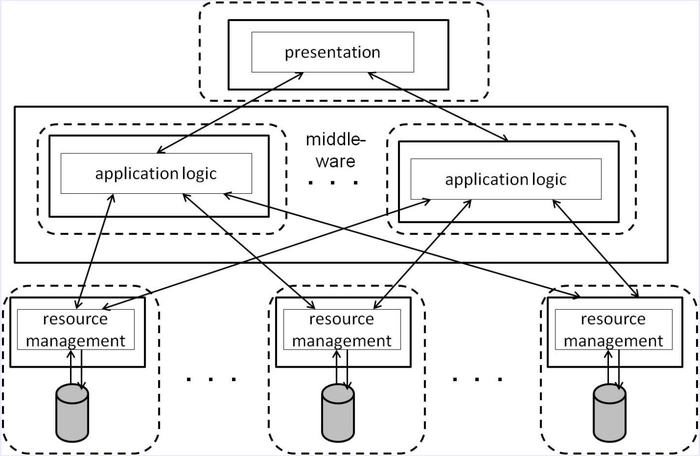
11. Replication and (Weaker) Consistency

Motivation

- Reliable and high-performance computation on a single instance of a data object is prone to failure.
- Replicate data to overcome single points of failure and performance bottlenecks.

Problem: Accessing replicas uncoordinatedly can lead to different values for each replica, jeopardizing consistency.

Basic architectural model

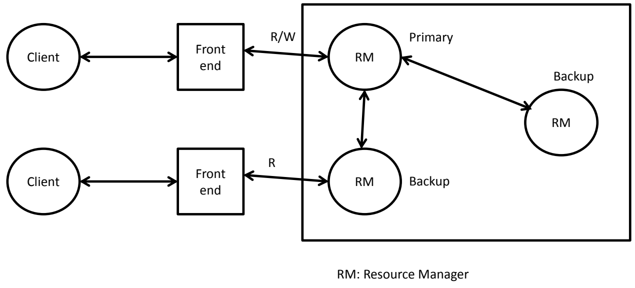


Classification of replication approaches

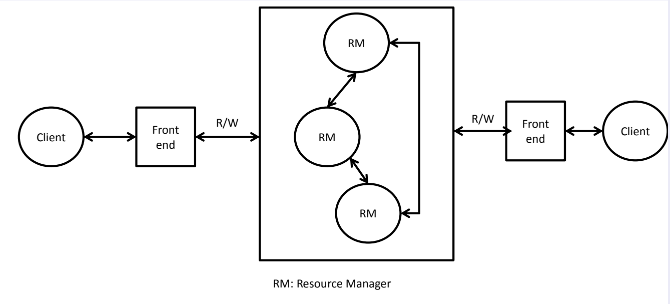
Two orthogonal dimensions

- Location of change:
  - Primary Copy: updates on a data item can only be performed on a single, dedicated replica
  - Write Anywhere: updates can be performed on any replica
- Propagation Speed
  - Immediate/Eager: At commit, all replicas contain the change
  - Delayed: only the modified replica contains the change at commit, the others will receive the changes later

Primary Copy replication model



Update anywhere replication model



Tradeoffs of application approaches

Overall Tradeoffs

- Location of change:
  - Primary Copy: Simple synchronization
  - Write Anywhere: flexible, no single bottleneck
- Propagation Speed
  - Immediate/Eager: strongly consistent, potentially long response times
  - Delayed/Lazy: fast response time, consistency problems

Method-Specific Tradeoffs

- Primary/Eager: resource contention on querying/updating/replication; strong consistency with simple implementation (e.g., with 2PC+local 2PL)
- Write anywhere/Eager: potentially prone to distributed deadlocks
- Primary/Lazy: typically fast (if not on multiple sites), outdated data
- Write anywhere/Lazy: fast, serializability not guaranteed

Tradeoffs of application approaches

Overall Tradeoffs

- Location of change:
  - Primary Copy: Simple synchronization
  - Write Anywhere: flexible, no single bottleneck
- Propagation Speed
  - Immediate/Eager: strongly consistent, potentially long response times
  - Delayed/Lazy: fast response time, consistency problems

