Distributed Systems (COMP9243)

Lecture 3a: Replication & Consistency

1. Replication
2. Consistency
   - Models vs Protocols
3. Update propagation
REPLICATION

Make copies of services on multiple machines.

Why?:

➡ Reliability
  ● Redundancy

➡ Performance
  ● Increase processing capacity
  ● Reduce communication

➡ Scalability (prevent centralisation)
  ● Prevent overloading of single server (size scalability)
  ● Avoid communication latencies (geographic scalability)
Data Replication (Server Replication/Mirroring):

- FTP Server
- GNU FTP mirror
- FTP

Data vs Control Replication
Data Replication (Caching):

What’s the difference between mirroring and caching?
Control Replication:

What are the challenges of doing this?

DATA VS CONTROL REPLICATION
We will be looking primarily at data replication (including combined data and control replication).
Replication Issues

Updates
- Consistency (how to deal with updated data)
- Update propagation

Replica placement
- How many replicas?
- Where to put them?

Redirection/Routing
- Which replica should clients use?
DISTRIBUTED DATA STORE

→ data-store stores data items

Client’s Point of View:

Client A
Client B
Client C
Client D
Distributed Data-Store’s Point of View:

Client A  
Client B  
Client C  
Client D  
Replica 1  
Replica 2  
Replica 3  
Replica 4

Data Store
Data Model:
- data item: simple variable
- data item values: explicit (0, 1), abstract (a, b)
- data store: collection of data items

Operations on a Data Store:
- Read. \( R_i(x)b \) Client i performs a read for data item \( x \) and it returns \( b \)
- Write. \( W_i(x)a \) Client i performs write on data item \( x \) setting it to \( a \)
- Operations not instantaneous
  - Time of issue (when request is sent by client)
  - Time of execution (when request is executed at a replica)
  - Time of completion (when reply is received by client)
- Coordination among replicas
Replica Managers:

1. Issue
2. Consistency protocol
3. Execution
4. Completion
Timeline:

- ClientA/Replica1: WA(x)1, WA(x)0
- ClientB/Replica2: RB(x)-, RB(x)1, RB(x)1, RB(x)0

Client A/Replica 1

Client B/Replica 2
CONSISTENCY

Conflicting Data:
- Do replicas have exactly the same data?
- What differences are permitted?

Consistency Dimensions:
- Time and Order

Time:
- How old is the data (staleness)?
- How old is the data allowed to be?
  - Time, Versions

Operation order:
- Were operations performed in the right order?
- What orderings are allowed?

Real world examples of inconsistency?
Updates and concurrency result in conflicting operations

Conflicting Operations:
- Read-write conflict (only 1 write)
- Write-write conflict (multiple concurrent writes)
- The order in which conflicting operations are performed affects consistency

Partial vs Total Ordering:
- partial order: order of a single client’s operations
- total order: interleaving of all conflicting operations
Example:

**Client A:** \[ x = 1; \ x = 0; \]

**Client B:** `print(x);`
`print(x);`

Possible results:

- -, 11, 10, 00

How about 01?

What are the conflicting ops? What are the partial orders? What are the total orders?

```
Client A    W(x) 1    W(x) 0
```

```
Client B    R(x) 0    R(x) 1
```

Can you sanely use a system like this?
CONSISTENCY MODEL

*Defines which interleavings of operations are valid (admissible)*

Consistency Model:

- Concerned with consistency of a data store.
- Specifies characteristics of valid total orderings

A data store that implements a particular model of consistency will provide a total ordering of operations that is valid according to the model.
Data Coherence vs Data Consistency:

**Data Coherence** ordering of operations for single data item
  ➔ e.g. a read of x will return the most recently written value of x

**Data Consistency** ordering of operations for whole data store
  ➔ implies data coherence
  ➔ includes ordering of operations on other data items too

Non-distributed data store:
  ➔ Data coherence is respected
  ➔ Program order is maintained
**DATA-CENTRIC CONSISTENCY MODEL**

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

- Multiple clients accessing the same data store
- Described consistency is experienced by all clients
  - Client A, Client B, Client C see same kinds of orderings
- Non-mobile clients (replica used doesn’t change)
STRONG ORDERING VS WEAK ORDERING

Strong Ordering (tight):
- All writes must be performed in the order that they are invoked
- Example: all replicas must see: $W(x)a \ W(x)b \ W(x)c$
- Strict (Linearisable), Sequential, Causal, FIFO (PRAM)

Weak Ordering (loose):
- Ordering of groups of writes, rather than individual writes
- Series of writes are grouped on a single replica
- Only results of grouped writes propagated.
- Example: $\{W(x)a \ W(x)b \ W(x)c\} \Rightarrow \{W(x)a \ W(x)c\} \Rightarrow \{W(x)c\}$
- Weak, Release, Entry
**Strict Consistency**

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

Absolute time ordering of all shared accesses

What is *most recent* in a distributed system?

