Distributed Systems (COMP9243)
Lecture 5 (B): Replication & Consistency

Slide 1

A. Replication
B. Consistency
  • Models vs Protocols
  • Update propagation

Distributed Data Store

→ data-store stores data items

Client’s Point of View:

Slide 2

Slide 3

Distributed Data-Store’s Point of View:

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: Ri(x)b Client i performs a read for data item x and it returns b
  → Write: Wi(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

Slide 4
**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-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 (Linearizable), 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: \( \{W(x)a \} \) \( \{W(x)b \} \) \( \{W(x)c \} \) == \( \{W(x)a \} \) \( W(x)c \) == \( W(x)c \)
- 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

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?

Causal Consistency

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)

Performance:
read time + write time >= minimal packet transfer time
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:
- Don’t send updates on release
- Acquire causes client to get newest state
- Added efficiency if acquire-release performed by same client (e.g., in a loop)

Slide 18

**ENTRY CONSISTENCY**

Synchronisation variable associated with specific shared data item (guarded data item)
- Each shared data item has own synchronisation variable
- acquire()
  - Provides ownership of synchronisation variable
  - Exclusive and nonexclusive access modes
  - Synchronises data
  - Requires communication with current owner
- release()
  - Relinquishes exclusive access (but not ownership)

Slide 19

Properties:
- Acquire does not complete until all guarded data is brought up to date locally
- If a client has exclusive access to a synchronisation variable, no other client can have any kind of access to it
- When acquiring nonexclusive access, a client must first get the updated values from the synchronisation variable’s current owner

Slide 20

**CAP THEORY**

You can only choose two of C A or P

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

Slide 21
You can only choose two of C A or P

Slide 22
You can only choose two of C A or P

Slide 23
You can only choose two of C A or P

Slide 24
CAP Impossibility Proof:
CAP Impossibility Proof:

- Write (fails) 
- No Partition Tolerance

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

Requirements:
- Few read-write conflicts (R > W)
- Few write-write conflicts
- Clients accept time inconsistency (i.e., old data)
- What about ordering?

Examples:
- DNS:
  - No write-write conflicts
  - Updates slowly (1-2 days) propagate to all caches
- WWW:
  - Few write-write conflicts
  - Mirrors eventually updated
  - Cached copies (browser or proxy) eventually replaced
  - Manual merging for write-write conflicts

EVENTUAL CONSISTENCY

- Client A
  - W(x)
  - R(x)
  - W(y)
  - M(x)
- Client B
  - R(x) Nil
  - W(y) Nil
  - R(z) M
- Client C
  - R(x) M
  - R(y) Nil
  - R(z) M
  - R(y) M

 eventual consistent
**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

**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*

- 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 $x_i[t]$

**Monotonic Writes**

*When is Monotonic Reads sufficient?*

<table>
<thead>
<tr>
<th>Replica 1</th>
<th>Replica 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$WS(x_1)$</td>
<td>$WS(x_1)$</td>
</tr>
<tr>
<td>$R(x_1)$</td>
<td>$R(x_2)$</td>
</tr>
</tbody>
</table>

- Monotonic−read consistent
- Not monotonic−read consistent
**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.

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.

**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?

**Choosing the Right Model**

Trade-offs

**Consistency and Redundancy:**
- All copies must be strongly consistent
- All copies must contain full state
- Reduced consistency → reduced reliability

**Consistency and Performance:**
- Consistency requires extra work and communication
  - Can result in loss of overall performance
  - Weaker consistency possible

**Consistency and Scalability:**
- Implementation of consistency must be scalable
  - don’t take a centralised approach
  - avoid too much extra communication
**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

**Local-Write Protocols**

Migration:
- Data item migrated to local server on access
- Performance (when not sharing data)
Migrating Primary (multiple reader/single writer):
✓ Performance for concurrent reads
✗ Performance for concurrent writes

Slide 45

Client
Old primary
for item x
New primary
for item x
Backup server

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

Data store

Slide 46

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

Slide 47

QUORUM-BASED PROTOCOLS
→ Voting
→ Versioned data
→ Read Quorum: Nr
→ Write Quorum: Nw
→ Nr + Nw > N Why?
→ Nw > N/2 Why?

Slide 48

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

Permanent Replicas:
- Initial set of replicas
- Created and maintained by data-store owner(s)
- Allow writes

Server-Initiated Replicas:
- Enhance performance
- Not maintained by owner
- Placed close to groups of clients
  - Manually
  - Dynamically

Client-Initiated Replicas:
- Client caches
- Temporary
- Owner not aware of replica
- Placed close to client
- Maintained by host (often client)
DYNAMIC REPLICA

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

MISCELLANEOUS IMPLEMENTATION AND DESIGN ISSUES

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


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

HOMEWORK

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?