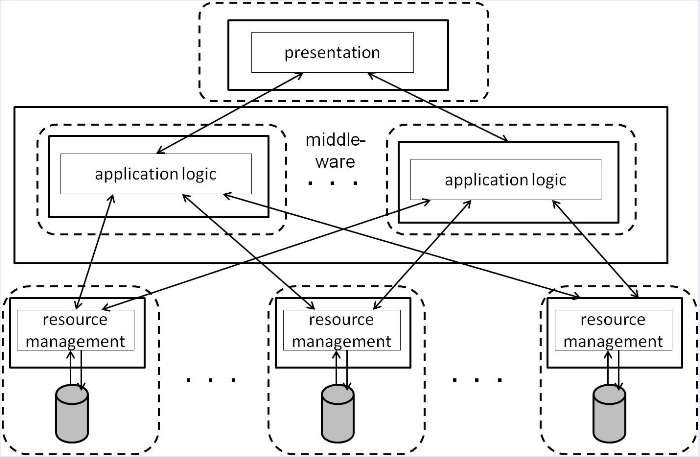
10. Replication

Motivation

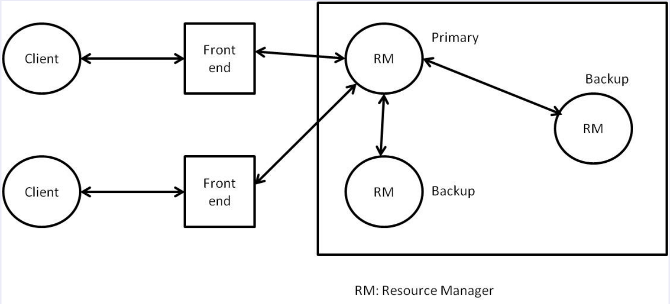
- Reliable and high-performance computation on a single instance of a data object is prone to failure.
- Replicate data to overcome single points of failure and performance bottlenecks.

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

Basic architectural model

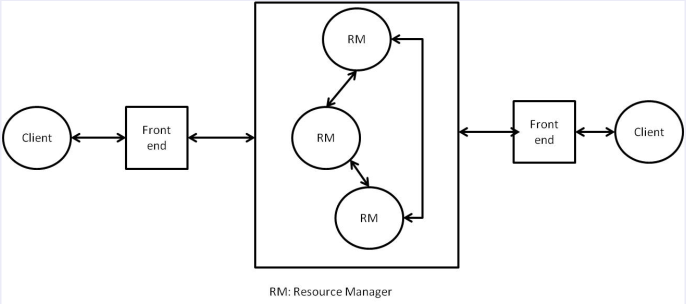


Passive (primary-backup) replication model



RM: Resource Manager

Active replication model



RM: Resource Manager

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.

the two 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: *eventually consistent*.

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

Eventually Consistent - Revisited. Werner Vogels (CTO at Amazon):[1]

- *Strong consistency*

  After the update completes, any subsequent access will return the updated value.

- *Weak consistency*

  The system does not guarantee that subsequent accesses will return the updated value. A number of conditions need to be met before the value will be returned. The period between the update and the moment when it is guaranteed that any observer will always see the updated value is dubbed the inconsistency window.

- *Eventual consistency*

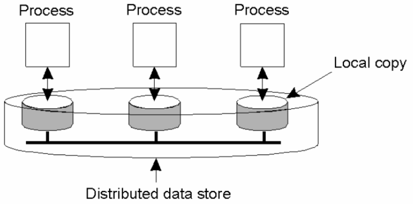
  This is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme. The most popular system that implements eventual consistency is DNS (Domain Name System). Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update.

[1] http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

10.1: Systemwide Consistency

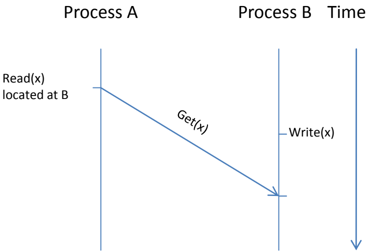
Systemwide consistent view on a data store.

- Processes read and write data in a data store.
  - Each process has a local (or near-by) copy of each object,
  - Write operations are propagated to all replicas.
- Even if rocesses are not considered to be transactions, we would expect, that read operations will always return the value of the last write – however what does "last" mean in the absense of a global clock?



The difficulty of strict consistency

- Any read on a data item returns the value of the most recent write on it.
- This is the expected model of a uniprocessor system.
- In a distributed system there does not exist a global clock!



