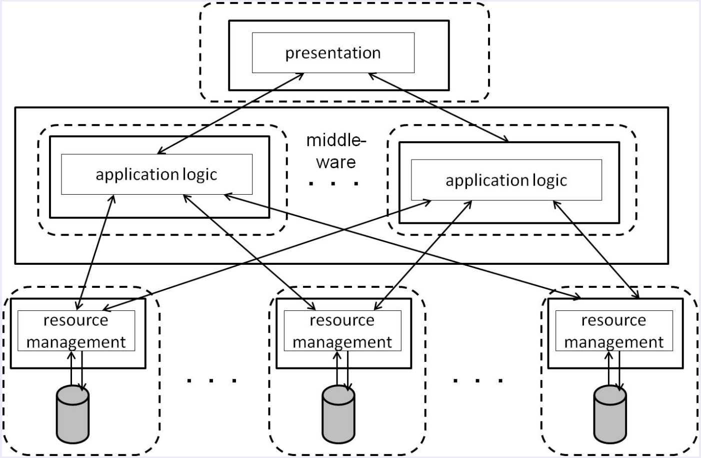
11. Replication

Motivation

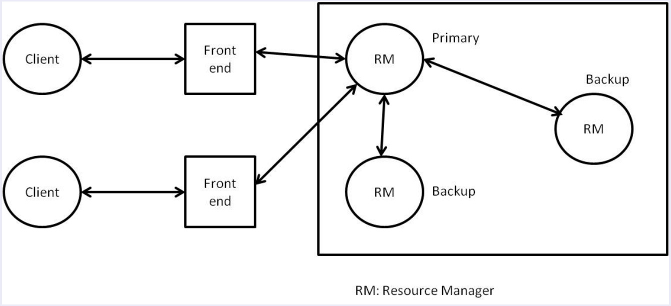
- Reliable and high-performance computation on a single instance of a data object is prone to failure.
- Replicate data to overcome single points of failure and performance bottlenecks.

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

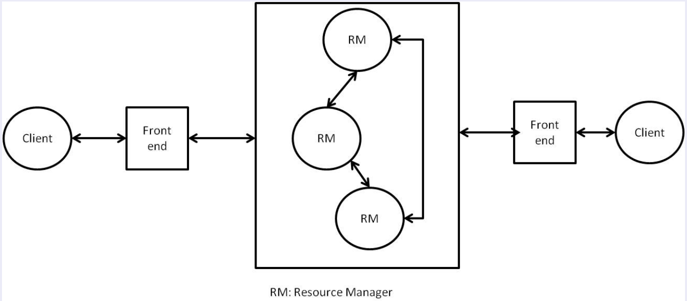
Basic architectural model



Passive (primary-backup) replication model



Active replication model



CAP-Theorm

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*

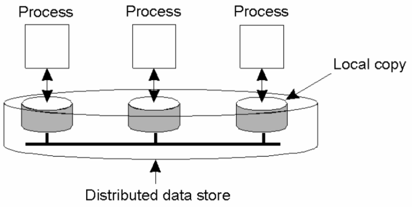
  This is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication 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

11.1: Systemwide Consistency

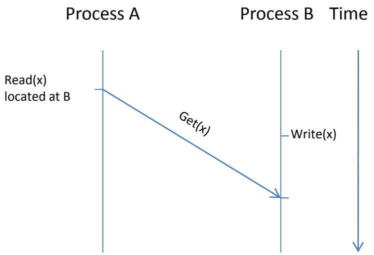
Systemwide consistent view on a data store.

- 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!



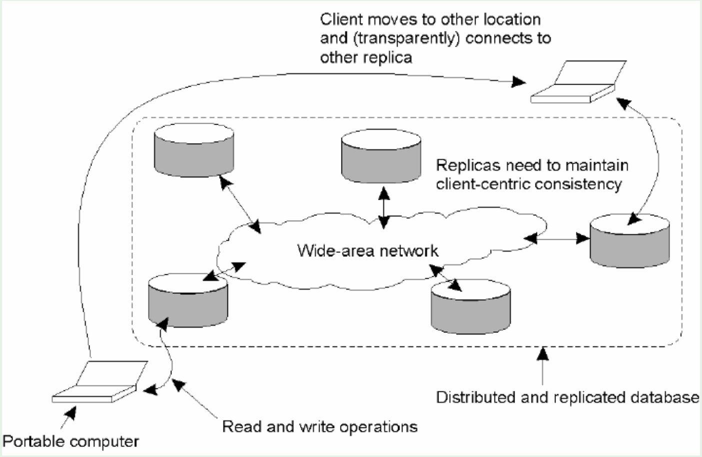
Which value shall be returned?
Old or new one?

11.2: 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

11.3 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

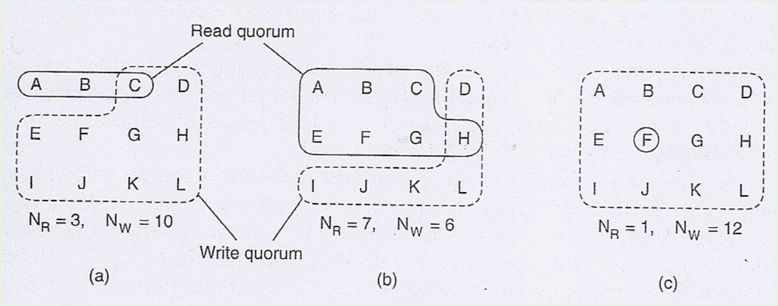
- Update operations are propagated to each replica.
- It has to be guaranteed, that different updates have to be processed in the same order for each replica.
- This can be achieved by totally-ordered multicast or by establishing a central coordinator called *sequencer*, which assigns unique sequence 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).