Consistency and Replication
(part a)

EECS 591
Farnam Jahanian
University of Michigan
Reading List

• Tanenbaum Chapter 6.1-6.5
Why replicate?

- Enhance reliability or availability

- Improve performance --- replication as a scaling technique

- Major issue:
  - Multiple copies \(\rightarrow\) replica consistency problem

- Example: consider access to popular web sites
  - Replicated servers
  - Caches copies: client cache, AKAMAI’s model
  - Read vs. update
Case Study

• Akamai’s architecture
• 10000+ cache servers
• Where are they deployed? How’s different from proxy caches?
• What’s the benefit?
  – Content provider, e.g. CNN
  – End user/networks
  – Regional providers
  – Transit providers
Replication as a scaling technique

- Replication and caching for performance
  - Replicate servers to handle load --- computational capacity
  - Replicate to keep data close to clients --- reduce bandwidth and access time

- Trade-off:
  - Keeping cache copies reduces network bandwidth
  - keeping copies consistent may require more network bandwidth
  - Access (read) to update (write) ratio

- Another issue:
  - Keeping copies consistent may be subject to scalability problems
  - Consistency semantics: keeping copies consistent may require global synchronization
  - In practice, loosen consistency based on the access and update patterns and the intended use of the data.
Consistency Models

- Used to describe formally the semantics of read & write operations on shared data (called data store)
- Applicable to
  - Distributed shared memory systems (discussed later)
  - Parallel processors
  - Distributed databases
  - Distributed file systems
  - ...
- A data store may be physically distributed across multiple machines
- Consistency model: a contract between process and the shared data store; if the processes agree to obey certain rules, the data store provides a precise semantics for read/write operations in the absence of a global clock.
- Why needed?
Data-Centric Consistency Models

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

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

Most stringent consistency model
Assumes absolute global time and ensures that all writes are instantaneously visible to other processes

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

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

Definition: *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 order of operations of each individual process (as specified by its program) is preserved in this sequence.*

Weaker than strict consistency model
Sequential Consistency

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

• Formal Definition of Sequential Consistency:
  – Each process Pi has an associated execution Ei, which is a sequence of r/w operations
  – H (called a history) is a merged (interleaved) sequence of operations of all processes in the system
  – i.e. H gives the order that operations would have been executed had there been a single centralized data store.

  – All legal histories must obey two constraints:
    • Program order must be maintained
    • Data (or memory) coherence must be respected

• What’s data (or memory) coherence?
  – A read R(x) must always return the values most recently written to x
  – Data coherence examines examines the sequence of operations on each data item in isolation; data consistency deals with the ordering of read/write operations to different data items
Sequential Consistency

• Comparing sequential consistency and serializability in DB transactions:
  – Main difference is granularity: sequential consistency is defined in terms of read and write ops; serializability is defined in terms of transactions (a collection of operations).
### Sequential Consistency

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

Three concurrently executing processes.
Sequential Consistency

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

Prints: 001011  Prints: 101011  Prints: 010111  Prints: 111111

Signature: 001011  Signature: 101011  Signature: 110101  Signature: 111111
(a)          (b)          (c)          (d)

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

Necessary condition:
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.

Vector timestamp can be used to implement causal consistency.
Causal Consistency

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

(a) A violation of a Causally-consistent store.

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

\[
\begin{array}{l}
P1: W(x)a \\
P2: \quad R(x)a \quad W(x)b \\
P3: \quad R(x)b \quad R(x)a \\
P4: \quad R(x)a \quad R(x)b \\
\end{array}
\]
Necessary Condition:
Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

Easy to implement by tagging each operation with a (process, seq#) pair.
FIFO Consistency

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

A valid sequence of events of FIFO consistency
FIFO Consistency

\[
x = 1; \\
\text{print } (y, z); \\
y = 1; \\
\text{print}(x, z); \\
z = 1; \\
\text{print } (x, y); \\
\text{print } (x, z); \\
\text{print } (y, z); \\
\text{print } (x, y); \\
\text{print } (x, y); \\
\text{print } (y, z);
\]

Prints: 00  Prints: 10  Prints: 01

(a)     (b)     (c)

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

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
</tr>
<tr>
<td>if (y == 0) kill (P2);</td>
<td>if (x == 0) kill (P1);</td>
</tr>
</tbody>
</table>

Two concurrent processes.

x and y are initialized to zero.

With FIFO consistency, one possible execution kills both P1 and P2.
Weak Consistency

- Previous consistency models enforce consistency on individual reads and writes.
- Weaker semantics is possible by enforcing consistency on groups of operations, e.g. critical section.

- Weak consistency:
  - Synchronization variable S
  - Programmer explicitly specifies synchronization variable
  - Sequential consistency is enforced between groups of operations delimited by a synchronization variable
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 (i.e. flush the pipeline).
- No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed. (i.e. get the most recent value)
Weak Consistency

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

(b) An invalid sequence for weak consistency.
Release Consistency

- Problem with weak consistency: when a sync variables is accessed, don’t know whether it is being done because a process is done writing the shared data or it is about to read it.

- Release consistency:
  - Two synchronization variables: acquire and release
  - Blocks of operations on data are made atomic by the acquire and release primitives to prevent interleaving
  - Programmer explicitly inserts synchronization variables in the code
  - Acquire: all local copies of shared data are brought up to date first
  - Release: shared data that have been changed locally are propagated to remote copies
Release Consistency

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.
Release Consistency

Rules:

• 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).
Entry Consistency
(skip)

Conditions:

• 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 (skip)

A valid event sequence for entry consistency.

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y)NIL
P3: Acq(Ly) R(y)b
Summary of Consistency Models

(a)

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters.</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore 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>

(b)

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Shared data can be counted on to be consistent only after a synchronization is done</td>
</tr>
<tr>
<td>Release</td>
<td>Shared data are made consistent when a critical region is exited</td>
</tr>
<tr>
<td>Entry</td>
<td>Shared data pertaining to a critical region are made consistent when a critical region is entered.</td>
</tr>
</tbody>
</table>

a) Consistency models not using synchronization operations.

b) Models with synchronization operations.