Method-Specific Tradeoffs

- Primary/Eager: resource contention on querying/updating/replication; strong consistency with simple implementation (e.g., with 2PC+local 2PL)
- Write anywhere/Eager: potentially prone to distributed deadlocks
- Primary/Lazy: typically fast (if not on multiple sites), outdated data
- Write anywhere/Lazy: fast, serializability not guaranteed

Synchronous replication protocols (basic)

ROWA

- Write the change to all replicas
- Read on (any) single replica
- Expensive write coordination (2PC)
- Cheap, highly available reads
- Low write availability (lower than without replication)

Primary Copy

- Write the change initially to single replica
- Propagate changes in bulk to other replicas
- Coordination with read locks: request from primary
- Reduce write cost
- Increased read cost

Synchronous replication protocols (basic)

ROWA

- Write the change to all replicas
- Read on (any) single replica
- Expensive write coordination (2PC)
- Cheap, highly available reads
- Low write availability (lower than without replication)

Primary Copy

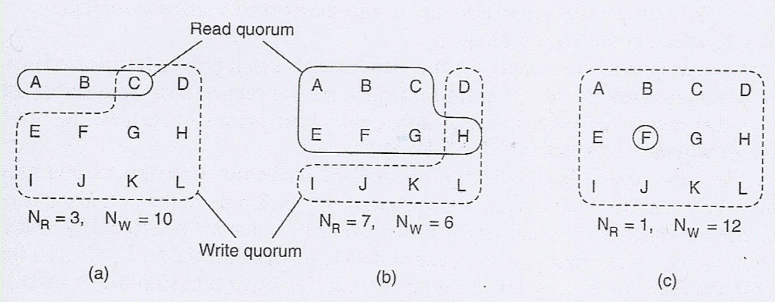
- Write the change initially to single replica
- Propagate changes in bulk to other replicas
- Coordination with read locks: request from primary
- Reduce write cost
- Increased read cost

Quorum-Based Protocols

- Idea: Clients have to request and acquire the permission of multiple servers before either reading or writing a replicated data item.
- Assume an object has N replicas.
  - For update, a client must first contact at least $\frac{N}{2} + 1$ servers and get them to agree to do the update. Once they have agreed, all contacted servers process the update assigning a new version number to the updated object.
  - For read, a client must first contact at least $\frac{N}{2} + 1$ servers and ask them to send the version number of their local version. The client will then read the replica with the highest version number.
- This approach can be generalized to an arbitrary read quorum N_R and write quorum N_W such that holds:
  - $N_R + N_W > N$,
  - $N_W > \frac{N}{2}$.

  This approach is called *quorum consensus* method.

Example



(a) Correct choice of read and write quorum.

(b) Choice running into possible inconsistencies.

(c) ROWA by quorum

CAP Theorem

From the three desirable properties of a distributed shared-data system:

- atomic data consistency (i.e. operations on a data item look as if they were completed at a single instant),
- system availability (i.e. every request received by a non-failing node must result in a response), and
- tolerance to network partition (i.e. the system is allowed to lose messages),

only two can be achieved at the same time at any given time.

$\Longrightarrow$ Given that in distributed large-scale systems network partitions cannot be avoided, consistency and availability cannot be achieved at the same time.

Two basic options:

- Distributed ACID-transactions:

  Consistency has priority, i.e. updating replicas is part of the transaction - thus availability is not guaranteed.

- Large-scale distributed systems:

  Availability has priority - thus a weaker form of consistency is accepted, accpeting access to outdated replicas

  $\Longrightarrow$ Inconsistent updates may happen and have to be resolved on the application level, in general.

Eventual Consistency

- Specific form of weak consistency
- Guarantees
  - if no new updates are made to the object
  - eventually all accesses will return the last updated value
- Probabilistic inconsistency window duration, impacted
  - failures occur,
  - communication delays
  - the load on the system,
  - the number of replicas involved
- Originally popular in large-scale, no-DB systems (DNS)
- Major factor the NoSQL movement

Is this the end of the consistency story?

