

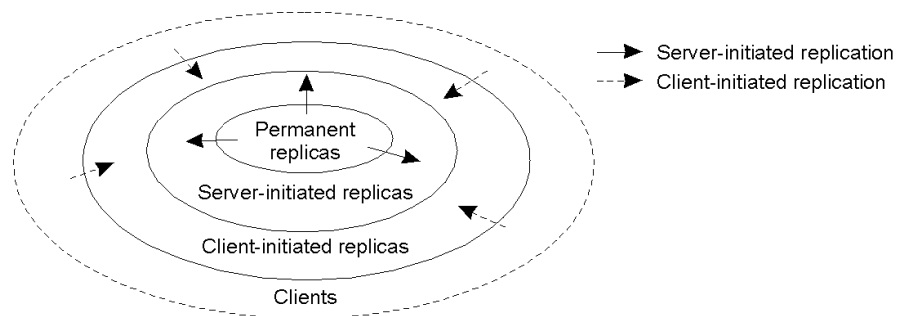
Replica Placement

Model: We consider objects (and don't worry whether they contain just data or code, or both)

Distinguish different processes: A process is capable of hosting a replica of an object or data:

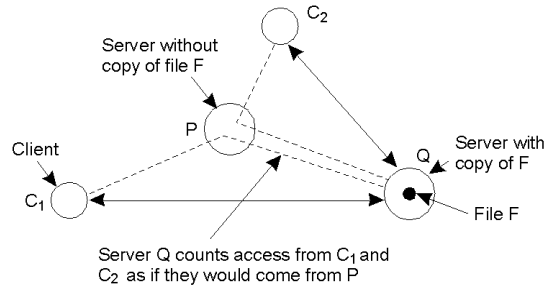
- **Permanent replicas:** Process/machine always having a replica
- **Server-initiated replica:** Process that can dynamically host a replica on request of another server in the data store
- **Client-initiated replica:** Process that can dynamically host a replica on request of a client (client cache)

Replica Placement



- The logical organization of different kinds of copies of a data store into three concentric rings.
- Examples: web servers file servers multicast trees

Server-Initiated Replicas



- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold $D \Rightarrow$ drop file
- Number of accesses exceeds threshold $R \Rightarrow$ replicate file
- Number of access between D and $R \Rightarrow$ migrate file

Client-Initiated Replicas

- More like a client cache
 - Keep it on disk?
 - Keep it in memory?
 - How much space to use?
 - How long to keep copy/replica?
 - How to detect data is stale?
- Read-only files work best
- Sharing data among client processes may be good. Sharing space is essential

Update Propagation (1/3)

- Propagate only notification/invalidation of update (often used for caches)
- Transfer data from one copy to another (distributed databases)
- Propagate the update *operation* to other copies (also called active replication)

Observation: No single approach is the best, but depends highly on available bandwidth and read-to-write ratio at replicas.

Update Propagation (2/3)

- Pushing updates: server-initiated approach, in which update is propagated regardless whether target asked for it.
- Pulling updates: client-initiated approach, in which client requests to be updated.

Issue	Push-based	Pull-based
1:	List of client caches	None
2:	Update (and possibly fetch update)	Poll and update
3:	Immediate (or fetch-update time)	Fetch-update time

1: *State at server*
2: *Messages to be exchanged*
3: *Response time at the client*

Update Propagation (3/3)

Observation: We can dynamically switch between pulling and pushing using **leases**: A contract in which the server promises to push updates to the client until the lease expires.

Issue: Make lease expiration time dependent on system's behavior (adaptive leases):

- Age-based leases: An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- Renewal-frequency based leases: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- State-based leases: The more loaded a server is, the shorter the expiration times become

Question: Why are we doing all this?

Epidemic Algorithms

Basic idea: Assume there are no write-write conflicts:

- Update operations are initially performed at one or only a few replicas
- A replica passes its updated state to a limited number of neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards

Gossiping: A replica which has just been updated (i.e., has been **contaminated**), tells a number of other replicas about its update (contaminating them as well).