- Assumes an absolute global time
- Assumes instant communication (atomic operation)
- Normal on a uniprocessor
- Impossible in a distributed system
LINEARISABLE CONSISTENCY

All operations are performed in a single sequential order

- Operations ordered according to a global (finite) timestamp.
- Program order of each client maintained

linearisable

not linearisable
**Sequential Consistency**

All operations are performed in some sequential order

- More than one correct sequential order possible
- All clients see the *same* order
- Program order of each client maintained
- Not ordered according to time *Why is this good?*

**Performance:**

read time + write time $\geq$ minimal packet transfer time
Potentially causally related writes are executed in the same order everywhere

Causally Related Operations:

- Read followed by a write (in same client)
- \( W(x) \) followed by \( R(x) \) (in same or different clients)

<table>
<thead>
<tr>
<th>Client A</th>
<th>( W(x) ) a</th>
<th>( W(x) ) c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client B</td>
<td>( W(x) ) b</td>
<td>( R(x) ) a</td>
</tr>
<tr>
<td>Client C</td>
<td>( R(x) ) b</td>
<td>( R(x) ) a</td>
</tr>
<tr>
<td>Client D</td>
<td>( R(x) ) b</td>
<td>( R(x) ) a</td>
</tr>
</tbody>
</table>

Causally consistent

<table>
<thead>
<tr>
<th>Client A</th>
<th>( W(x) ) a</th>
<th>( W(x) ) c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client B</td>
<td>( R(x) ) a</td>
<td>( W(x) ) b</td>
</tr>
<tr>
<td>Client C</td>
<td>( R(x) ) a</td>
<td>( R(x) ) b</td>
</tr>
<tr>
<td>Client D</td>
<td>( R(x) ) b</td>
<td>( R(x) ) a</td>
</tr>
</tbody>
</table>

Not causally consistent

How could we make this valid?
FIFO (PRAM) Consistency

Only partial orderings of writes maintained

FIFO consistent

How could we make this valid?
Weak Consistency

Shared data can be counted on to be consistent only after a synchronisation is done

Enforces consistency on a group of operations, rather than single operations

- Synchronisation variable (s)
- Synchronise operation (synchronise(S))
- Define ‘critical section’ with synchronise operations

Properties:

- Order of synchronise operations sequentially consistent
- Synchronise operation cannot be performed until all previous writes have completed everywhere
- Read or Write operations cannot be performed until all previous synchronise operations have completed
Example:

- synchronise(S) \ W(x)a W(y)b W(x)c synchronise(S)
- Writes performed locally
- Updates propagated only upon synchronisation
- Only \ W(y)b \ and \ W(x)c \ have \ to \ be \ propagated

How could we make this valid?
**Release Consistency**

Explicit separation of synchronisation tasks

- **acquire(S)** - bring local state up to date
- **release(S)** - propagate all local updates
- acquire-release pair defines ‘critical region’

**Properties:**

- Order of synchronisation operations are FIFO consistent
- Release cannot be performed until all previous reads and writes done by the client have completed
- Read or Write operations cannot be performed until all previous acquires done by the client have completed
What is an example of an invalid ordering?
Lazy Release Consistency and Entry Consistency

Lazy Release Consistency:

→ Don’t send updates on release
→ Acquire causes client to get newest state

Entry Consistency:

→ Each shared data item has own synchronisation variable
→ Exclusive and non-exclusive access modes
→ acquire() provides ownership of synchronisation variable
→ acquire() synchronises data
→ release() relinquishes exclusive access (but not ownership)
CAP Theory

C: Consistency: Linearisability
A: Availability: Timely response
P: Partition-Tolerance: Functions in the face of a partition

You can only choose two of C A or P
CAP Theory

C: Consistency: Linearisability
A: Availability: Timely response
P: Partition-Tolerance: Functions in the face of a partition

You can only choose two of C A or P
CAP Theory

C: Consistency: Linearisability
A: Availability: Timely response
P: Partition-Tolerance: Functions in the face of a partition

You can only choose two of C A or P
**CAP Theory**

C: Consistency: Linearisability
A: Availability: Timely response
P: Partition-Tolerance: Functions in the face of a partition

You can only choose **two** of C A or P
CAP Impossibility Proof:
CAP Impossibility Proof:
CAP Impossibility Proof:

Diagram:

- Replica A
- Replica B
- Client

Write connection between Replica A and Replica B, indicating a possible failure or partition scenario.
CAP Impossibility Proof:

Replica A \rightarrow \text{Read} \rightarrow \text{Client} \leftarrow \text{Write} \rightarrow \text{Replica B}

No Consistency
CAP Impossibility Proof:

Write (does not return)

No Availability
CAP Impossibility Proof:

Replica A

Replica B

Write (fails)

No Partition Tolerance

Client
CAP Consequences

For wide-area systems:

➜ Must choose: Consistency or Availability

➜ Choosing Availability
  • Give up on consistency?
  • Eventual consistency

➜ Choosing Consistency
  • No availability
  • delayed (and potentially failing) operations

Why can’t we choose C and A and forget about P?
**Eventual Consistency**

*If no updates take place for a long time, all replicas will gradually become consistent*

![Diagram showing eventual consistency with Client A, Client B, and Client C showing different operations: W(x)a, R(x)a, W(y)b, W(z)c, R(x)Nil, R(y)b, R(z)c, R(y)Nil, R(z)c, R(y)b.]

**Requirements:**
- Few read-write conflicts (R \(\gg\) W)
- Few write-write conflicts
- Clients accept time inconsistency (i.e., old data)
- What about ordering?
**CLIENT-CENTRIC CONSISTENCY MODELS**

*Provides guarantees about ordering of operations for a single client*

- Single client accessing data store
- Client accesses different replicas (modified data store model)
- Data isn’t shared by clients
- Client A, Client B, Client C may see different kinds of orderings

In other words:

- The effect of an operation depends on the client performing it
- Effect also depends on the history of operations that client has performed.
Data-Store Model for Client-Centric Consistency:

- Data-items have an owner
- No write-write conflicts
Notation and Timeline for Client-Centric Consistency:

→ \( x_i[t] \): version of \( x \) at replica \( i \) at time \( t \)
→ Write Set: \( WS(x_i[t]) \): set of writes at replica \( i \) that led to \( x_i(t) \)
→ \( WS(x_i[t1];x_j[t2]) \): \( WS(x_j(t2)) \) contains same operations as \( WS(x_i(t1)) \)
→ \( WS(!x_i[t1];x_j[t2]) \): \( WS(x_j(t2)) \) does not contain the same operations as \( WS(x_i(t1)) \)
→ \( R(x_i[t]) \): a read of \( x \) returns \( x_i(t) \)

Replica 1
- \( W(x_1) \)
- \( WS(x_1) \)
- \( R(x_1) \)

Replica 2
- \( W(x_1) \)
- \( WS(x_1) \)
- \( W(x_2) \)
- \( WS(x_1;x_2) \)
- \( R(x_2) \)
MONOTONIC READS

If a client has seen a value of $x$ at a time $t$, it will never see an older version of $x$ at a later time.

When is Monotonic Reads sufficient?
**MONOTONIC WRITES**

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

All writes by a single client are sequentially ordered.

Replica 1  
$W(x_1)$  
Replica 2  
$W(x_1)$  
$WS(x_1)$  
$W(x_2)$  

monotonic-write consistent

Replica 1  
$W(x_1)$  
Replica 2  
$WS(!x_1;x_0)$  
$W(x_2)$  

not monotonic-write consistent

How is this different from FIFO consistency?

- Only applies to write operations of single client.
- Writes from clients not requiring monotonic writes may appear in different orders.
**Read Your Writes**

The effect of a write on $x$ will always be seen by a successive read of $x$ by the same client

When is Read Your Writes sufficient?
**WRITE FOLLOWS READS**

A write operation on x will be performed on a copy of x that is up to date with the value most recently read by the same client.

When is Write Follows Reads sufficient?

```plaintext
<table>
<thead>
<tr>
<th>Replica 1</th>
<th>Replica 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(x1)</td>
<td>WS(x1; x2) W(x3)</td>
</tr>
<tr>
<td>R(x1)</td>
<td>W(x3)</td>
</tr>
</tbody>
</table>

writes−follow−reads consistent

```
CONSISTENCY PROTOCOLS

Consistency Protocol: implementation of a consistency model

Primary-Based Protocols:

► Remote-write protocols
► Local-write protocols

Replicated-Write Protocols:

► Active Replication
► Quorum-Based Protocols
REMOTE-WRITE PROTOCOLS

Single Server:

- All writes and reads executed at single server
- No replication of data

![Diagram of Remote-Write Protocols]

- W1. Write request
- W2. Forward request to server for x
- W3. Acknowledge write completed
- W4. Acknowledge write completed

- R1. Read request
- R2. Forward request to server for x
- R3. Return response
- R4. Return response
Primary-Backup:

- All writes executed at single server, Reads are local
- Updates block until executed on all backups

Performance
LOCAL-WRITE PROTOCOLS

Migration:

→ Data item migrated to local server on access
✓ Performance (when not sharing data)

1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server
Migrating Primary (multiple reader/single writer):

- Performance for concurrent reads
- Performance for concurrent writes

Diagram:

- Client
- Data store
- Backup server
- Old primary for item x
- New primary for item x
- 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
ACTIVE REPLICATION

- Updates (write operation) sent to all replicas
- Need totally-ordered multicast (for sequential consistency)
- e.g. sequencer/coordinator to add sequence numbers

![Diagram showing active replication]
**Quorum-Based Protocols**

- Voting
- Versioned data
- Read Quorum: $N_r$
- Write Quorum: $N_w$
- $N_r + N_w > N$ *Why?*
- $N_w > N/2$ *Why?*

---

**Diagram:**

- **Read quorum**
  - (a) $N_r = 3, N_w = 10$
  - (b) $N_r = 7, N_w = 6$
  - (c) $N_r = 1, N_w = 12$

---
**Push vs Pull**

Pull:
- Updates propagated only on request
- Also called *client-based*
- R/W low
- Polling delay

Push:
- Push updates to replicas
- Also called *server-based*
- When low staleness required
- R » W
  - Have to keep track of all replicas
Push Update Propagation:

What to propagate?

- Data
  - R/W high
- Update operation
  - low bandwidth costs
- Notification/Invalidation
  - R/W low
Compromise: Leases:

Server promises to push updates until lease expires

Lease length depends on:

- **age**: Last time item was modified
- **renewal-frequency**: How often replica needs to be updated
- **state-space overhead**: lower expiration time to reduce bookkeeping when many clients
REPLICA PLACEMENT

Permanent replicas
Server-initiated replicas
Client-initiated replicas
Clients

→ Server-initiated replication
→ Client-initiated replication
**Dynamic Replication**

Situation changes over time

- Number of users, Amount of data
- Flash crowds
- R/W ratio

**Dynamic Replica Placement:**

- Network of replica servers
- Keep track of data item requests at each replica
- Thresholds:
  - Deletion threshold
  - Replication threshold
  - Migration threshold
- Clients always send requests to nearest server
End-to-End argument:
- Where to implement replication mechanisms?
- Application? Middleware? OS?

Policy vs Mechanism:
- Consistency models built into middleware?
- One-size-fits-all?

Determining Policy:
- Who determines the consistency model used?
  - Application, Middleware
  - Client, Server

Keep It Simple, Stupid:
- Will the programmer understand the consistency model?
**Reading List**

**Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services** An overview of the CAP theorem and its proof.

**Eventual Consistency** An overview of eventual consistency and client-centric consistency models.
Consistency Models:

- Research consistency models used in existing Distributed Systems
- Why are those models being used?
- In the systems you looked at, could other models have been used? Would that have made the system better?

Hacker’s Edition:

- Find a system that provides Eventual Consistency
  - (alternatively, implement (possibly in Erlang) a system that provides Eventual Consistency)
- Replicate some data and perform queries. How often do you get inconsistent results?
- If you can tweak replication parameters, how do they affect the consistency of results?