object (data item) - many physical copies

Consistency Problems: keep replica consistent - in general, ensure that all conflicting operations (e.g., from the world of transactions: RW, and WW) are executed in the same order everywhere.

Guaranteeing global ordering costly operation, downgrade scalability

Weaken consistency requirements

Object Replication

Organization of a distributed remote object shared by two different clients.

Client machine

Server machine

Client machine
Distributed Systems, Spring 2004

1. **Object Replication**

(a) A remote object capable of handling concurrent invocations on its own.

(b) A remote object for which an object adapter is required to handle concurrent invocations.

2. **Object Replication**

(a) A distributed system for replication-aware distributed objects.

(b) A distributed system responsible for replica management.

3. **Consistency Models**

**Data-Centric**

- **Client-Centric**

4. **Data-Centric Consistency Models**

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute global time ordering of all shared accesses matters.</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes see all shared accesses in the same order. Accesses are further ordered according to a nonunique global timestamp.</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time.</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order.</td>
</tr>
<tr>
<td>FIFO</td>
<td>All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.</td>
</tr>
</tbody>
</table>

5. **Data-Centric Consistency Models**

- There are no explicit synchronization operations.
- There are explicit synchronization operations - updates are propagated only when such operations are used.

6. **Data-Centric Consistency Models**

- **Strict Consistency**

Any read on a data item x returns a value corresponding to the results of the most recent write on x.

Absolute global time

Note: strict consistency is what we get in the normal sequential case, when each program does not interfere with any other program.

Example:

\[
\begin{array}{cccccc}
& W(x) & W(y) & W(z) & R(x) & R(y) \\
(a) & P_1 & & & R_0 & \\
(b) & P_2 & & & R_1 & \\
\end{array}
\]

(a) A strictly consistent store (b) A store that is not strictly consistent.
Data-Centric Consistency Models

Strict Consistency

Problem: It relies on absolute global time.

Example:

\[
\begin{align*}
p_1 & \xrightarrow{W_1(x)a} \quad p_2 \xrightarrow{W_2(x)b} \\
R_3(x) & \xrightarrow{send(R_3(x))} \\
p_1 & \xrightarrow{receive(R_3(x))}
\end{align*}
\]

If strict, \(p_2\) should read the value \(a\).

All writes are instantaneously visible to all processes and an absolute global time order is maintained.

Sequential Consistency

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

Example:

\[
\begin{align*}
p_1 & \xrightarrow{W_1(x)a} \\
p_2 & \xrightarrow{W_2(x)b} \\
p_3 & \xrightarrow{R_3(x)a} \\
p_4 & \xrightarrow{R_4(x)b}
\end{align*}
\]

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

- Note: a process sees writes from all processes but only its own reads
- Similar with (transaction) serializability but difference in granularity (transactions vs single read and write operations)

All processes see the same interleaving of operations.

Distributed Systems, Spring 2004

Sequential Consistency (2nd definition)

A consistency model between strict consistency and sequential consistency that uses loosely synchronized clocks.

In particular, we assume that operations receive a timestamp using a loosely synchronized clock (a finite precision global clock)

Notation: \(ts_{OP}(x)\) where \(OP = R\) or \(W\)

Linearizability

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. In addition, if \(ts_{OP1}(x) < ts_{OP2}(y)\) then operation \(OP1(x)\) should precede \(OP2(y)\) in this sequence.

- A linearizable data store is also sequentially consistent.
- The additional requirements of ordering according to timestamps makes it more expensive to implement.

Distributed Systems, Spring 2004

Ways to Express Consistency

Consider an associated history (execution)

\[
\begin{align*}
P_1 & \xrightarrow{W_1(x)a} \\
P_2 & \xrightarrow{W_2(x)b} \\
P_3 & \xrightarrow{R_3(x)b} \\
P_4 & \xrightarrow{R_4(x)b}
\end{align*}
\]

Merge individual histories to get the execution history \(H\):

\[
\begin{align*}
W_1(x)a & \xrightarrow{W_2(x)b} R_3(x)b \xrightarrow{R_4(x)b}
\end{align*}
\]

Legal history \(H\), if:

- Present program order (order of individual histories)
- A read to \(x\) must always return the value most recently written to \(x\) (data coherency)

Data Coherency

A \(R(x)\) must return the value most recently written to \(x\), that is, the value written by the \(W(x)\) immediately preceding it in \(H\)