System Model

- We consider a collection servers, each storing a number of objects
- Each object O has a *primary* server at which updates for O are always initiated (avoiding write-write conflicts)
- An update of object O at server S is always time-stamped; the value of O at S is denoted $VAL(O,S)$
- $T(O,S)$ denotes the timestamp of the value of object O at server S

Anti-Entropy

- **Basic issue:** When a server S contacts another server S^* to exchange state information, three different strategies can be followed:
 - **Push:** S only forwards all its updates to S^* :
if $T(O,S^*) < T(O,S)$
then $VAL(O,S^*) \leq VAL(O,S)$
 - **Pull:** S only fetched updates from S^* :
if $T(O,S^*) > T(O,S)$
then $VAL(O,S^*) \leq VAL(O,S)$
 - **Push-Pull:** S and S^* exchange their updates by pushing and pulling values.
 - **Observation:** if each server periodically randomly chooses another server for exchanging updates, an update is propagated in $O(\log(N))$ time units.
- Question:** why is pushing alone not efficient when many servers have already been updated?

Gossiping

Basic model: A server S having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, S stops contacting other servers with probability $1/k$. If s is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers:

$$s = e^{-(k+1)(1-s)}$$

k	s
1	0.2000
2	0.0600
3	0.0200
4	0.0070
5	0.0025

Observation: If we really have to ensure that all servers are eventually updated, gossiping alone is not enough

Deleting Values

Fundamental problem: We cannot remove an old value from a server and expect the removal to propagate. Instead, mere removal will be undone in due time using epidemic algorithms

Solution: Removal has to be registered as a special update by inserting a *death certificate*

Next problem: When to remove a death certificate (it is not allowed to stay forever):

- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)
- Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers)

Note: it is necessary that a removal actually reaches all servers.

Question: What's the scalability problem here?

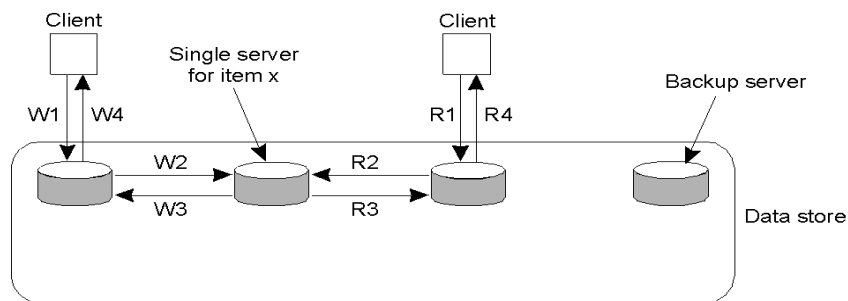
Consistency Protocols

Consistency protocol: describes the implementation of a specific consistency model. We will concentrate only on sequential consistency.

- Primary-based protocols
- Replicated-write protocols
- Cache-coherence protocols

Primary-Based Protocols (1/4)

- All read and write operations go to server
- Example: used in traditional client-server systems that do not support replication.

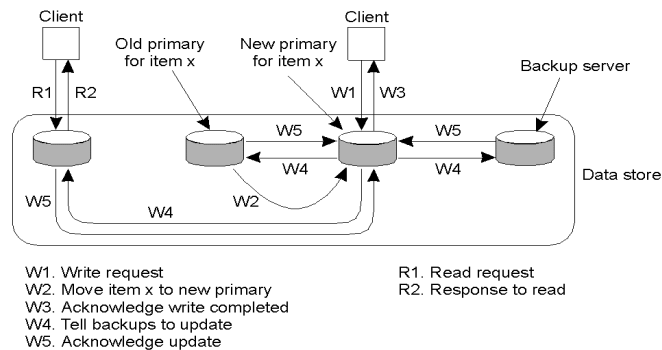


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

Primary-Based Protocols (4/4)

