Consistency and Replication

Chapter 6

Reasons for replication:
Replicate: 같은 서비스를 해 주는 게 여러 개가 존재
Reliability: possible to continue working after one replica crashes
Performance: improve by replicating the server or dividing the work
Consistency: 여러 개의 replica들의 상태가 동일해야 한다. 한 쪽에서 update가 일어나면 다른 쪽에서도 update 되어야 한다.
Object replication: important in many distributed systems.

**Object Replication (1)**

![Diagram of object replication](image)

Organization of a distributed remote object shared by two different clients.
Object Replication (2)

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
Object Replication (3)

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

b) A distributed system responsible for replica management
Data-Centric Consistency Models

Each process that can access data from the store is assumed to have a local (or nearby) copy available of the entire store.

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

Any read on a data item \( x \) returns a value corresponding to the result of the most recent write on \( x \).

\( \text{한쪽이 수정할 때마다, 금방 모든 프로세스에게 보여야 한다. 마치 한 군데만 r/w하긴 숨기기이—very ideal case, but eg: Behavior of two processes, operating on the same data item.} \)

\[
\begin{array}{c}
\text{P1: } W(x) a \\
\text{P2: } R(x) a
\end{array} \quad \begin{array}{c}
\text{P1: } W(x) a \\
\text{P2: } R(x) \text{NIL} \quad R(x) a
\end{array}
\]

\( \text{(a)} \) \( \text{(b)} \)

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

- A strictly consistent store.
- A store that is not strictly consistent.

Problem: (b), P2 does a read after the write and gets NIL. A subsequent read returns a. Such behavior is incorrect for a strictly consistent data store.
Linearizability and Sequential Consistency (1)

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.

Program 돌아가는 순서가 interleaving발 턴데, 그 결과가 하나씩 몰린 결과와 같다.

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

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

Problem: b) to P3, it appears as if the data item has first been changed to b, and later to a, but P4 indicate the final value is b.

P3와 P4의 순서가 바뀌었다.
Linearizability and Sequential Consistency (2)

**Linearizability Consistency (each operation is timestamped):** weaker than strict consistency but stronger than sequential consistency. The result of any execution is the same as if the 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 $OP(y)$ in this sequence.

Timestamp를 사용 timestamp 순서는 지켜줘야 함 → finite precision

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

possible execution sequences is $720(6!)$, but only for total of 90 valid execution sequences.

Three concurrently executing processes.
## Linearizability and Sequential Consistency (3)

Broadcast의 total order만 지켜지면 Sequential consistency는 지켜진다.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>x = 1;</td>
<td>y = 1;</td>
<td>y = 1;</td>
</tr>
<tr>
<td>print ((y, z));</td>
<td>y = 1;</td>
<td>z = 1;</td>
<td>x = 1;</td>
</tr>
<tr>
<td>y = 1;</td>
<td>print (x, z);</td>
<td>print (x, y);</td>
<td>print (x, z);</td>
</tr>
<tr>
<td>print (x, z);</td>
<td>print (y, z);</td>
<td>print (x, z);</td>
<td>print (y, z);</td>
</tr>
<tr>
<td>z = 1;</td>
<td>z = 1;</td>
<td>x = 1;</td>
<td>print (x, y);</td>
</tr>
<tr>
<td>print (x, y);</td>
<td>print (x, y);</td>
<td>print (y, z);</td>
<td>print (x, y);</td>
</tr>
</tbody>
</table>

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

Signature:  
001011  
101011  
110101  
111111

(a)  
(b)  
(c)  
(d)

### Four valid execution sequences for the processes of the previous slide. The vertical axis is time.
Casual Consistency (1)

Necessary condition:
Rewrites that are potentially casually related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

Happened-before relation있는 경우에만 따진다, compiler 가 optimization할 때, 상관 관계가 없는 것들은 순서에 상관없이 같이 compile 하듯이
Casual Consistency (2)

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

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

the writes W2(x)b and W1(x)c are concurrent, so not required that all processes see them in the same order
Casual Consistency (3)

P1: W(x)a

P2: R(x)a W(x)b

P3: R(x)b R(x)a

P4: R(x)a R(x)b

(a)

P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)a R(x)b

(b)

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

(b) A correct sequence of events in a casually-consistent store.
FIFO Consistency (1)

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.

