Distributed Systems
Principles and Paradigms

Chapter 06
(version 11th October 2001)

Maarten van Steen
Vrije Universiteit Amsterdam, Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 444 7784
E-mail: steen@cs.vu.nl, URL: www.cs.vu.nl/∼steen

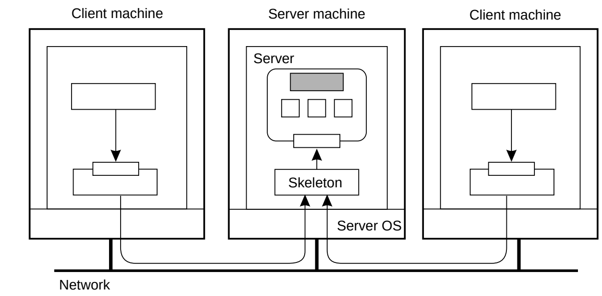
01 Introduction
02 Communication
03 Processes
04 Naming
05 Synchronization
06 Consistency and Replication
07 Fault Tolerance
08 Security
09 Distributed Object-Based Systems
10 Distributed File Systems
11 Distributed Document-Based Systems
12 Distributed Coordination-Based Systems

Consistency & Replication

- Introduction (what's it all about)
- Data-centric consistency
- Client-centric consistency
- Distribution protocols
- Consistency protocols
- Examples

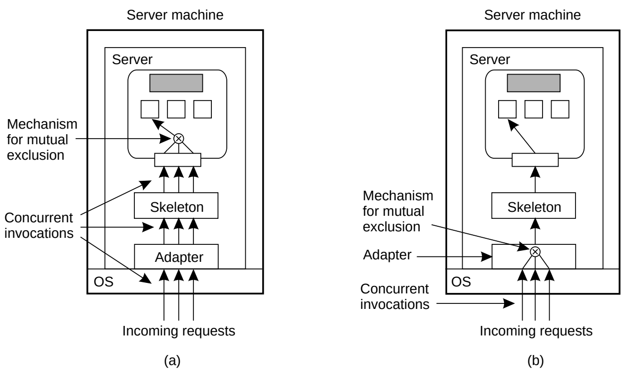
Shared Objects

Problem: If objects (or data) are shared, we need to do something about concurrent accesses to guarantee state consistency.



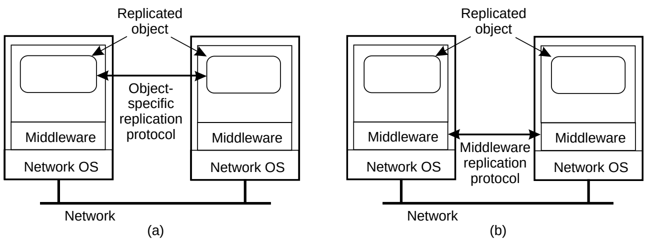
Concurrency Control (1/2)

Problem: Is the remote object already thread-safe or not?



Concurrency Control (2/2)

Problem: Should we seek for object-specific solutions, or generally applicable ones?



Question: Why would we want object-specific replication protocols?

Performance and Scalability

Main issue: To keep replicas consistent, we generally need to ensure that all *conflicting* operations are done in the the same order everywhere

Conflicting operations: From the world of transactions:

- Read–write conflict: a read operation and a write operation act concurrently
- Write–write conflicts: two concurrent write operations

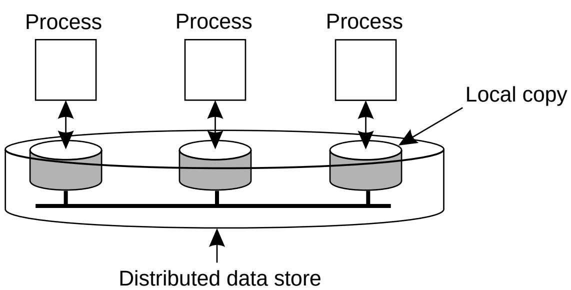
Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability

Solution: weaken consistency requirements so that hopefully global synchronization can be avoided

Data-Centric Consistency Models (1/2)