Coherence examines each data item in isolation

Called memory coherence when dealing with memory locations instead of data items

Distributed Systems, Spring 2004
### Data-Centric Consistency Models

#### Sequential Consistency

**Example:** Four valid execution sequences for the processes of the previous slide.

<table>
<thead>
<tr>
<th>Process</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>x = 1; y = 1; z = 1; print(x, y); print(x, z);</td>
</tr>
<tr>
<td>P2</td>
<td>x = 1; y = 1; z = 1; print(y, z);</td>
</tr>
<tr>
<td>P3</td>
<td>x = 1; y = 1; z = 1; print(y, z);</td>
</tr>
<tr>
<td>P4</td>
<td>x = 1; y = 1; z = 1; print(y, z);</td>
</tr>
</tbody>
</table>

**Signature:** output of P1 Output of P2 output of P3 – 64 different signatures, valid ones?

90 different valid statement orderings produce a variety of different signatures.

### Casual Consistency

**Example:**

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

(a) A violation of a casually-consistent store. (b) A correct sequence of events in a casually-consistent store. - assume W(x)a depends on W(x)b
Data-Centric Consistency Models

**FIFO Consistency**

- Writes of a single process are seen by all other processes in the order in which they were issued, but writes of different processes may be seen in a different order by different processes.

In other words: There are no guarantees about the order in which different processes see writes, except that two or more writes of the same process must arrive in order (that is, all writes generated by different processes are concurrent).

Also called PRAM consistency in the case of distributed shared memory.

Pipelined RAM

A valid sequence of events of FIFO consistency but not for causal.

**Example:**

Implementation: need just to guarantee that writes from the same process arrive in order, tag writes with (process-id, sequence-number).

Perform writes with the same id based on sequence-number.

---

**Statement execution as seen by the three processes from the previous slide. The statements in bold are the ones that generate the output shown.**

| x = 1; | y = 1; | z = 1; |
| print (y, z); | print (x, z); | print (x, z); |
| y = 1; | y = 1; | y = 1; |
| print (x, z); | print (y, z); | print (y, z); |
| z = 1; | z = 1; | z = 1; |
| print (x, y); | print (y, z); | print (y, z); |

Prints: 00 01 10

(a) P1's view (b) P2's view (c) P3's view

---

**Strong Consistency Models:** Operations on shared data are synchronized:

- **Strict consistency (related to time)**
- **Sequential Consistency (similar to database serializability, what we are used to)**
- **Causal Consistency (maintains only causal relations)**
- **FIFO consistency (maintains only individual ordering)**
Weak Consistency

Don’t care that the reads and writes of a series of operations are immediately known to other processes. Just want the effect of the series itself to be known.

Each process operates on its own local copy of the data store.

Changes are propagated only when an explicit synchronization takes place.

A synchronization variable $S$ with one associated operation $\text{synchronize}(S)$ which synchronizes all local copies of the data store.

When the data store is synchronized all local copies of process $P$ are propagated to the other copies, whereas writes by other processes are brought into $P$’s copies.

Don’t care that the reads and writes of a series of operations are immediately known to other processes. Just want the effect of the series itself to be known.

Weak Consistency

1. Accesses to synchronization variables with a data store are sequentially consistent. (All operations are in all operations on synchronization variables in the same order)
2. No operation on a synchronized variable is allowed to be performed until all previous writes are completed everywhere.
3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

Example

(a) A valid sequence of events for weak consistency. (b) An invalid sequence for weak consistency.

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

Release Consistency

Divide access to a synchronization variable into two parts: an acquire (for entering a critical region) and a release (for leaving a critical region) phase.

Acquire forces a requestor to wait until the shared data can be accessed.

Release sends requestor’s local value to other servers in data store.

1. When a process does an acquire, the store will ensure that all the local copies of the protected data are brought up to date.
2. When a release is done, protected data that have been changed are propagated to all local copies of the store.

Example

A valid event sequence for release consistency.

Release Consistency

1. Before a read or write operation on shared data is performed, all previous acquires done by the process must have been completed successfully.
2. Before a release is allowed to be performed, all previous reads and writes done by the process must have been completed.
3. Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).
Data-Centric Consistency Models

Entry Consistency

With release consistency, all local updates are propagated to other copies/servers during release of shared data.

With entry consistency, each shared data item is associated with a synchronization variable.

When acquiring the synchronization variable, the most recent of its associated shared data are fetched.

Whereas release consistency affects all data, entry consistency affects only those shared data associated with a synchronization variable.