- Serializability and Eventual Consistency are (almost) at the extreme end of the spectrum
- Is there anything in between that would provide practically useful combinations of consistency and availability?
- In fact, there is wide of consistency models proposed in various communities
  - Database transaction models
  - Distributed systems single object models
- The CAP theorem does not talk about serializability, but linearizability
- Let's survey the space
- There is recent work that structures the space and makes proofs around the availability classes

Overview on Consistency

- We have a system with state and operations on the state
- Operations form a history
- Consistency models determine which histories are permissible
- Simplest model: cpu register
    - Instant application
    - strict order
- Challenges
    - Concurrent histories
    - Propagation delay

Database Consistency: Anomalies (1)

Dirty Writes

$w_1X \ldots w_2X \ldots (c_1 \text{ or } a_1)$

Dirty Read

$w_1X \ldots r_2X \ldots (c_1 \text{ or } a_1)$

Lost Update

$r_1X \ldots w_2X \ldots w_1X (c_1)$

Database Consistency: Anomalies (2)

Fuzzy Read

$r_1 X \ldots w_2 X \ldots r_1 X (c_1 \text{ or } a_1)$

Phantom

$r_1[P] \ldots w_2[yinP] \ldots r_1 X (c_1 \text{ or } a_1)$

Write Skew

$r_1 X \ldots r_2 Y \ldots w_1 Y \ldots w_2 X \ldots c_1 c_2$

Database Consistency Classes

ANSI SQL classes

Prevent typical anomalies from happening

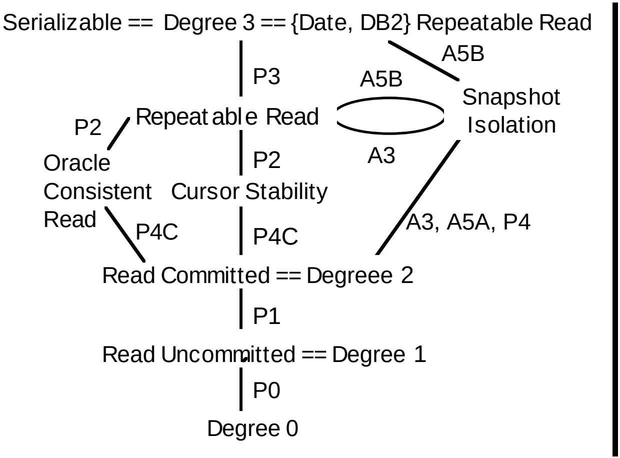
- Read Uncomitted:
- Read Committed:
- Repeatable Read:
- Serializable:

Modelled around typical locking strategies

Other classes

- Cursor Stability:
- Snapshot Isolation:
  - Perform all reads and writes on a snapshot created at t_s
  - At commit, check if any change by other TA on modified objects since t_s

Database Consistency: Classification



DS Consistency Classes

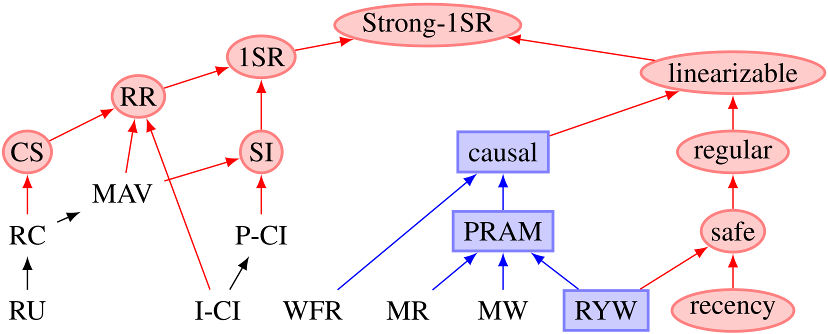
Session Guarantees

- Monotonic Reads: never return previous values
- Monotonic writes: writes in session appear in order
- Writes Follow Reads: happens-before on transactions

Sticky Session Guarantees

- Read Your Writes: get your updated value (or later)
- PRAM: serial execution within session (like RAM)
- Causal consistency/PL-2L: PRAM+WFR

Overall Consistency Classification



Which of them are (un-)available and why?

Causes for unavailability

Preventing Lost Updates

Dectecting competing writes needs coordination

Preventing Write Skew

Generalization of Lost Updates

Recency Guarantess

Network splits may delay process arbitrarily long

Causes for unavailability

Preventing Lost Updates

Dectecting competing writes needs coordination

Preventing Write Skew

Generalization of Lost Updates

Recency Guarantess

Network splits may delay process arbitrarily long

Causes for unavailability

Preventing Lost Updates

Dectecting competing writes needs coordination

Preventing Write Skew

Generalization of Lost Updates

Recency Guarantess

Network splits may delay process arbitrarily long