Consistency model: a contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

Essence: A data store is a distributed collection of storages accessible to clients:



Data-Centric Consistency Models (2/2)

Strong consistency models: Operations on shared data are synchronized:

- Strict consistency (related to time)
- Sequential consistency (what we are used to)
- Causal consistency (maintains only causal relations)
- FIFO consistency (maintains only individual ordering)

Weak consistency models: Synchronization occurs only when shared data is locked and unlocked:

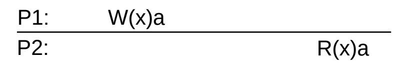
- General weak consistency
- Release consistency
- Entry consistency

Observation: The weaker the consistency model, the easier it is to build a scalable solution.

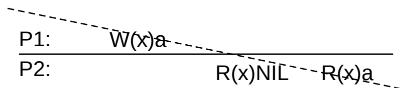
Strict Consistency

Any read to a shared data item X returns the value stored by the most recent write operation on X.

Observation: It doesn't make sense to talk about "the most recent" in a distributed environment.



(a)



(b)

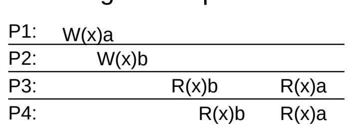
- Assume all data items have been initialized to NIL
- W(x)a: value a is written to x
- R(x)a: reading x returns the value a

Note: Strict consistency is what you get in the normal sequential case, where your program does not interfere with any other program.

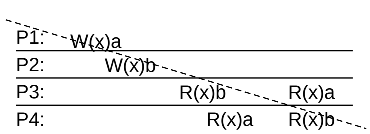
Sequential Consistency

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

Note: We're talking about **interleaved** executions: there is some total ordering for all operations taken together.



(a)



(b)

Linearizable: Sequential plus operations are ordered according to a global time.

Causal Consistency

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order by different processes.

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a			
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

FIFO Consistency

Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

P1: W(x)a						
P2:	R(x)a	W(x)b	W(x)c			
P3:				R(x)b	R(x)a	R(x)c
P4:				R(x)a	R(x)b	R(x)c

Weak Consistency (1/2)

- *Accesses to **synchronization variables** are sequentially consistent.*
- *No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
- *No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.*

Basic idea: You don't care that reads and writes of a *series* of operations are immediately known to other processes. You just want the *effect* of the series itself to be known.

Weak Consistency (2/2)

P1:	W(x)a	W(x)b	S			
P2:				R(x)a	R(x)b	S
P3:				R(x)b	R(x)a	S

(a)

P1:	W(x)a	W(x)b	S		
P2:				S	R(x)a

(b)

Observation: Weak consistency implies that we need to lock and unlock data (implicitly or not).

Release Consistency

Idea: Divide access to a synchronization variable into two parts: an **acquire** and a **release** phase. Acquire forces a requester to wait until the shared data can be accessed; release sends requester's local value to other servers in data store.

P1:	Acq(L)	W(x)a	W(x)b	Rel(L)				
P2:					Acq(L)	R(x)b	Rel(L)	
P3:								R(x)a

Entry Consistency

- With release consistency, all local updates are propagated to other copies/servers during release of shared data.
- With entry consistency, each shared data item is associated with a synchronization variable.
- When acquiring the synchronization variable, the most recent values of its associated shared data item are fetched.

Note: Where release consistency affects *all* shared data, entry consistency affects only those shared data associated with a synchronization variable.

Question: What would be a convenient way of making entry consistency more or less transparent to programmers?

Summary

Model	Description
Strict	Absolute time ordering of all shared accesses
Lin.	All processes see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Seq.	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order
FIFO	All processes see writes from each other in the order they were issued. Writes from different processes may not always be seen in that order
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered

Client-Centric Coherence Models

- System model
- Monotonic reads
- Monotonic writes
- Read-your-writes
- Write-follows-reads

Client-Centric Consistency Models

Goal: Show how we can perhaps avoid systemwide consistency, by concentrating on what specific *clients* want, instead of what should be maintained by servers.

Background: Most large-scale distributed systems (i.e., databases) apply replication for scalability, but can support only weak consistency:

DNS: Updates are propagated slowly, and inserts may not be immediately visible.