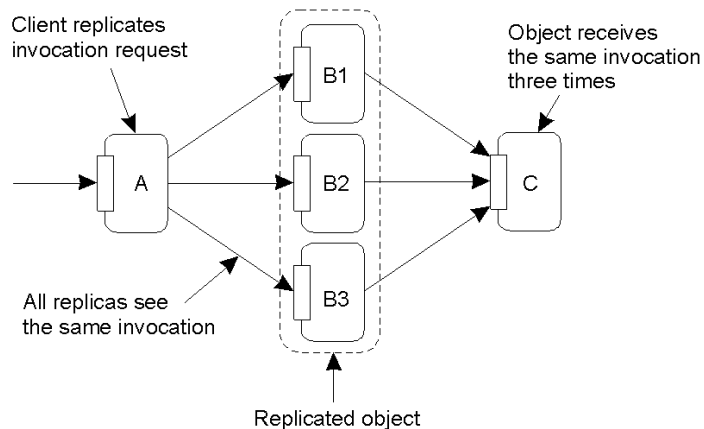
Primary-backup protocol with local writes: replicate data only for reading



Example: Distributed shared memory systems, but also mobile computing in *disconnected mode* (ship all relevant files to user before disconnecting, and update later on).

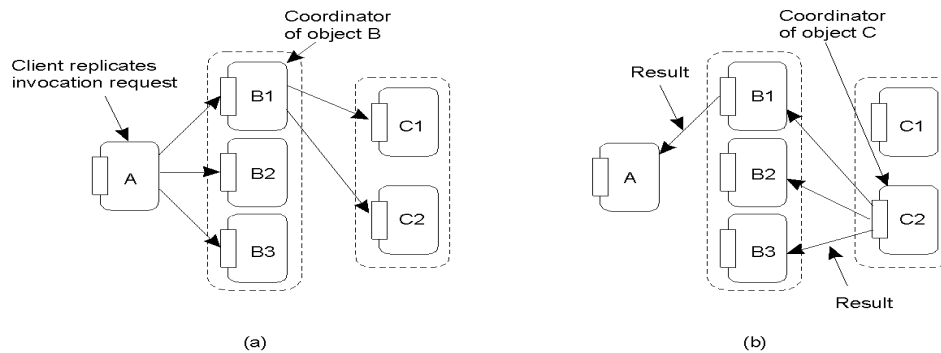
Replicated-Write Protocols(1/2)

- **Active replication:** Updates are forwarded to multiple replicas, where they are carried out.
- One problem to deal with: replicated invocations:



Replicated-Write Protocols (2/2)

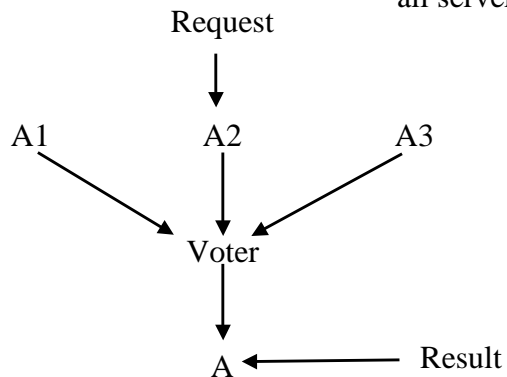
Replicated invocations: “Centralized” Solution Assign a coordinator on each side (client and server), which ensures that only one invocation (a), and one reply is send (b).



Triple Modular Redundancy

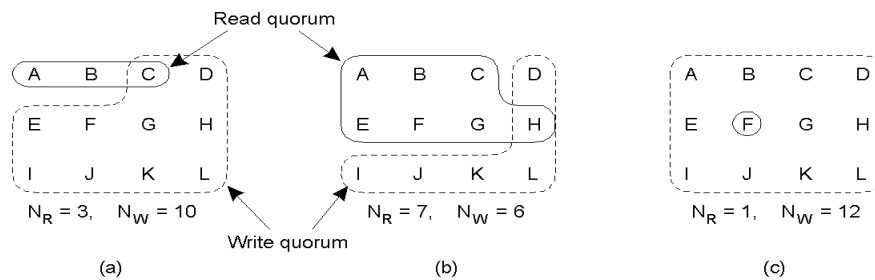
- Simple to implement
- Vote on all three results
- Majority (50% + 1) wins

Request is replicated to all servers.



Quorum-Based Protocols

Quorum-based protocols: Ensure that each operation is carried out in such a way that a majority vote is established: distinguish **read quorum** and **write quorum**:



Example: Lazy Replication

- Basic model: number of replica servers jointly implement a causal-consistent data store.
- Clients normally talk to **front ends** which maintain data to ensure causal consistency.