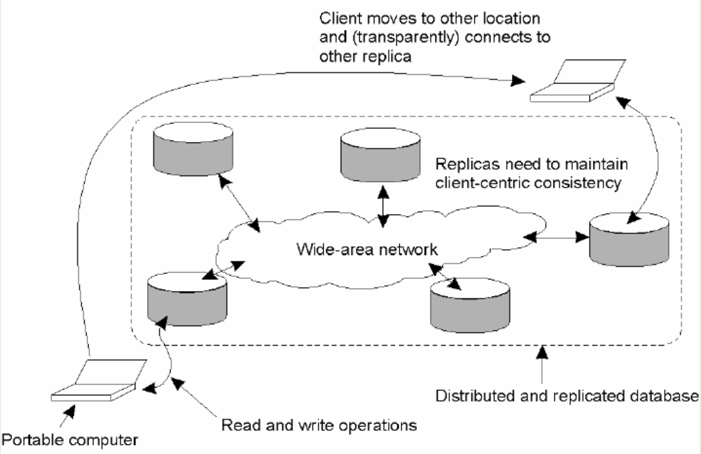
Which value shall be returned?
Old or new one?

10.2: Client-side consistency

Consistent view on a data store shall be guaranteed for clients, not necessarily for the whole system.

- Goal: *eventual consistency*.
  - In the absence of updates, all replicas *converge* towards identical copies of each other.
  - However, it should be guaranteed, that if a client has access to different replica, it sees consistent data.

Example: Client works with two different replica.


Client moves to other location
and (transparently) connects to
other replica
Replicas need to maintain
client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

10.3 Server-side Consistency

Problem

- We would like to achieve consistency between the different replicas of one object.
- This is an issue for active replication.
- It is further complicated by the possibility of network partitioning.

Active Replication

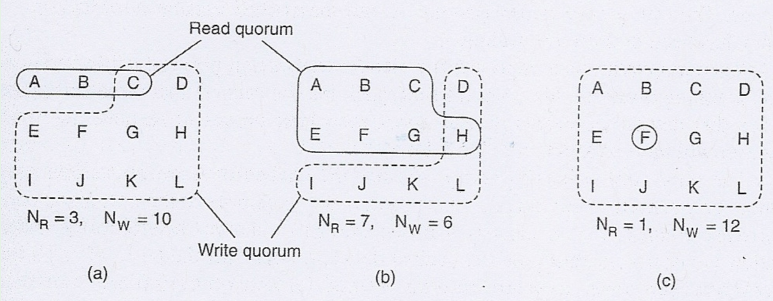
- Update operations are propagated to each replica.
- It has to be guaranteed, that different updates have to be processed in the same order for each replica.
- This can be achieved by totally-ordered multicast or by establishing a central coordinator called *sequencer*, which assigns unique sequence numbers which define the order in which updates have to be carried out.
- These approaches do not scale well in large distributed systems.

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) Correct choice known as ROWA (read one, write all).

11. Real-World Considerations

Obervations

- CAP theorem puts a natural limit on classical distributed transactions
- Maintaining consistency very expensive due to large number of messages (high latency)
- Web-facing datamanagement poses new challenges:
  - Large scalability (towards billions of users)
  - High availability (24x7 operations, global use)
  - Low response times
  - Write-intensive operations (shopping baskets, social media)

Approaches

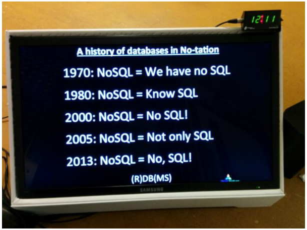
- NoSQL systems (2005 - now): simple data model, limited operations, reduced consistency
- NewSQL systems (2010 - now): SQL+ACID+scalability

$\Longrightarrow$ very active area in research and product development

11.1: NoSQL

Overview

- Catch-all phrase for basically all non-relational and/or non-ACID systems
- Very promiment subclass: Distributed Key/Value-Stores
- automatic partitioning and replication
- relaxed and tuneable consistency: trading off availability and consistency
- Examples: Apache Cassandra, MongoDB, ...



A history of databases in No-tation

1970: NoSQL = We have no SQL

1980: NoSQL = Know SQL

2000: NoSQL = No SQL!

2005: NoSQL = Not only SQL

2013: NoSQL = No, SQL!

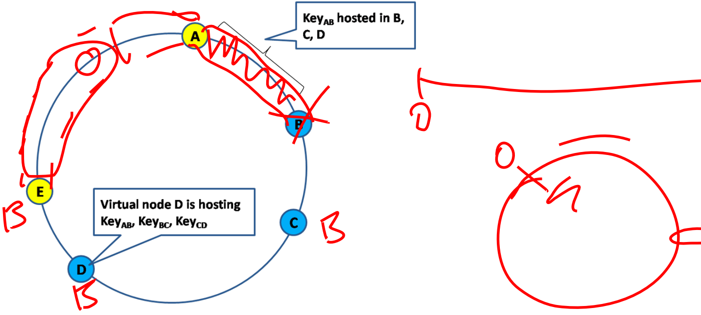
(R)DB(MS)

Cassandra: Background and Concepts

- Based on Amazon Dynamo/Google BigTable ideas
- Open-source software
- Key-value store distributed across nodes by key
  - Not a relational table with many column, many access possibilities
  - Instead a key-¿value mapping like in a hash table
- A value can have a complex structure as it is inside the node - in Cassandra it is columns and super columns

Partitioning and Replication

- Consistent Hashing: Hash Keys and node IDs mapped to (same) circled space
- Each node covers a segment of the rings
- Easy additions/removal and balancing
- content replicated on N subsequent nodes of the ring



Key$_{AB}$ hosted in B, C, D

Virtual node D is hosting Key$_{AB}$, Key$_{BC}$, Key$_{CD}$

Consistency Levels

- CAP theorem allows 2 out of 3
  - (C)onsistency
  - (A)vailability
  - (P)artition tolerance
- Options
  - CA: corruption possible
  - CP: not available if any nodes are down/blocked
  - AP: always available but clients may not always read most recent updates
- Most systems provide CP or AP (why not CA?)
- Cassandra prefers AP but makes "C versus A" configurable by allowing the user to specify a consistency level for each operation
- Consistency levels are handled by setting the quorum for read and write operations

Dealing with eventual consistency

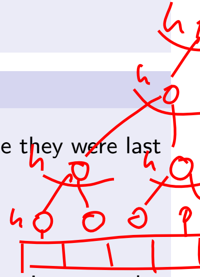
- When W < N (not all replicas are updated) the update is propagated in background
- Version resolution:
  - Each value in a database has a timestamp =¿>key, value, timestemp
  - The timestamp is the timestamp of the latest update of the value (the client must provide a timestamp with each update)
  - When an update is propagated, the latest timestamp wins
- There are two mechanisms to propagate updates:
  - Read repair: hot keys
  - Anti-Entropy: cold keys

Read Repair

- Perform read on multiple replicas.
- Perform reconciliation (e.g. pick the most recent)
- Update all read replicas to the chosen version

Anti-Entropy

- AE is used to repair cold keys - keys that have not been read, since they were last written
- AE works as follows:
  - It generates Merkle Trees for tables periodically
  - These trees are then exchanged with remote nodes using a gossip protocol
  - When ranges in the trees disagree, the corresponding data are transferred between replicas to repair those ranges



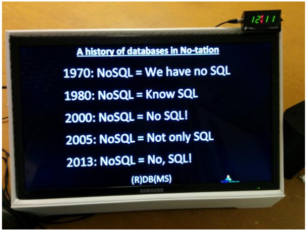
N.B. Merkle Tree (or hash trees) are a compact representation of data for comparison:

- A Merkle tree is a hash tree where leaves are hashes of individual values.
- Parent nodes higher in the tree are hashes of their respective children.

11.2: NewSQL

Overview

- Absence of transactional guarantees cumbersome for developers
- Focus on getting as many SQL/ACID properities while providing (close to) NoSQL scalability
- Main directions:
  1. speeding up/scaling up closely-coupled systems
  2. "Tweaking" consistency models and coordination
  3. clever implementations



A history of databases in No-tation

1970: NoSQL = We have no SQL

1980: NoSQL = Know SQL

2000: NoSQL = No SQL!

2005: NoSQL = Not only SQL

2013: NoSQL = No, SQL!

(R)DB(MS)

H-Store/Volt-DB: Concepts

- Aimed at transaction-heavy workloads, providing ACID and high scalability
- Breaks several assumptions how do DB design with updates
- Aimed at small-scale clusteres (no latency penalty), but providing very high speeds

Design Considerations

- Main-Memory Databases, since most transactional workloads need less than 1 TB
- Single-Thread per core execution model
  - No delays from disk I/O
  - No long-running transactions allowed: typical update queries take few milliseconds!
  - reduces synchronisation cost
- Availability via replication, not log shipping

Implementation

- Shared-nothing architecture over a cluster
- Further shared-nothing decomposition among CPU cores
  - dedicated data structures: tables, indexes
  - transactions run sequentially/serial on a core
- Transactions are known in advance
  - Expressed as stored procedures
  - Information on data access drives scheduling and replication
- Transactions are executed as much as possible on a single site

$\Longrightarrow$ very high transaction rates: several 100K transactions per second per node

Google Spanner/F1

- Massively distributed database, aimed at million servers over the whole world
- Provide synchronous replication
- ACID-style transactional semantics

Replication

- Replicas coordinated with Paxos
- Application specify
  - Datacenters
  - Distance from application (read latency)
  - Distances among replicas (write latency)
  - Number of replicas (durability, availability, read performance)¡

Synchronisation via Time

- Global time hard to achieve!
- Accept bounded uncertainty
  - Establish via GPS and atomic clocks
  - use interval time to define now, before, after
- Use Timestamp-based synchronisation and two-phase commit

Performance

- High commit latency incurred by 2PC over distributed replicas
- Latency hiding via
  - Hierarchical schema: provide partitoning/placement info at schema level (US data at US, European data in the EU)
  - Efficient transfer: protocol buffers
  - Batch reading and writing