NEWS: Articles and reactions are pushed and pulled throughout the Internet, such that reactions can be seen before postings.

Lotus Notes: Geographically dispersed servers replicate documents, but make no attempt to keep (concurrent) updates mutually consistent.

WWW: Caches all over the place, but there need be no guarantee that you are reading the most recent version of a page.

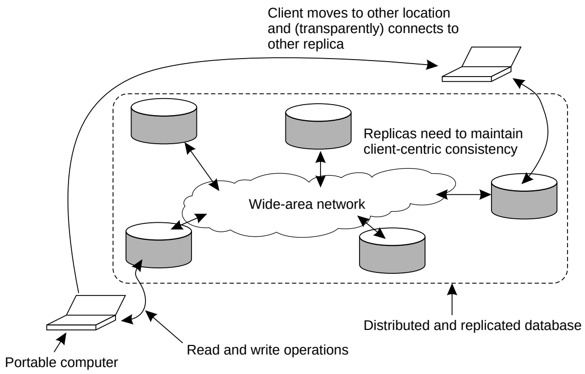
Consistency for Mobile Users

Example: Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.

- At location A you access the database doing reads and updates.
- At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
  - your updates at A may not have yet been propagated to B
  - you may be reading newer entries than the ones available at A
  - your updates at B may eventually conflict with those at A

Note: The only thing you really want is that the entries you updated and/or read at A, are in B the way you left them in A. In that case, the database will appear to be consistent *to you*.

Basic Architecture



Monotonic Reads (1/2)

If a process reads the value of a data item x, any successive read operation on x by that process will always return that same or a more recent value.

L1:	$WS(x_1)$		$R(x_1)$	
L2:		$WS(x_1;x_2)$		$R(x_2)$

(a)

L1:	$WS(x_1)$		$R(x_1)$		
L2:		$WS(x_2)$		$R(x_2)$	$WS(x_1;x_2)$

(b)

Notation: $WS(x_i[t])$ is the set of write operations (at L_i) that lead to version x_i of x (at time t); $WS(x_i[t_1];x_j[t_2])$ indicates that it is known that $WS(x_i[t_1])$ is part of $WS(x_j[t_2])$.

Note: Parameter t is omitted from figures

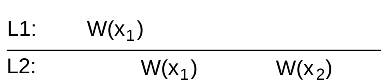
Monotonic Reads (2/2)

Example: Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

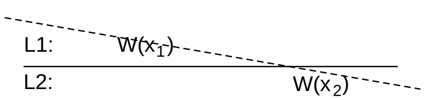
Example: Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

Monotonic Writes

A write operation by a process on a data item x is completed before any successive write operation on x by the same process.



(a)



(b)

Example: Updating a program at server S_2, and ensuring that all components on which compilation and linking depends, are also placed at S_2.

Example: Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

Read Your Writes

The effect of a write operation by a process on data item x, will always be seen by a successive read operation on x by the same process.

L1: $W(x_1)$
L2: $WS(x_1;x_2)$ $R(x_2)$

(a)

L1: $W(x_1)$
L2: $WS(x_2)$ $R(x_2)$

(b)

Example: Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

Writes Follow Reads

A write operation by a process on a data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x that was read.

L1: $WS(x_1)$ $R(x_1)$
L2: $WS(x_1;x_2)$ $W(x_2)$

(a)

L1: $WS(x_1)$ $R(x_1)$
L2: $WS(x_2)$ $W(x_2)$

(b)

Example: See reactions to posted articles only if you have the original posting (a read "pulls in" the corresponding write operation).