다른 progress에서 write한 것은 순서 지키지 않아도 된다. 같은 progress에서의 write만 지켜주면 된다. 실제 실현이 쉽다.
**FIFO Consistency (2)**

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

A valid sequence of events of FIFO consistency

*a*는 어디에 와도 상관 없다. $b \rightarrow c$ 순서만 지키면 된다.
FIFO Consistency (3)

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 (4)

<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.

it is impossible with sequential consisteny→둘 중 하나는 산다.
FIFO→둘 다 서로는 죽어버릴 수도 있다.
Weak Consistency (1)

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.
A program fragment in which some variables may be kept in registers.
Weak Consistency (3)

P1 does two writes to a data item and then synchronizes. If p2 and p3 have not yet been synchronized, no guarantees are given about what they see, so this sequence is valid.

\[
\begin{array}{c}
\text{P1: } W(x)a & W(x)b & S \\
\text{P2: } & R(x)a & R(x)b & S \\
\text{P3: } & R(x)b & R(x)a & S \\
\end{array}
\]

(a)

P2 has been synchronized, it means that its local copy of the data store is brought up to date. When it read x, it must get the value b.

\[
\begin{array}{c}
\text{P1: } W(x)a & W(x)b & S \\
\text{P2: } & S & R(x)a \\
\end{array}
\]

(b)

a) A valid sequence of events for weak consistency.

b) An invalid sequence for weak consistency.
Release Consistency (1)

Lock과 비슷하다. 이 variable을 나만 쓰겠다. Release했을 때, 내가 update한 것 broadcast.

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

A valid event sequence for release consistency.

It is guaranteed to get the value that x had at the time of the release, namely b.
Release Consistency (2)

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 (1)

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 (1)**

<table>
<thead>
<tr>
<th>P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2:</td>
</tr>
<tr>
<td>Acq(Lx) R(x)a R(y)NIL</td>
</tr>
<tr>
<td>P3:</td>
</tr>
<tr>
<td>Acq(Ly) R(y)b</td>
</tr>
</tbody>
</table>

A valid event sequence for entry consistency.

P2 does an acquire for x but not for y, so that it will read value a for x, though may read NIL for y, but it doesn’t matter.
## Summary of Consistency Models

<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>

(a)

<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>

(b)

- **a)** Consistency models not using synchronization operations.
- **b)** Models with synchronization operations.
Eventual Consistency

Eventually update를 멈추었다면 어느 시간이 지나면 모든 copy에게 update가 갈 것이다.

The principle of a mobile user accessing different replicas of a distributed database.
Monotonic Reads

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.
Monotonic Writes

한번에 write하면, write 순서가 어디서나 지켜야 한다.

![Diagram](image)

The write operations performed by a single process \( P \) at two different local copies of the same data store

\[
\begin{align*}
L1: & \quad W(x_1) \\
L2: & \quad W(x_1) \quad W(x_2)
\end{align*}
\]

(a)

\[
\begin{align*}
L1: & \quad W(x_1) \\
L2: & \quad W(x_2)
\end{align*}
\]

(b)

a) A monotonic-write consistent data store.
b) A data store that does not provide monotonic-write consistency.
Read Your Writes

L1: \( W(x_1) \)
L2: \( WS(x_1; x_2) \quad R(x_2) \)

(a)

L1: \( W(x_1) \)
L2: \( WS(x_2) \quad R(x_2) \)

\( w_1 \) 값이 반영하지 않다.

(b)

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

\[ \begin{align*}
L1: & \quad WS(x_1) \quad \quad R(x_1) \\
L2: & \quad WS(x_1; x_2) \quad \text{전달 된다} \quad W(x_2)
\end{align*} \]

(a)

\[ \begin{align*}
L1: & \quad WS(x_1) \quad \quad R(x_1) \\
L2: & \quad WS(x_2) \quad \text{전달 되지 않아} \quad W(x_2)
\end{align*} \]

(b)

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

• Monotonic read:
  – When a client performs a read at a server, that server is handed the client’s read set to check whether all the identified writes have been taken place locally.
  – If not, the read is postponed or forwarded to the other server.

• Monotonic write:
  – When a client performs a write at a server, that server is handed the client’s write set to check whether all the identified writes have been performed first and in correct order.

• Read your writes:
  – When a client performs a read at a server, that server is handed the client’s write set to check whether all the identified writes have been taken place locally.

• Write follow reads:
  – When a client performs a write at a server, that server is handed the client’s read set to check whether all the identified writes have been taken place locally.
Improving Efficiency

Problem:
- The read set and write set can be prohibitively large.

Solution:
- Employ vector timestamp
  - Whenever a server accepts a new write operation $W$, a globally unique identifier $WID$ is assigned along with a timestamp $ts(WID)$.
  - Each server $S_i$ maintains a vector timestamp $RCVD(i)$, where $RCVD(i)[j]$ is equal to the timestamp of the latest write operation initiated at server $S_j$ that has been received by $S_i$.
  - Whenever a client issues a request, the server returns its current timestamp.
  - Read and write sets are subsequently represented by vector timestamps.
  - $VT(A)[i]$ is set to the maximum timestamp of all operations in $A$ that were initiated at server $S_i$. 
Replica Placement

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

Migration: 반 이상의 request가 어떤 server에서 왔다면 자기가 file을 갖고 있을 필요가 없다.
제일 많이 request를 보낼 쪽에 replicate, if replica exist delete

Counting access requests from different clients.
## Pull versus Push Protocols

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

<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>
Remote-Write Protocols (1)

Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.
Remote-Write Protocols (2)

The principle of 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

Client

Primary server for item x

Backup server

Data store

혼자만 write하지 말고 backup에게 update하라고 한다.

여기 끝나야 client에게 ack 보냄
Primary-based local-write protocol in which a single copy is migrated between processes.
Local-Write Protocols (2)

Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

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
The problem of replicated invocations.
Active Replication (2)

- a) Forwarding an invocation request from a replicated object.
- b) Returning a reply to a replicated object.
Quorum-Based Protocols

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)
ORCA – a distributed object system

- High degree of transparency:
  - Strong consistency + distributed objects.

- DSM with replicated objects
  - Shared data + fork()
  - ORCA language + compiler + runtime
  - Similar to Modula 2
  - Type secure with no pointer, no aliasing, no inheritance
    - Array bound check at runtime

- Operation: (guard + block of statements)
  - Guard: boolean expression with no side effect
    - Evaluated when the operation is invoked
    - Operation is executed if guard is true
  - Atomic and sequentially consistent on shared objects
  - Synchronization using critical region
A simplified stack object in Orca, with internal data and two operations.
Management of Shared Objects in Orca

Four cases of a process $P$ performing an operation on an object $O$ in Orca.
The general organization of a distributed data store. Clients are assumed to also handle consistency-related communication.
Processing Read Operations

1. $	ext{DEP}(R) := \text{LOCAL}(C)$
2. $	ext{DEP}(R) \leq \text{VAL}(i)$
3. Data & $\text{VAL}(i)$
4. $	ext{LOCAL}(C) := \max\{\text{LOCAL}(C), \text{VAL}(i)\}$

Performing a read operation at a local copy.
Processing Write Operations

1. \( \text{DEP}(W) := \text{LOCAL}(C) \)

2. \( \text{WORK}(i)[i] := \text{WORK}(i)[i] + 1 \)
   \( \text{ts}(W)[i] := \text{WORK}(i)[i] \)
   \( \text{ts}(W)[j] := \text{DEP}(W)[j] \)

3. \( \text{ts}(W) \)

4. \( \text{LOCAL}(C) := \max\{\text{LOCAL}(C), \text{ts}(W)\} \)

5. \( \text{DEP}(W) \leq \text{VAL}(i) \)

Performing a write operation at a local copy.