Weak Consistency Models: Synchronization occurs only when shared data are locked and unlocked:

- General Weak Consistency
- Release Consistency
- Entry consistency

barriers: synchronization mechanism that prevents any process from starting phase n+1 until all processes have finished phase n

When a process reaches a barrier, it must wait others to get there. When all arrive, data are synchronized, all processes are resumed.

(a)

Concurrent Consistency: All processes see writes from each other in the order they were used. Write from different processes may not always be seen in that order.

FIFO

All processes see writes from each other in the order they were used.

(b)

Concurrent Consistency: All processes see all shared accesses in the same order. Accesses are furthermore ordered according to a nonunique global timestamp.

Linearizability

All processes see all shared accesses in the same order.

Sequential

All processes see all shared accesses in the same order. Accesses are not ordered in time.

Causal

All processes see causally-related shared accesses in the same order.

FIFO

All processes see writes from each other in the order they were used. Write from different processes may not always be seen in that order.

Entry Consistency

1. Any 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.

2. 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.

3. After an exclusive mode access to a 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.

Example

P1: Acq(a) W(x) R(y) Acq(b) R(a, b) Acq(y) Re(y) Re(x)

P2: Acq(a) R(x, b) Re(b)

P3: Acq(b) R(x, y)

A valid event sequence for entry consistency.

Show how we can avoid system-wide consistency, by concentrating on what specific clients want, instead of what should be maintained by the servers.

Eventual consistency: if no updates take place for a long time, all replicas will gradually become consistent.
The principle of a mobile user accessing different replicas of a distributed database.

**Client-Centric Consistency Models**

**Monotonic Reads**

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

**Notation**

WS(x[t]): the set of write operations (at site L_i) that lead to version x(t) at time t.

WS(x[t1];x[t2]) indicates that WS(x[t1]) is part of WS(x[t2]).

Example: reading incoming email while on the move: each time you connect, read monotonically.

**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.

**Example**

May not reflect updates at site L1.

**Read Your Writes**

The effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process.

**Example**

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

Example: updating your web page and guaranteeing that your web browser shows the newest version instead of the cached copy.

Similar with changing passwords.
Example: 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.

\[
\begin{array}{c|c|c}
L1: & WS(\delta) & R(x1) \\
L2: & WS(x2) & R(x2) \\
\end{array}
\]

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

Example: See reactions to posted articles only if you have the original posting (a read "pulls in" the corresponding write operation)

---

**Replica Placement**

- Where, when and by whom data copies are placed in a distributed system?

**Permanent Replicas**

The initial set of replicas that constitute a distributed data store.

**Server-Initiated Replicas**

Copies of data to enhance performance, created by the servers.

\[
\begin{array}{c}
\text{Client initiated replicas} \\
\end{array}
\]

- Keep track of access counts per file plus the client that requested the file aggregated by considering server closest to requesting clients.
- Number of accesses drop below threshold D: drop file
- Number of accesses exceed threshold R: replicate file
- Number of accesses between D and R: file can only be migrated (no drop or replication), when? If the requests from a specific server exceed half of the total requests.

Example, when two clients (C1 and C2) share the same closest server (P)

**Client-Initiated Replicas**

Client initiated replicas or (client) caches
General kept for a limited amount of time (replaced or become stale)
Cache hit
Share cache among clients
Normally placed at the same machine as the client.
Update Propagation

**State vs Operation**

- Propagate only notification/invalidation of update
  - Often used for caches
  - Called invalidation protocols
  - Works well when read-to-write ratio is small
- Transfer values/copies from one copy to the other
  - Works well when read-to-write ratio is relatively high
- Log the changes, aggregate updates
- Propagate the update operation to other copies (aka active replication)
  - Less bandwidth, more processing power

**Push vs Pull**

<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>Invalidation (and possibly update)</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>

**A Hybrid Protocol: Leases**

- **Lease**: A contract in which the server promises to push updates to the client until the lease expires
  - Make lease expiration time depends on system behavior (adaptive leases)
  - **Age-based leases**: An object that has not changed for long time, will not change in the near future, so provide a long-lasting lease
  - **Renewal-frequency based leases**: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
  - **State-based leases**: The more loaded a server is, the shorter the expiration times become

**Overview**

**Epidemic Algorithms**

- Basic idea: assume there are no write-write conflicts (e.g., updates for a specific item are initiated at a single server)
  - Update operations are initially performed at one or only a few replicas
  - A replica passes its updated state to a limited number of neighbors
  - Update propagation is lazy, i.e., not immediate
  - Eventually, each update should reach every replica

**Anti-entropy**

- Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards

**Gossiping**

- A replica that has just been updated (i.e., has been contaminated) tells a number of other replicas about its update (contaminating them as well)

**System Model**

- A collection of servers, each storing a number of objects
  - Each object O has a primary server at which updates for O are initiated
  - An update of an object O at server S is timestamped

**Notation**

- timestamp: \( T(O, S) \)
- value: \( VAL(O, S) \)

**Infected server/susceptible server**

**Anti-Entropy**

- A server S picks another server S' randomly and exchange updates with it
  - When S contacts S', S exchanges state information, three strategies:
    - **PUSH**: S only forwards all its updates to S'
      - if \( T(O, S') > T(O, S) \) then \( VAL(O, S') = VAL(O, S) \)
    - **PULL**: S only fetches updates from S'
      - if \( T(O, S') > T(O, S) \) then \( VAL(O, S) = VAL(O, S') \)
    - **PUSH&PULL**: S both forwards and fetches updates
      - S' and S exchange by pushing and pulling values

- If each server randomly chooses another server for exchanging updates, an update is propagated in \( O\log(N) \) time units

- Why pushing alone is not efficient when many servers have already been infected?
Gossiping

A server $S$ having an update to report, contacts other servers. If a server is contacted to which the update has already been propagated, $S$ stops contacting other servers with probability $1/k$.

If $s$ is the fraction of susceptible servers (i.e., which are unaware of the update), it can be shown that with many servers:

$$ s = 1 - e^{-(k+1)} $$

$k$  $s$
---  ---
1  0.2
2  0.06
3  0.02
4  0.007
5  0.0025

If we really have to ensure that all servers are eventually updated, gossiping alone is not enough.

Deleting Values

We cannot remove an old value from a server and expect the removal to propagate. Why?

Treat removal as a special update by inserting a death certificate.

When to remove a death certificate:

- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection).
- Assume that death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at the risk of not reaching all servers).

It is necessary that a removal actually reached all servers.

Consistency Protocols

Primary-Based Protocols

Replicated-Write Protocols

Cache-Coherence Protocols

Client-Centric Consistency Models

Implementation

Use timestamps and maintain read and write sets.

Sessional
Primary-Based Protocols

Remote-Write protocols

Primary-backup protocol: reads on local copies, but writes at a (fixed) primary copy

An updated is applied as a blocking operation.
Sequential consistency? Why?
Non-blocking write variant: as soon as the primary has updated its local copy, it returns an ack, then it tells the backup to perform the update as well. Consistency?

Local-Write protocols

Case 1: there is only a single copy of each data item x (no replication) a single copy is migrated between processes

Useful when writes are expected to come in series from the same client (e.g., mobile computing without replication)

Case 2 (primary back-up) the primary copy migrates

distributable shared memory systems, but also mobile computing in disconnected mode

Active Replication: updates are propagated to multiple replicas where they are carried out.

The problem of replicated invocations.

Replicated-Write Protocols

(a) Forwarding an invocation request from a replicated object. (b) Returning a reply to a replicated object

Assign a coordinator on each side (client and server) that ensures that only one invocation and one reply is sent

Three examples of the voting algorithm:

1. 
2. 

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)
Cache Consistency Protocols

- Write-through caches: clients directly modify the cached data and forward the update to the servers.
- Write-back caches: delay the propagation of updates by allowing multiple writes to take place.

Casually-Consistent Lazy Replication

Number of replica servers jointly implement a causal-consistent data store. Clients normally talk to front ends which maintain data to ensure causal consistency (eventual consistency but also causal relationships between operations).

The general organization of a distributed data store. Clients are assumed to also handle consistency-related communication.

Vector Timestamps

Two vector timestamps per local copy $L_i$:
- $VAL(i): VAL(i)[i]$ denotes the total number of write operations sent directly by a front end (client).
- $VAL(i)[j]$ denotes the number of updates sent from local copy $L_j$.
- $WORK(i): WORK(i)[i]$ total number of write operation directly from front ends, including the pending ones.
- $WORK(i)[j]$ is the total number of updates from local copy $L_j$, including pending ones.
- $LOCAL(C): LOCAL(C)[j]$ is (almost) most recent value of $VAL(j)$ known to front end $C$ (will be refined).
- $DEP(R)$: Timestamp associated with a request, reflecting what the request depends on.

Performing a read operation at a local copy.

Performing a write operation at a local copy.