Distribution Protocols

- Replica Placement
- Update Propagation
- Epidemic Protocols

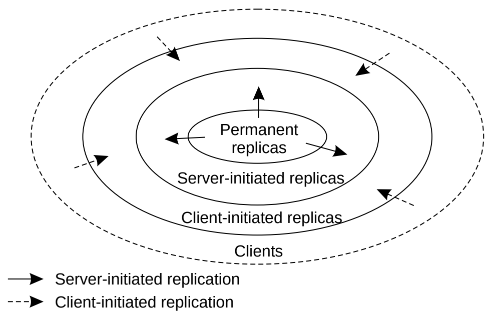
Replica Placement (1/2)

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

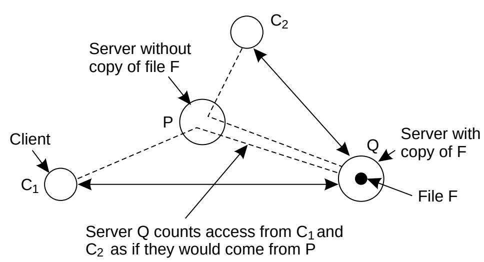
Distinguish different processes: A process is capable of hosting a replica of an object or data:

- **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 (2/2)



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

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

- General background
- Update models
- Removing objects

Principles

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 timestamped; 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 $VAL(O,S^*) \leftarrow VAL(O,S)$

Pull: S only fetches updates from S^*:

if $T(O,S^*) > T(O,S)$
then $VAL(O,S) \leftarrow VAL(O,S^*)$

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 for ever):

- 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?

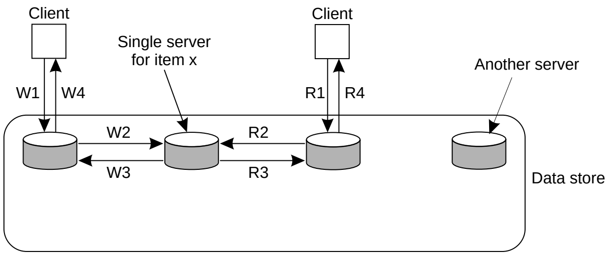
Consistency Protocols

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)

Primary-based, remote-write, fixed server:



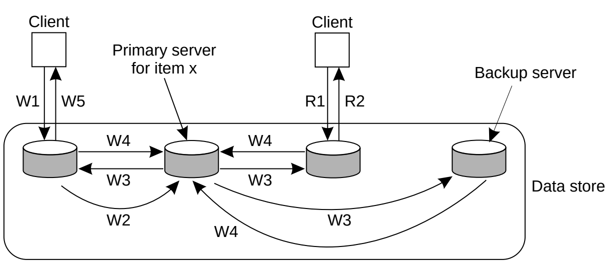
W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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

Example: Used in traditional client-server systems that do not support replication.

Primary-Based Protocols (2/4)

Primary-backup protocol:



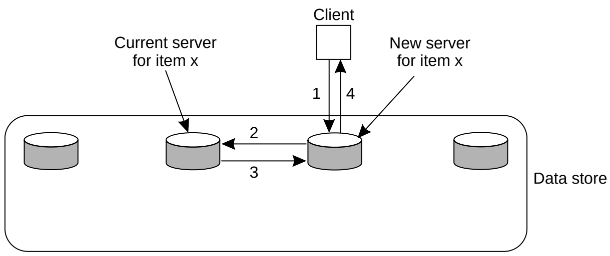
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Example: Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on same LAN.

Primary-Based Protocols (3/4)

Primary-based, local-write protocol:

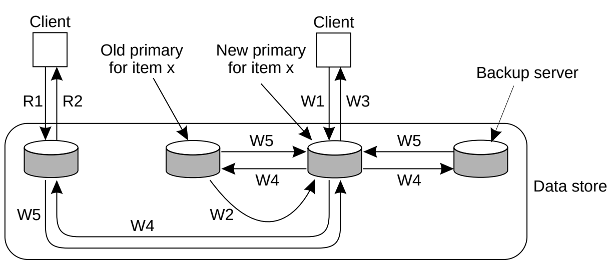


1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Example: Establishes only a fully distributed, non-replicated data store. Useful when writes are expected to come in series from the same client (e.g., mobile computing without replication)

Primary-Based Protocols (4/4)

Primary-backup protocol with local writes:



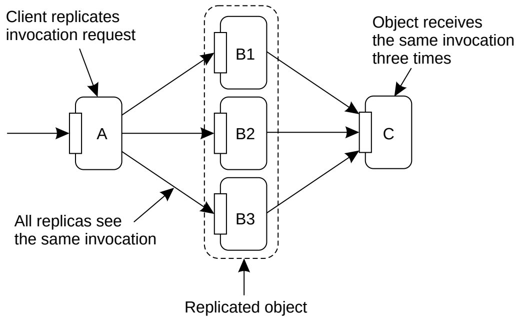
W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

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

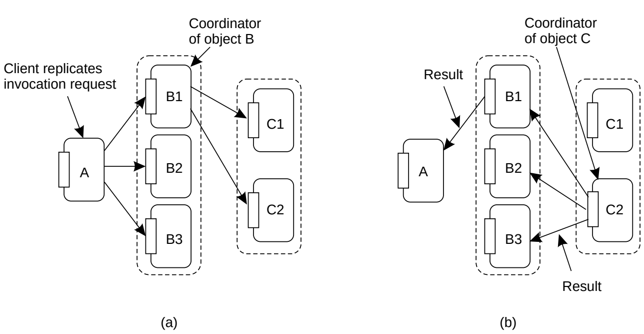
Replicated-Write Protocols (1/3)

Active replication: Updates are forwarded to multiple replicas, where they are carried out. There are some problems to deal with in the face of replicated invocations:



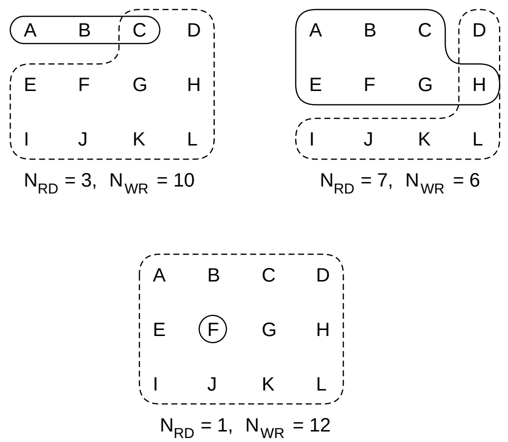
Replicated-Write Protocols (2/3)

Replicated invocations: Assign a coordinator on each side (client and server), which ensures that only one invocation, and one reply is sent:



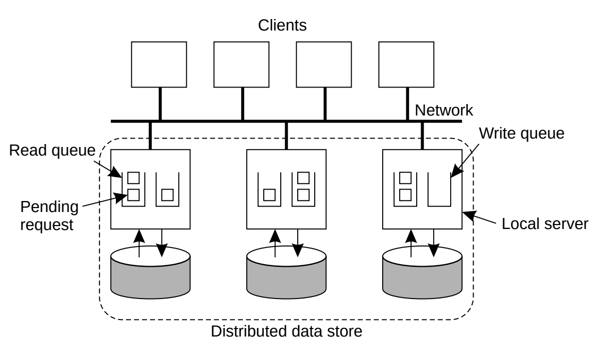
Replicated-Write Protocols (3/3)

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.



Lazy Replication: Vector Timestamps

VAL(i): VAL(i)[i] denotes the total number of write operations sent directly by a front end (client). VAL(i)[j] denotes the number of updates sent from replica #j.

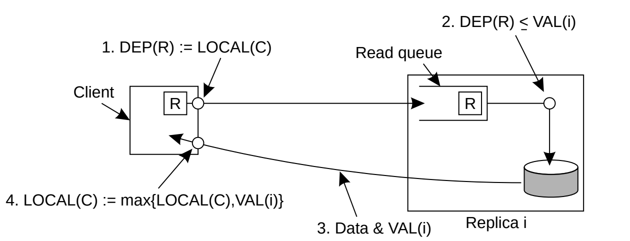
WORK(i): WORK(i)[i] total number of write operations directly from front ends, including the pending ones. WORK(i)[j] is total number of updates from replica #j, including pending ones.

LOCAL(C): LOCAL(C)[j] is (almost) most recent value of VAL(j)[j] known to front end C (will be refined in just a moment)

DEP(R): Timestamp associated with a request, reflecting what the request depends on.

Operations

Read operations:



Write operations: