Replication & Consistency


Hongfei Yan
School of EECS, Peking University
4/7/2008

Outline
- Introduction (what’s it all about)
- Data-centric consistency
- Client-centric consistency
- Replica management
- Consistency protocols

Replication
Replication: Creating and using multiple copies of data or services

Why replicate?
- Improve reliability
  - Data survival
  - Availability
  - Increase confidence: e.g. deal with byzantine failures
- Improve performance
  - Scaling
  - Reduce access times

Data vs. Computation
Replication:
- It could be data replication if the same data is stored on multiple storage devices
  - Eg. Web site mirror, browser cache, DNS
- or computation replication if the same computing task is executed many times.

What are the issues?
Updating replicas
- Consistency (how to deal with updated data)
  - (luckily) applications do not always require strict consistency
  - How are updates distributed?
Replica management
- How many replicas?
- Where to place them?
- When to get rid of them?
Redirection/Routing
- Which replica should clients use?

Example: Costs
- As a scaling technique, may not always be applicable.

Scalability ← CONFLICT management overheads

As a scaling technique, may not always be applicable. What if N << M?
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

- **Problem:** 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

Client’s view of data-store

Ideally: black box -- complete ‘transparency’ over how data is stored and managed

Management system view on data store

Management system: Controls the allocated resources and aims to provide transparency

In the Following

- **Consistency model:** Contract between the data store and the clients
  - the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

- **Protocols**
  - To manage replicas: creation, placement
  - To manage updates: propagation
  - To assign client requests to replicas

Data Centric Consistency Models

- **Operations on a Data Store**

  **Notations:**
  - Read: $R_i(x) a$ -- client $i$ reads $a$ from location $x$
  - Write: $W_i(x) b$ -- client $i$ writes $b$ at location $x$
Strict Consistency

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

- Implicitly assumes the existence of absolute global time.
- Naturally available in uni-processor systems, but impossible to implement in distributed systems.

Behavior of two processes, operating on the same data item.

Sequential Consistency

The result of any execution is the same as if
- 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.

Note: we talk about interleaved execution - there is some total ordering for all operations taken together.

Still expensive: \( R + W > T \)

Linearizability Consistency

A data store is linearizable if it provides:
- Sequential consistency, and
- When assuming some limited granularity global clock
  if \( TS_{w1}(x) < TS_{w2}(y) \), then operation \( OP1(x) \) should precede \( OP2(y) \)

Causal Consistency

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.

Causally related relationship:
- A read is causally related to the write that provided the data the read got.
- A write is causally related to a read that happened before this write in the same process.
- If write1 → read, and read → write2, then write1 → write2.

Concurrent \( \leftrightarrow \) not causally related

Causal Consistency (Example)

This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Note: \( W_1(x) \rightarrow W_2(x) \), but not \( W_2(x) \rightarrow W_1(x) \)

Causal Consistency (More Examples)

A violation of a causally-consistent store.

A correct sequence of events in a causally-consistent store.
FIFO Consistency

Causally related writes done by a single process are seen by all other processes in the order in which they were issued.

(Æ but writes from different processes may be seen in a different order by different processes.)

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

Grouping Operations

- As viewed by an external, data-centric process, what do locks do?
  - They turn non-atomic operations into atomic ones (functionally).
  - In other words, they group them.

- Synchronization Variables
  - Operations are grouped via synchronization variables (locks).
  - Each sync var protects an associated data.
  - Each kind of sync var has some associated properties.

Weak Consistency

**Intuition:** 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.

Weak consistency properties

- Accesses to synchronization variables associated with a data store are sequentially consistent.
- No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.
- No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

**Weak Consistency (Example)**

P1: W(x)a W(x)b S
P2: R(x)a R(x)b S
P3: R(x)a S R(x)a

A valid sequence of events for weak consistency.

**Release Consistency**

**Intuition:** Weak consistency uses the same primitive to synchronize before reading and writing the data Æ introduce locking primitives acquire and release

Rules for release consistency:

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

**Release Consistency - Example**

P1: Acq(L) W(x)a W(x)b Rel(L)
P2: Acq(L) R(x)b Rel(L)
P3: R(x)a

A valid event sequence for release consistency.
Entry Consistency

Intuition: for release consistency data is synchronized when lock is released for all variables.

Solution:
- Variable specific synchronization
- Synchronize at entry only

Rules:
- An acquire access of a synchronisation 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 synchronisation variable by a process is allowed to perform with respect to that process, no other process may hold the synchronisation variable, not even in nonexclusive mode.
- After an exclusive mode access to a synchronisation variable has been performed, any other process’s next nonexclusive mode access to that synchronisation variable may not be performed until it has performed with respect to that variable’s owner.

Entry Consistency (Example)

P1: Acq(Lx) Wx(y) Acq(Ly) Wy(b) Rel(Lx) Rel(Ly)

P2: Acq(Lx) Rx(a) Ry(b) IL

P3: Acq(Ly) Rx(b) Ry(b)

A valid event sequence for entry consistency.

Summary - data centric consistency

Consistency models not using explicit synchronization operations.

<table>
<thead>
<tr>
<th>Strict</th>
<th>Absolute time ordering of all shared accesses matters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Plus accessed are 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 issued. Nothing is guaranteed about writes from different processes.</td>
</tr>
</tbody>
</table>

Models with explicit synchronization operations.

| Weak | Shared data can be counted on to be consistent only after a synchronisation is done. |
| Release | Shared data are made consistent when a critical region is exited (or entered). |
| Entry | Shared data pertaining to a critical region are made consistent when a critical region is entered. |

Client-Centric Consistency Models

- System model
- Monotonic reads
- Monotonic writes
- Read-your-writes
- Write-follows-reads

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

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.
  - Idea: the database will appear to be consistent to the user.

Example: Basic Architecture

- How well does EC work for mobile clients?
- Client-centric is for this. Consistent for a single client.
Eventual Consistency

If no updates take place for a long enough period time, all replicas will gradually (i.e., eventually) become consistent.

- Situations where eventual consistency models may make sense
  - Mostly read-only workloads
  - No concurrent updates

- Advantages/Drawbacks

Client-centric Consistency

Idea: Guarantees some degree of data access consistency for a single client.

Notations:
- \( X_i(t) \) → Version of data item \( x \) at time \( t \) at local copy \( L_i \)
- \( WS(x_i[t]) \) → all write operations at \( L_i \) since init
- \( WS(x_i[t], x_j[t]) \) → indicates that it is known that \( WS(x_i[t]) \) is part of \( WS(x_j[t]) \)

Monotonic-Read Consistency

**Def:** 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.

**Intuition:** Client “sees” only same or newer version of data.

Monotonic-Write Consistency

**Def:** A write operation by a process on a data item \( x \) is completed before any successive write operation on \( x \) by the same process.

**Intuition:** Write happens on a copy only if it’s brought up to date with preceding write operations on same data (but possibly at different copies)

Monotonic reads - Examples

- 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.

Monotonic writes - Examples

- Updating a program at server \( S_2 \), and ensuring that all components on which compilation and linking depends, are also placed at \( S_2 \).
- 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 Consistency

**Def:** 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.

**Intuition:** All previous writes are always completed before any successive read.

```
L1: W(x)
L2: W(x, y)
R(x)
```

Read-Your-Writes - Examples

- Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.
- Password database

Writes-Follow-Reads Consistency

**Def:** 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.

**Intuition:** Any successive write operation on \( x \) will be performed on a copy of \( x \) that is same or more recent than the last read.

```
L1: W(x)
L2: W(x, y)
R(x)
```

Writes-Follow-Reads - Examples

- Examples: news groups
  - See reactions to posted articles only if you have the original posting (a read pulls in the corresponding write operation).

Implementation of monotonic-read consistency

**Sketch:**
- globally unique identifier for each write operation
- each client has two sets:
  - write ids relevant to the read operations
  - write ids relevant to the write operations
- the server updates \( x \) according to the client’s read set before the operation

Sketch a Design for a Consistency Model Implementation

- Monotonic-writes
- Read-your-writes
- Writes-follow-reads
Summary - client centric consistency

- **Main take**
  - We can avoid system-wide consistency, by concentrating on what specific clients want, instead of what should be maintained by servers. Relax consistency requirements even further.
  - Only concerned about single client-view

Replica Management

- Replica server placement
- Content replication and placement
- Update distribution/propagation

Replica Server Placement

**Essence:** Figure out what the best $K$ places are out of $N$ possible locations.

- Select best location out of $N - k$ for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location minimizes the average distance to all clients.) Computationally expensive.
- Select the $k$-th largest autonomous system and place a server at the best-connected host. Computationally expensive.
- Position nodes in a $d$-dimensional geometric space, where distance reflects latency. Identify the $K$ regions with highest density and place a server in every one. Computationally cheap.

Clustering

- One idea, identify the $K$ largest clusters, then put one server in each cluster.
- How do you find clusters?
  - One way, divide space up into cells, pick $K$ most populated ones.

Content replication

**Model:** We consider objects (and don't worry whether they contain just data or code, or both)

**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)
Content Replication (contd.)

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

Content Replication: Server-Initiated Replicas

- 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

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

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

Propagating updates (1/3)

- 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.

<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>

Propagating updates: Leases (2/3)

Observation: We can dynamically switch between pull and push using leases.

- Lease: A contract in which the server promises to push updates to the client until the lease expires.

Issue: 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
- 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

Propagating updates: Epidemic protocols (3/3)

- Problem so far: server scalability
  - Server needs to provide updates to all system participants
  - What if Internet-scale system (P2P)?
- Epidemic protocols:
  - Nodes periodically pair up and exchange state (or updates)
  - Push and/or pull exchanges
- Issues
  - Generated traffic – mapping on physical topology
  - When is an update distributed to everyone?
  - Probabilistic guarantees
  - Freeriding
- Use: server-less environments, volatile nodes
  - mobile networks, P2P systems
Consistency Protocols

- Primary-Based Protocols
- Replicated-Write Protocols

Primary-Based Protocols (1/2)

- Primary-backup protocol with remote writes:

Example: 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 (2/2)

- Primary-backup protocol with local writes:

Example: 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:

Summary

- Introduction (what’s it all about)
- Data-centric consistency
- Client-centric consistency
- Replica management
- Consistency protocols

References

- Chapter 6 of [Tanenbaum, 2002]
- Or Chapter 8 of [Xining Li, 2006]
  - book1, [Coulouri, 2005]
  - book2, [Birman, 2005]
  - book3, [Tanenbaum, 2006]
  - book4, [Tanenbaum, 2002]
  - Book5, [Xining Li, 2006]