Distributed Systems

15 Replication Management

July 13 2009

Gerd Liefländer

System Architecture Group

Outline

- Replica-Server Placement
- Content Replication and Placement
  - Permanent Replicas
  - Server-Initiated Replicas
  - Client Initiated Replicas
- Content Distribution
  - State versus Operation
  - Pull versus Push Protocols
  - Uni- versus Multicasting
- Consistency Protocols
  - Primary-Based Protocols
  - Replicated-Write Protocols
  - Cache-Coherence Protocols
  - Implementing Client-Centric Consistency
- Examples

Replica Management

Placement Problem

Content Replication

Node Initiatives

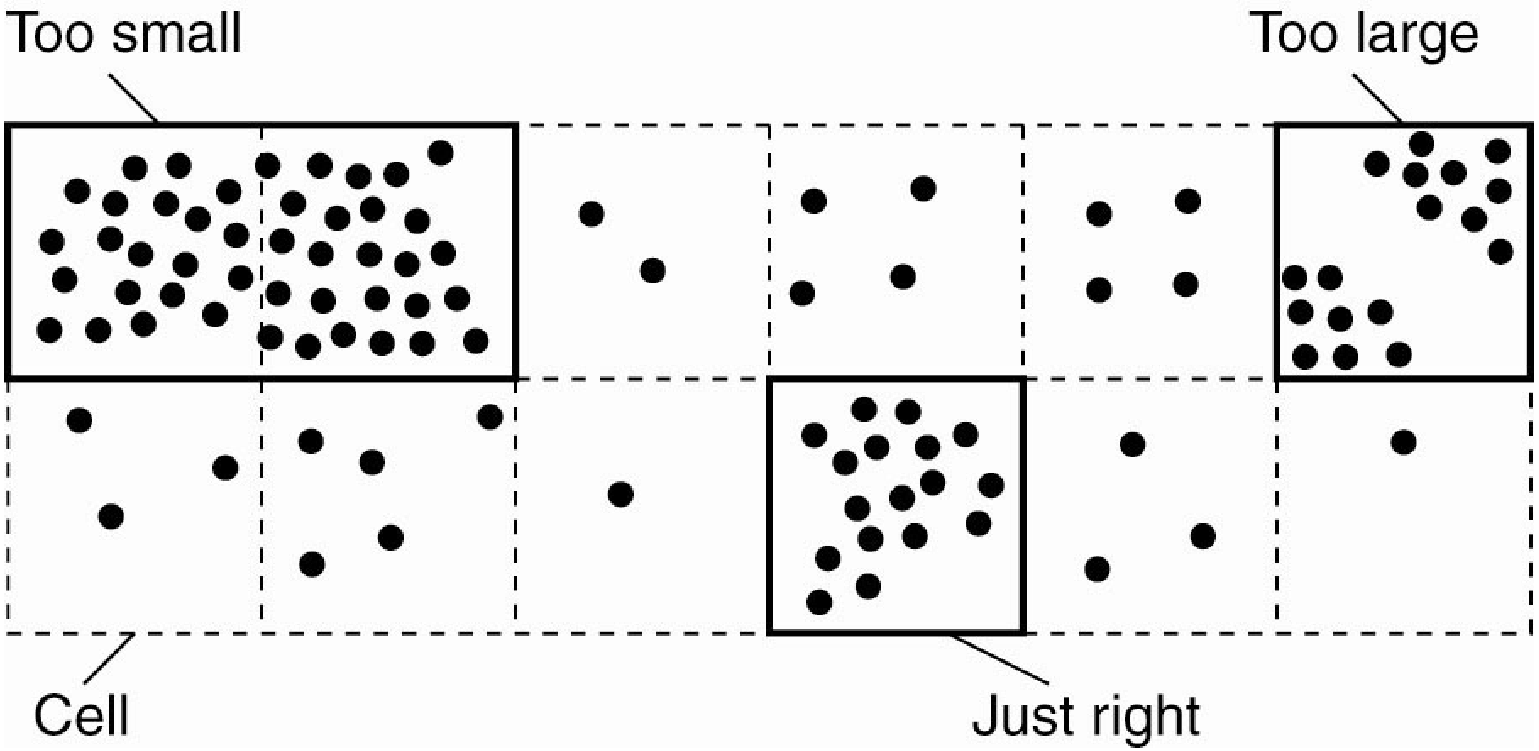
Placement Problem

- *Where to install replica servers?*
  - Find the appropriate (best) node(s) to place a replica server that can host (part of) the DDS
- *Where and how to store the content of a DDS?*
  - Find best server for placing a content of the DDS
- Before we discuss content placement in a DS, replication servers have to be installed

Replica-Server Placement

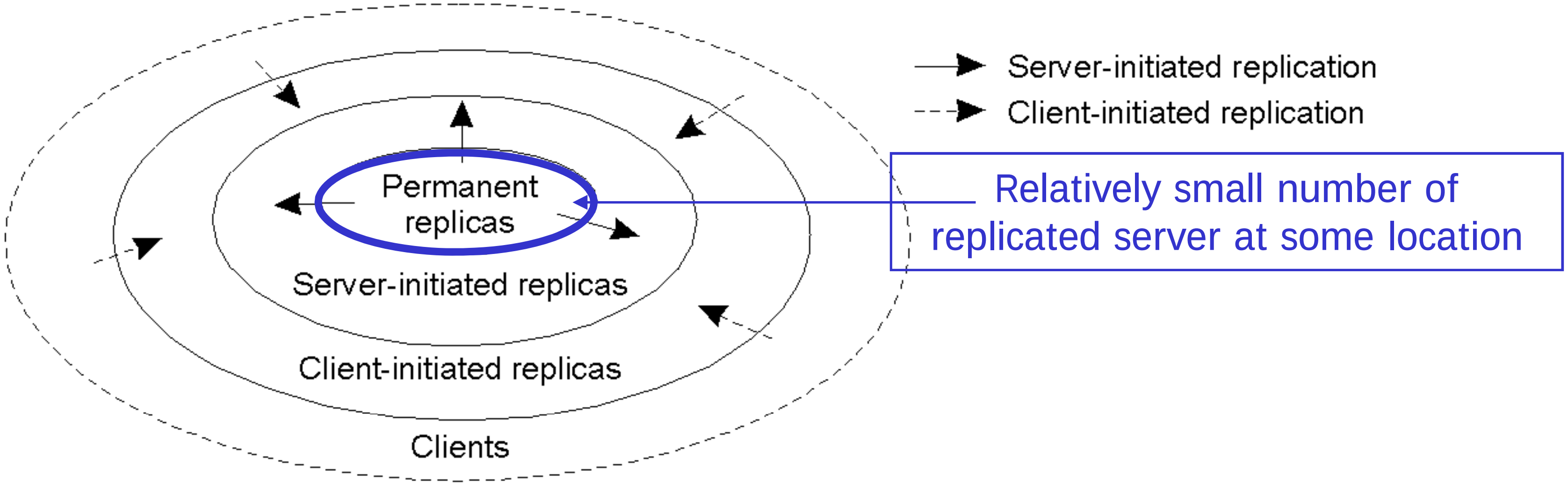
- Suppose ∃ N>1 nodes
- Find the best k<N nodes to host the replicas
- Qiu's solution:
  - Measure the distance (in terms of delay or latency)
  - Take the host that minimizes the average distance between clients and server
- Radoslavov's solution:
  - Take topology of the Internet as formed by autonomous systems (AS)
  - Place the server on a host with the largest number of network interfaces, ...

Replica-Server Placement



- Choosing a proper cell size for server placement
- Goal: find well-suited clusters of nearby host and chose one host among each cluster

Content Replication & Placement



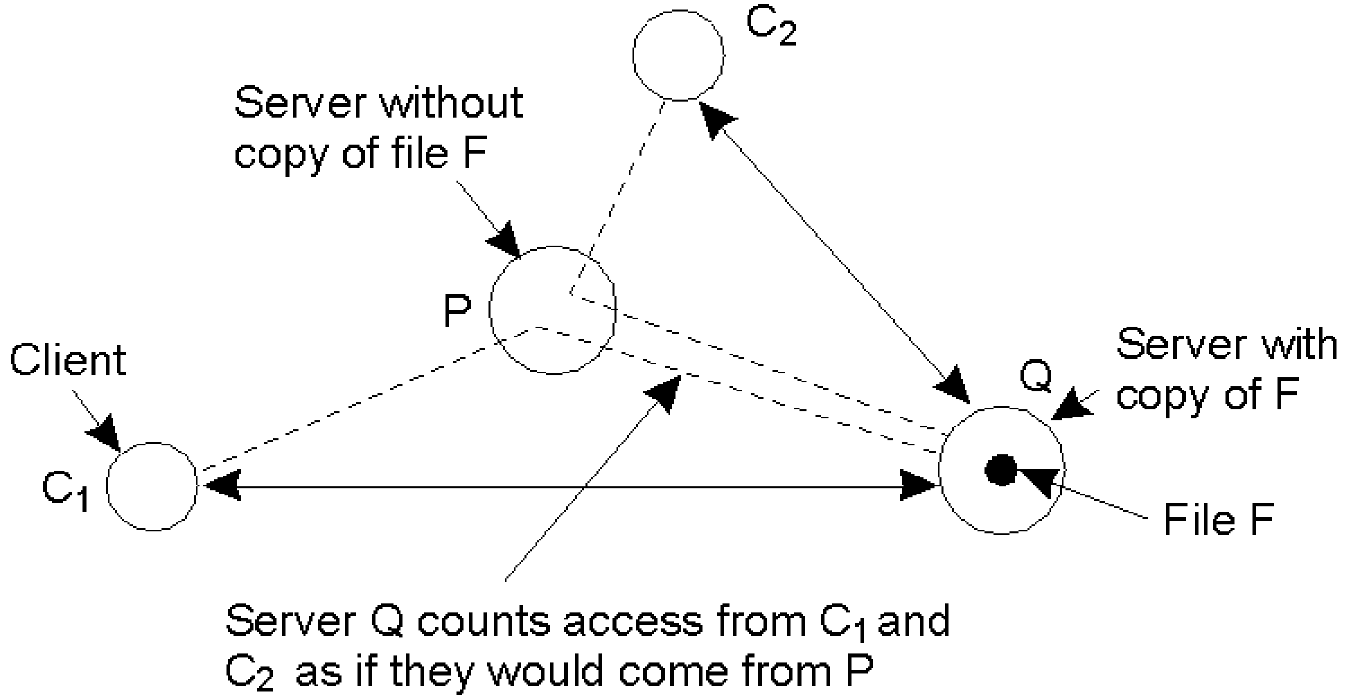
- Logical organization of different kinds of replicas of a DDS using three concentric rings
- *Where to store which replicas and for how long?*
  - Static versus dynamic replicas
  - Server or client initiated

Permanent Replicas

- Initial set of replicas
  - Created and maintained by DDS-owner
  - Writes are only allowed by DDS-owner
  - Prefer strong consistency models
  - Often geographically distributed to improve
    - performance
    - reliability

- <u>Examples:</u>
  - DNS-server: primary- and secondary server

Server-Initiated Replicas



- Counting access requests from different clients sites
- Server Q installs an additional replica P if too many request are counted from clients site C_1 and C_2
- Replicate total DDS or only parts of the DDS

Server-Initiated Replicas

- Dynamically installed replicas due to server contention $\Rightarrow$
  - Enhance performance and reliability
  - Often not maintained by owner of DDS
  - Placed close to mega-groups of clients
- Replicas are created close to the majority of (new) clients whenever $\exists$ demand "spikes"
- Only delete replica when demand significantly falls below a low threshold
- Use weaker consistency models for server initiated replicas than for permanent ones

Client-Initiated Replicas

- Dynamic installation by client’s actions, e.g.
  - Temporary client caches
    - DNS-caching server
    - Web-browser
- DDS-Owner is not aware of those “replicas”
- Placed very close to a client
- Maintained by host (often the client)
- Especially useful when #reads >> #writes

Client-Initiated Replicas (Caches)

- Managing content of client caches is left to clients
- Problem: stale data in client's cache
  - Data are cached only for a limited amount of time
  - Clients can rely on their local physical clock
- Data have to be removed, if space in client's cache is needed for other data to be cached
  - *What replacement policy is appropriate?*
- Caches can be shared by more than one client ⇒ improves the number of cache hits if clients access the same part of the DDS
- Servers very close to clients may keep those data

Content Distribution

State versus Operations
Pull versus Push Protocols
Unicasting versus Multicasting

State versus Operations

Possibilities for propagation:

1. Propagate only a notification of an update
2. Transfer "updated or new data" from one copy to another (e.g. complete files with version numbers)
3. Propagate update operations (including all parameters) to other copies

gl1

Slide 14

gl1 Gerd Liefländer; 09.07.2007

Invalidation Notifications

- Updating node notifies all other replicas that a specific part of the DDS has changed, i.e. that local replicated data are no longer valid
- Invalidation notifications are relatively short, thus needing only few network bandwidth
- These method works quite well when there are many updates in relation to reads
- It is up to the replicas when they will update their contents, e.g. only when clients access the updated parts of the DDS

Propagate Notifications

- Propagate only a notification of an update (e.g. to invalidate outdated replicas)
- Via a notification a local replica knows that an update has taken place somewhere $\Rightarrow$ local replica must be updated before next read can take place
- Update of a local replica can be done lazily, i.e. you might collect a set of invalidation notifications
  - Typical for invalidation protocols
  - Can include information which part of the DDS has been updated
  - Works best, when ratio of #reads/#write is low

Propagate Updated Data

- Propagate updated data from one replica to another
  - Works well when tha ratio of reads/writes is high
  - If many data have to be changed $\Rightarrow$ too much overhead
  - Again, you can collect u>1 updates before propagating
- An update message tells local replica how the DDS has changed
- Often correlated with the push-model (i.e. server initiated)
- <u>Advantage:</u>
  - No additional communication needed to update
  - Might be done asynchronously to all application processes

Propagate Update-Operation

- Sometimes also called "active replication"
- Replica gets a message telling what to do on what data (part of the DDS)
- Advantages:
  - Approach works well if size of parameters + operation is small compared to updated data
- Disadvantage:
  - Local operations must deliver the same result

Push Protocol

- Server based protocol
  - i.e. updates are propagated to all other replicas (whether those replicas have asked for or not)
  - Often used between permanent replicas and server initiated replicas, i.e. to achieve a *relatively high degree of consistence* (i.e. replicas stay in close synchrony)
  - Efficient if #reads >> #writes
  - Whenever a rare update occurs propagate the updated values ASAP to the companion replicas

Pull Protocol

- Client-based protocol
  - Client (or other server) asks another server to provide its updates
  - Used by client caches, e.g. when a client requests a website, not having updated for a longer period of time, it checks the original web site, whether updates have been made in the mean time
  - Efficient if #reads ~~>~~> #writes

Pull versus Push Protocols

Less fault tolerant

	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Pull and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

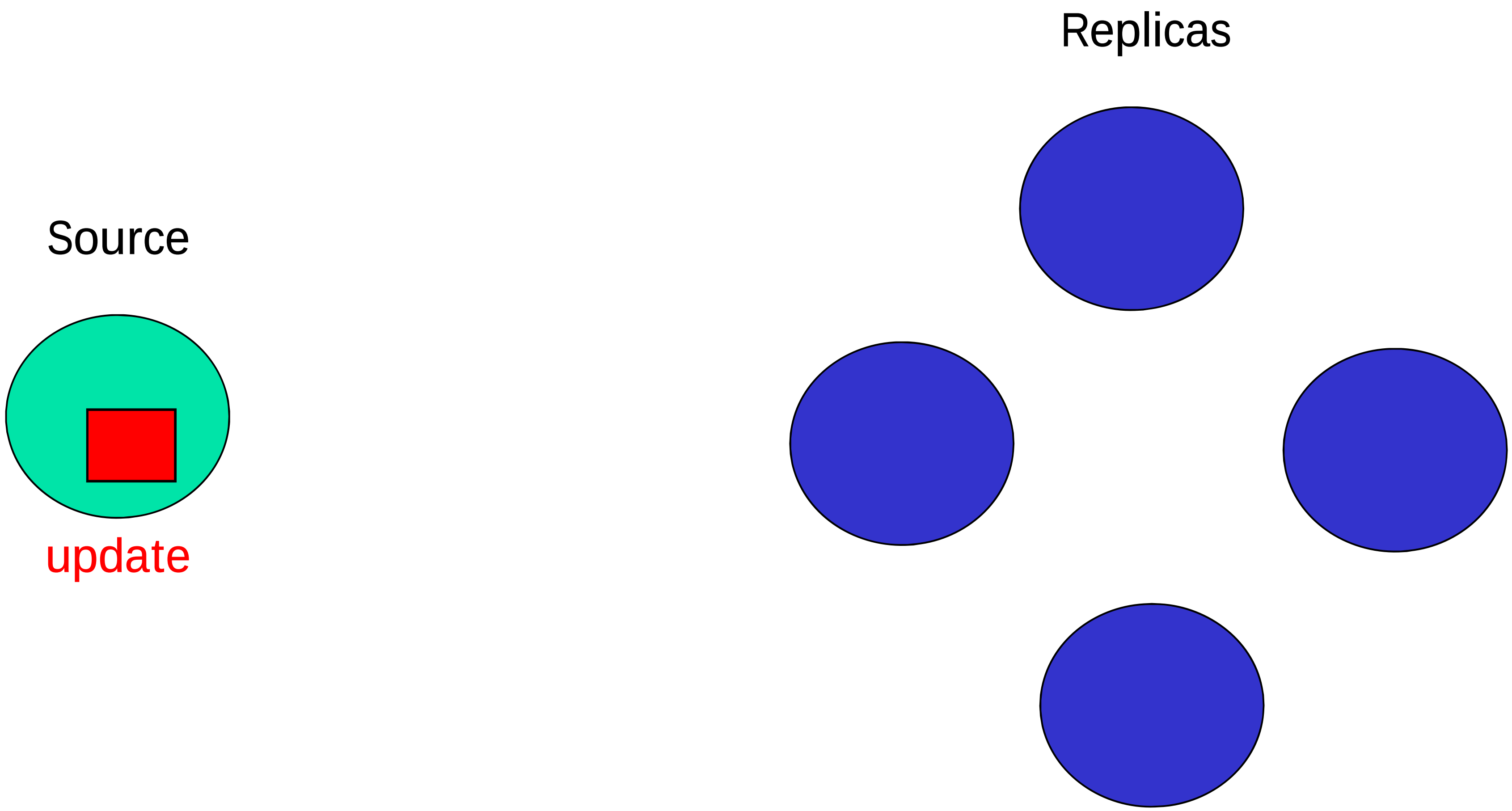
- Comparison between push-based and pull-based protocols in case of multiple clients, single server systems, (i.e. without any replicas)

Lease Protocol[1]

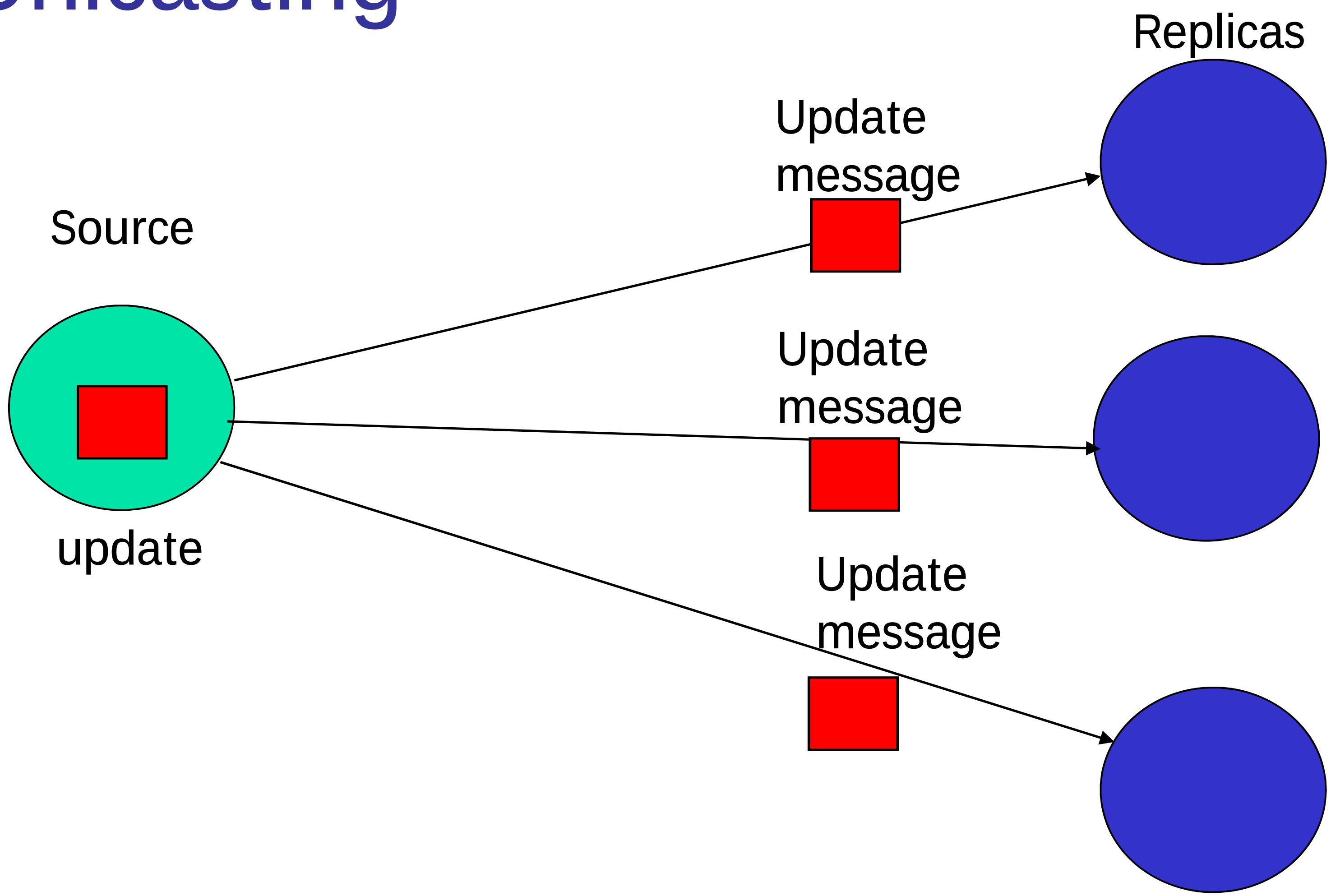
- Lease is a promise by a server to push updates to a client for a specified time
- When a lease expires, client must pull updates from the server
- Lease duration can depend on
  - Last time the data item has been updated, i.e. long leases for data that has not been updated for a long period of time
  - Frequency of updates
  - State space overhead at server, if states space overhead is too much, server lowers expiration time of new leases

[1]Duvvuri et al.: Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web", IEEE Trans.Kow.Data Eng., 2003

Problem: Update Propagation

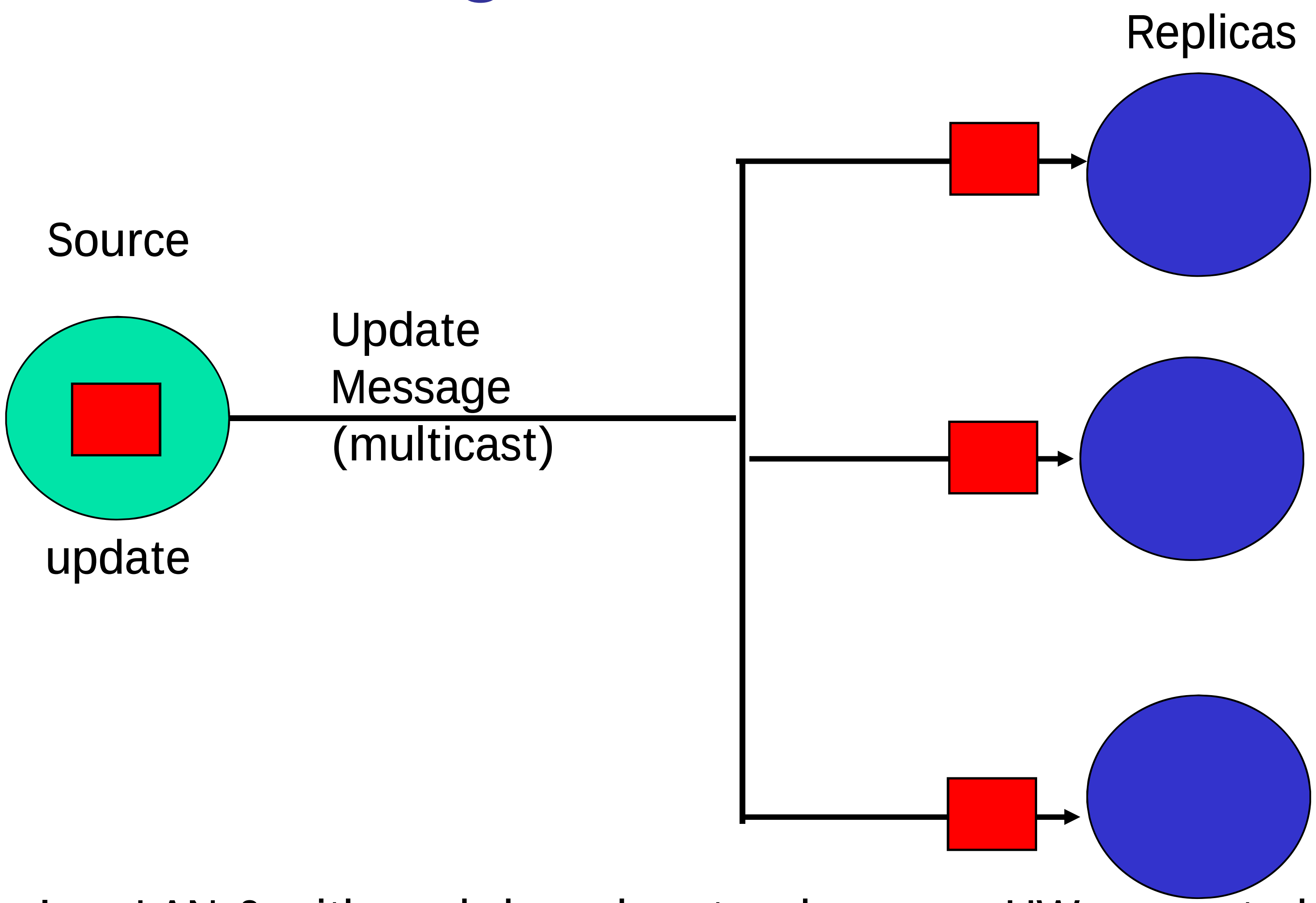


Unicasting



- With push based protocols avoidable overhead with unicasting in a LAN

Multicasting



- In a LAN & with push-based protocol you use HW-supported multicast

Consistency Protocols

- Continuous Consistency
- Primary-Based Protocols
- Replicated-Write Protocols
- Cache-Coherence Protocols
- Client-Centric Consistency

Limiting Numerical Deviation

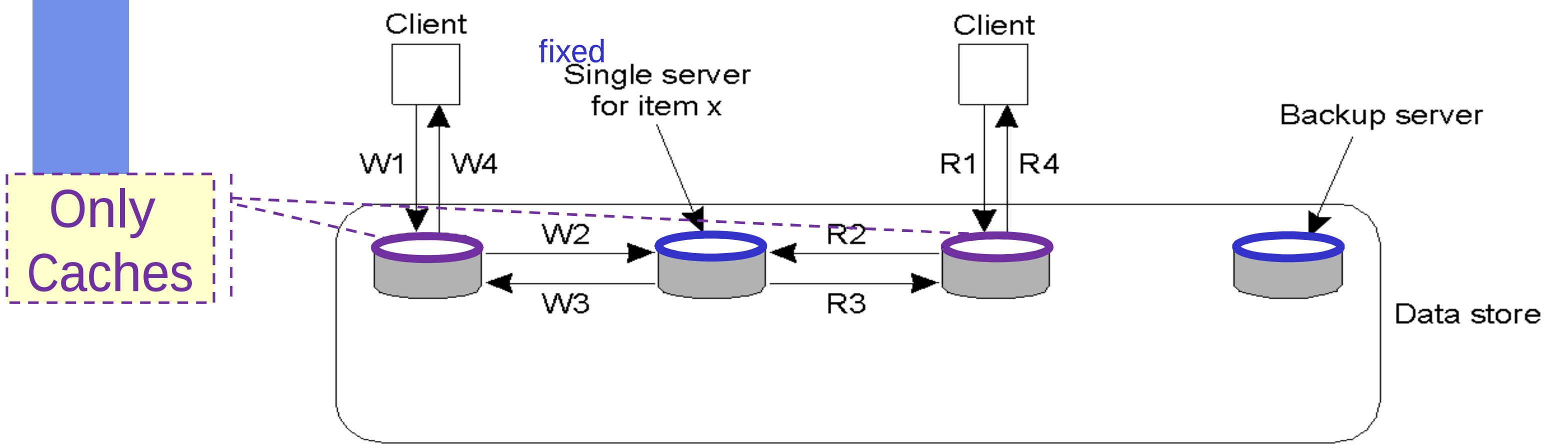
- Focus on writes to a single data item x
- Idea: Each site s_i will keep track of a log L_i of writes that it has performed on its own replica of x
- Propagation can use epidemic algorithms to spread everywhere (at least after some time)
- If some server detects that a certain site does not keep pace with all other sites it can propagate the missing writes to that server

Primary-Based Protocols

<u>Preliminaries:</u>

- Each data item x of a DDS has an associated primary, responsible for coordinating write operations on x
- Often (a larger subset of) the DDS is hosted on only one primary server
- A primary server can be installed as a
  - fixed server, i.e. a specific remote server, i.e. most of the updates are remote-writes
  - dynamic server, i.e. the primary migrates to the location of the next write

Remote-Write Protocols (1)

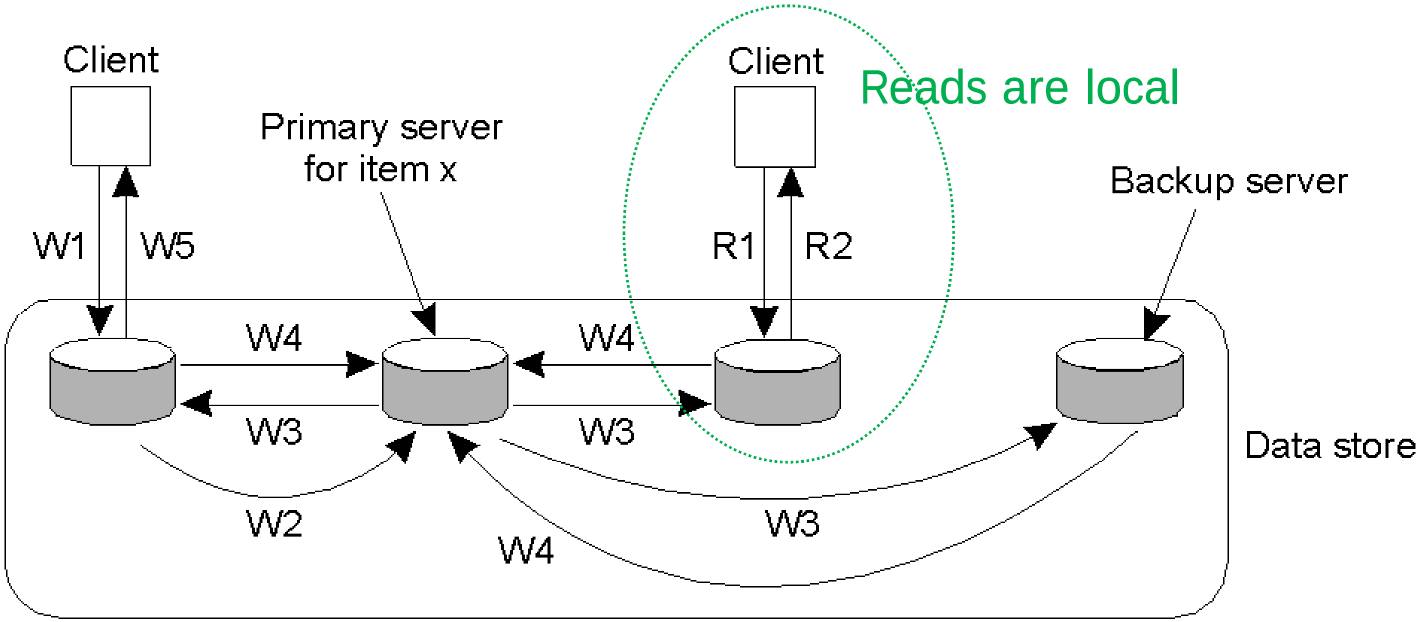


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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

- Primary-based remote-write protocol with a fixed server to which **all** read and write operations are forwarded
- Primary server will be a bottleneck (without caching)
- DDS = {primary server , backup server}, the other sites are only caches

Remote-Write Protocols (2)

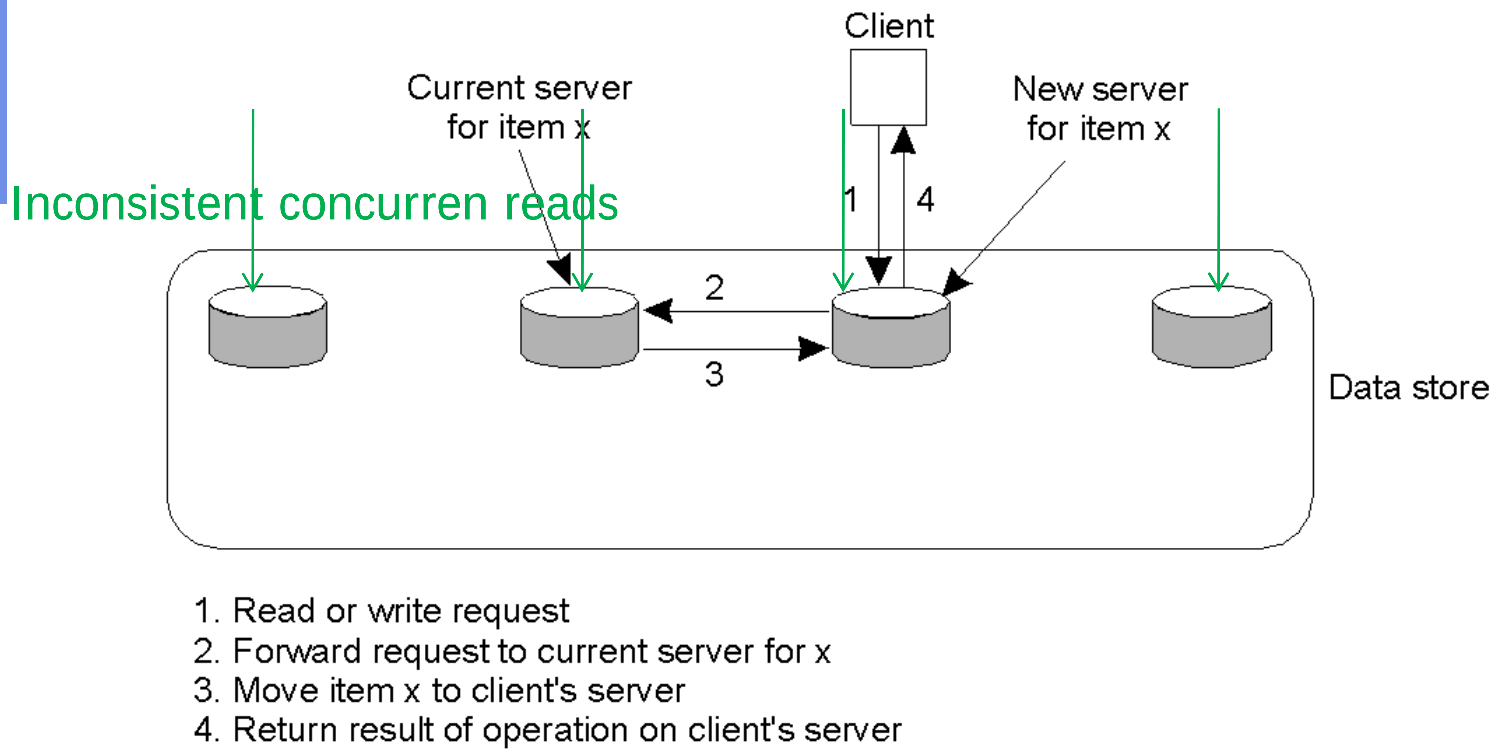


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

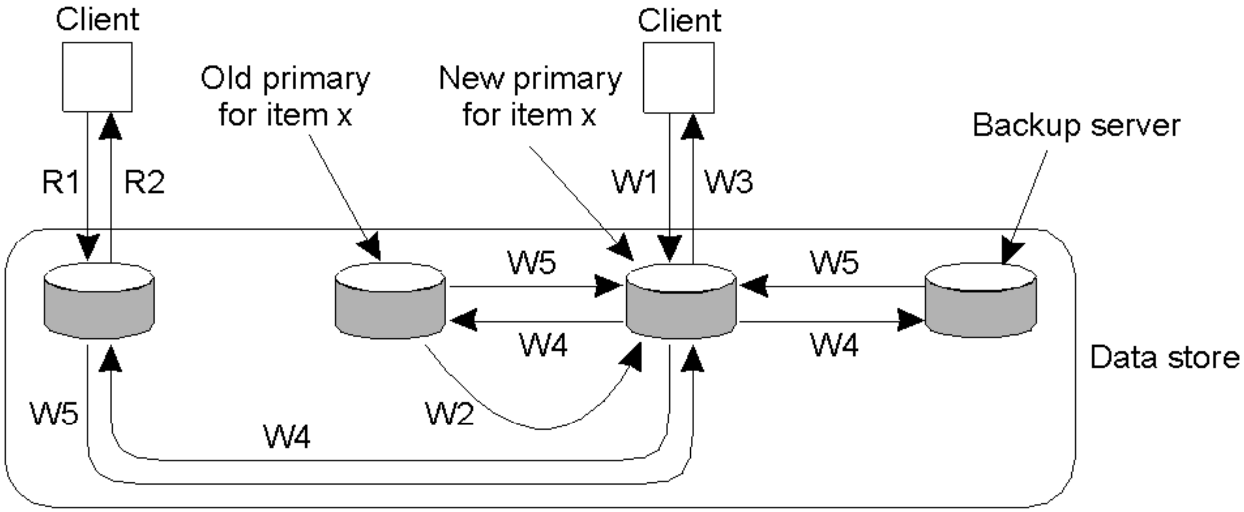
- The principle of primary-backup protocol
- Write to primary, propagate updates to all replicas

Local-Write Protocols (1)



- Primary-based local-write protocol in which a single copy is migrated between processes (no replicas)
- Multiple successive writes are done localy, propagation to the other replicas is done lazily, only eventual consistency is achievable

Local-Write Protocols (2)



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

- Primary-backup protocol in which the primary copy always migrates to the process wanting to perform an update
- Reads can be done locally, however stale data can be read
- You can improve this solution, if before writing to data item x, you invalidate all current replica of x

Replicated-Write Protocols

Preliminaries:

Writes take place at multiple replicas, i.e. no longer restricted to happen on a static or dynamic primary

- Active replication
  - Operation is forwarded to all replicas
- Majority voting
  - Before reading or writing ask a subset of all replicas

Active Replication

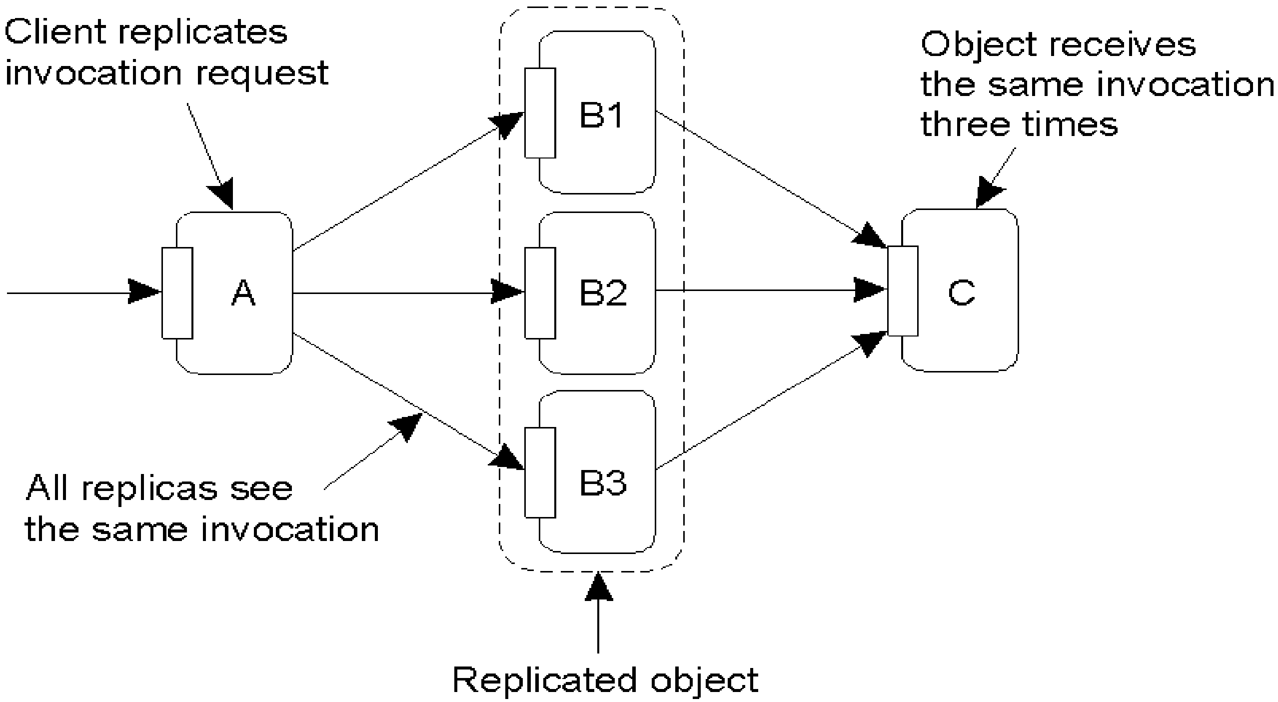
- Execute the update operation on all replicas

Preconditions:

Identical sequence of updates on all replicas (according to a strong consistency model):

- Via time stamps
- Via totally ordered multi-cast transport protocol
- Via a centralized coordinator (sequencer)
  - adding sequence number per update-operation
- Via a distributed consensus algorithms

Problem with Active Replication

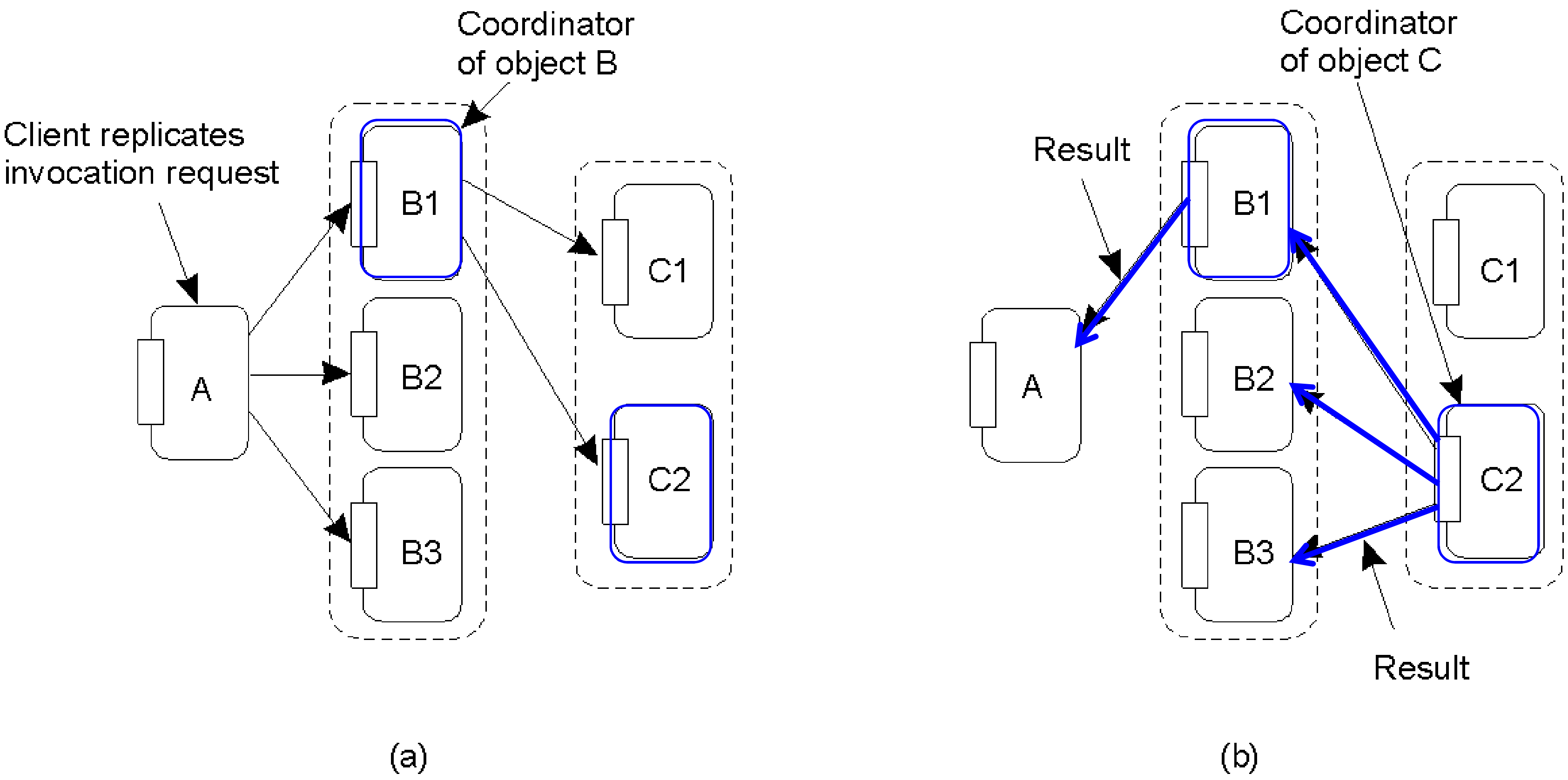


- "Chained or hierarchical remote object invocations"
- Calling object C from replicated object B will take place as often as an update to a replicated object B is done

Solution

- Suppose: ∃ a centralized coordinator in one of the replicated objects, e.g. in B_0
  - This special object forwards the call to a lower object and receives its reply
  - This special object B_0 distributes this result from C to all corresponding replicated objects B_i

Solution: Active Replication



a) Forwarding an invocation request from a replicated object.

b) Returning a reply to a replicated object.

Voting & Epidemic Protocols

Voting Algorithms

Thomas Quorum

Clifford Quorum

Epidemic Algorithms

Anti Entropy

Gossiping

Quorum-Based Protocol (R.Thomas)

Preliminaries:

If a client wants to read or write, it first must request and acquire permission of a majority of all servers.

Example:

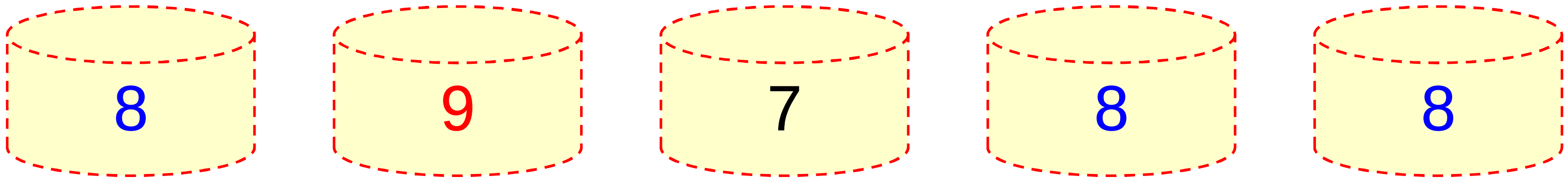
A DFS with file F being replicated on N>1 file servers. If a client wants to write to F, it first has to contact (N/2 + 1) servers, and get them to agree to do its intended update.

Once, they have agreed, file F gets a new version number v_n
To read file F, client must contact at least (N/2+1) servers and ask them to send the current version number of F.

- If all have same $v_n \Rightarrow$ file F represents the most recent version
- If not, take the newest version v_n , and propagate this new version to all stale servers

Example

- Suppose you have 5 replicas
  - Client wants to read file F and contacts 3 of them
  - All servers return the version number 8 for file F
  - Client can be sure that the other two replicas do not contain a newer version of F (e.g. version no 9), because any successful update from version 8 to 9 on any replica would had required that at least 3 replicas had agreed to it before



Another Quorum-Based Protocol[1]

Gifford quorum scheme is a bit more general:

To read a file f a client must use a *read-quorum,* an arbitrary assemble of N_r servers.

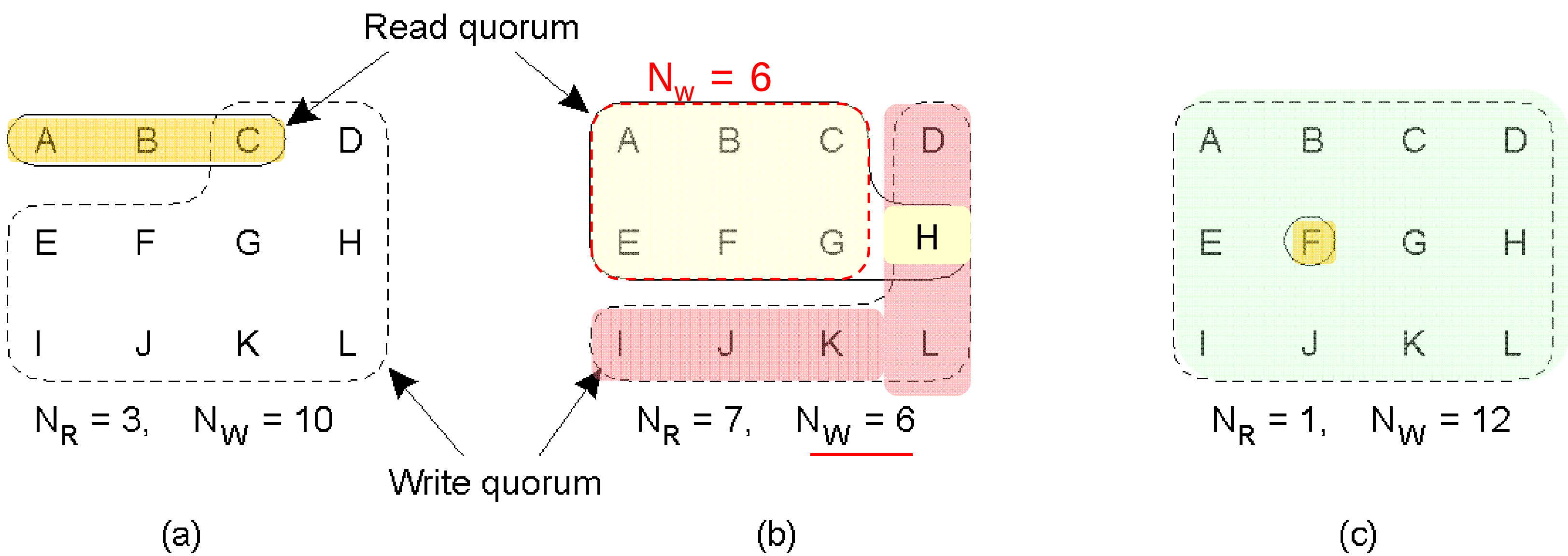
To write a file F, at least N_w servers = the *write quorum* is required. The following must hold:

1. $N_R + N_W > N$
2. $N_W > N/2$

1. Is used to prevent read-write conflicts
2. Is used to prevent write-write conflicts

[1]D. Gifford: “Weighted Voting for Replicated Data”, 7. SOSP, 79

Quorum-Based Protocols



- Three examples of Clifford's voting algorithm:

a) A correct choice of read and write set

b) A bad choice that may lead to write-write conflicts, because N_w is too small (violation of rule 2)

c) A correct choice, known as the ROWA protocol (read one, write all)

Epidemic Protocols

- To implement eventual consistency you can use epidemic protocols
- No guarantees for absolute consistency, but after some time epidemic protocols tend to have propagated all updates to all replicas
- To avoid write/write conflicts it is assumed that each update for a specific data item x is always done on a specific replica (static primary per data item) or by a specific process (owner)
- Goal: update all replicas or in other words: infect as many servers as fast as possible

Measures for Quality of Epidemics

- Propagation time required to propagate an updated data item to all replicas
- Network traffic generated in propagating the updates

Epidemic Protocols

Notions:

- An infectious server is a server with an up-to-date replica that is willingly to contact other servers in order to propagate its up-to-date values
- A susceptible server is a server that has not yet been updated, i.e. its content might be stale, i.e. it is not yet infectious
- A removed server is a server that does no longer want to contact other servers for updating new information

Anti-Entropy Protocol

Each server P periodically picks another server Q at random to exchange updates with Q:

∃ 3 approaches how to propagate updates:

- P only pushes its own updates to Q (i.e. pure push model)
- P only pulls in new updates from Q (i.e. pure pull model)
- P and Q exchange to each other their updates (i.e. push-pull approach)

Performance of anti-entropy approach:

- It can be shown that all servers are updated as long as algorithm starts with at least one infectious server
- Performance can be improved with n>1 infectious servers

Implementation Problem

- *How to determine which replica is up-to-date and which one is stale?*
- Exchange complete data base and compare
- Exchange checksums and ...
- Exchange update-logs and ...

Analysis: Anti-Entropy Protocol

<u>Pure push model</u>:

- Suppose already many servers are infectious ⇒
- It is quite probable that a random choice of Q will get an already infectious server ⇒
- It might take some time until the last server is updated

<u>Pure pull model or push/pull model?</u>

- ...

Gossip[1] Protocols

Rumor spreading or gossiping works as follows:

If server P has been updated (with a new value for data item x), it contacts another arbitrary server Q and pushes its new update of x to Q

However, if Q got this update already by some other server, P is so much disappointed, that it will stop gossiping with a probability 1/k

[1]works excellent in daily life

Gossip Protocols

Although gossiping really works quite well on average, you cannot guarantee that every server will be updated.

Demers showed, that in a DDS with a “large” number of replicas, the fraction s of servers remaining ignorant towards an update, i.e. are still susceptible is:

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

Example: $k = 1 \Rightarrow$ 20 % will miss the rumor
$k = 2 \Rightarrow$ only 6% will miss the rumor

Analysis of Epidemic Protocols

Advantages:

- Scalability, due to limited number of update messages

Disadvantage:

- Spreading the deletion of data is a problem (due to an unwanted side effect):
- Suppose, you have deleted on server S data item x, but you may receive again an old copy of data item x from some other server Q due to still ongoing gossiping
- Solution: Introduce death certificates

Cache Coherence Protocols

Study of your own

Not examined

Cache-Coherence Protocols

- Cache = special replica
  - Often controlled by clients instead of servers
  - Multiple caches with more or less outdated data
- Two major design criteria
  - Coherence detection
  - Coherence implementation

Cache Coherence Detection

- *How and when can you detect that there are inconsistencies between the (primary) replica an one of the client caches*
  - A client cache can check the server periodically (or when its TTS has expired) whether the cached data is still valid
  - Check during an access, e.g. within transactions with rollback
  - Checks after an access (e.g. transactions), i.e. before committing a transaction. In case of inconsistency just roll back the transaction

Cache-Coherence Approaches

- Cache = special replica
  - Centralized primary replica
  - Multiple caches with more or less outdated data
- Two major design criteria
  - Coherence detection
  - Coherence implementation

Cache Coherence Detection

- Consistency checks, i.e. check whether cached data are still consistent
  - Check before a new access
  - Check during an access, e.g. within transactions with rollback
  - Checks after an access (e.g. transactions)

Cache Coherence Implementation

- No replicas of shared data
- Invalidation
  - Write access invalidates all cached entries
- Cache updates
  - Write access updates cached entries
    - Via snooping or primary copy

Cache Enforcement Policy

1. No Caching of shared data. Shared data are only kept at the primary servers, which maintain consistency using one of the primary-based replication protocols

2. If caching of shared data is allowed
    1. Invalidation notifications from the server to all caches whenever a data item is updated
    2. Propagate the update

Cache Enforcement Policy

- *What to do when a process updates a cached data?*
  - In case of read-only caches the update operation is written to the responsible server, which has to propagate it to all replicas to some propagation rule
  - In many cases a pull-based approach is used, i.e. a cache detects that its data is stale and requests the server for an update
  - In case of a read/ write cache the process directly update that data item x and forwards this update to its server (immediately or lazily)
    - Write-through or write-back caches

Implementing Client-Centric Consistency

Naive Implementation

- Each write operation gets a globally unique identifier
- For each site we keep 2 sets or writes
  - Read set consists of all writes relevant for the read operation performed by a client; per write you also add where this write has taken place
  - Write sets consists of all writes performed by the client

Monotonic Read

- When client wants to read from a server, it compares its own read set with the write set of the server
- If the server is not up to date, it first has to pull all missing writes before handling the local read
- Alternatively the read is only forwarded to a sever that has already done all client's writes
- Similarly, you can implement the other three client-centric consistency protocols
- More efficient solution use vector time to eliminate the large read & write sets

Examples

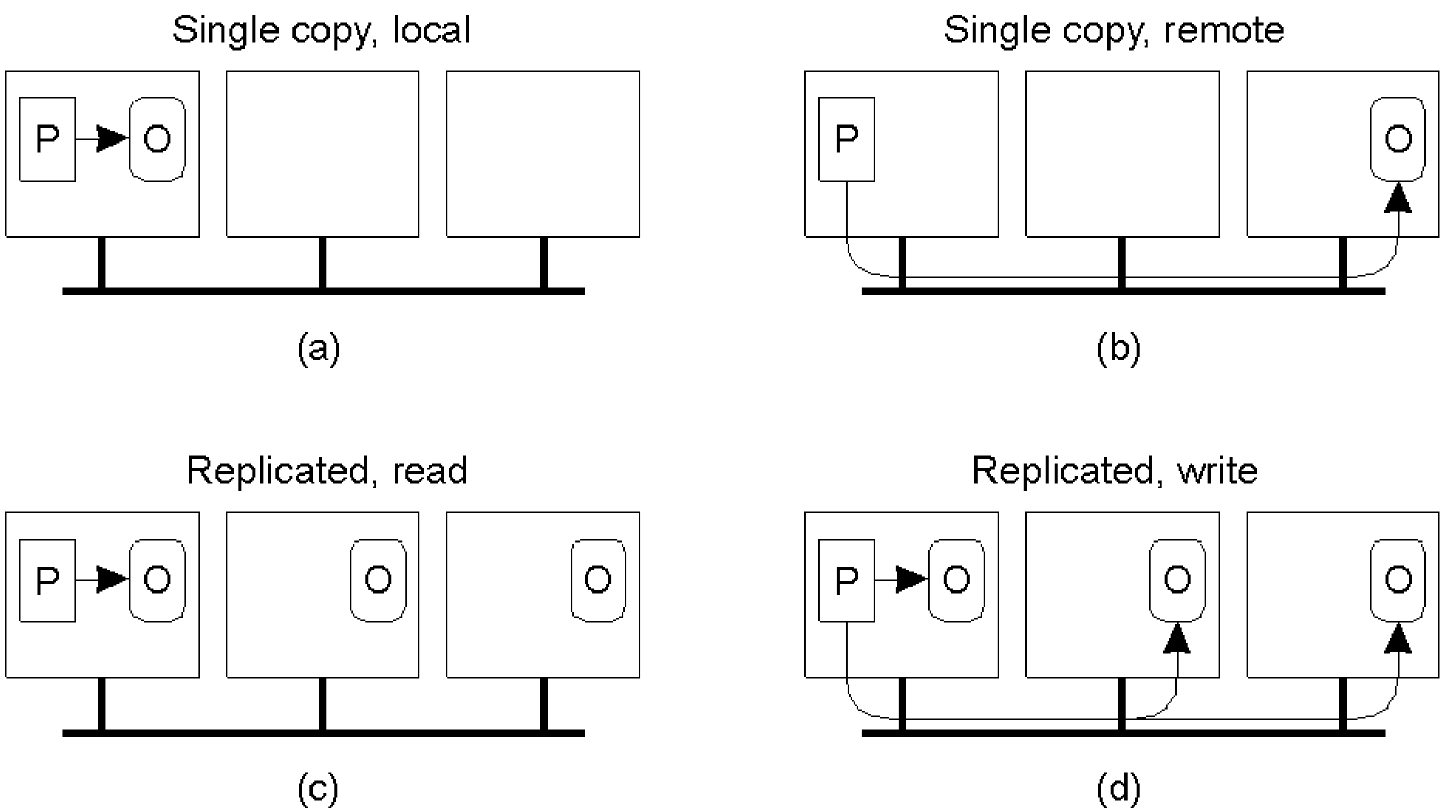
- Orca
  - Orca Language + Runtime System
  - Management of Shared Objects in Orca
- Causally-Consistent Lazy Replication
  - Processing Read Operations
  - Processing Write operations
  - Update Propagation

Orca

```
OBJECT IMPLEMENTATION stack;
  top: integer;                           # variable indicating the top
  stack: ARRAY[integer 0..N-1] OF integer # storage for the stack
  OPERATION push (item: integer)          # function returning nothing
  BEGIN
    GUARD top < N DO
        stack [top] := item;              # push item onto the stack
        top := top + 1;                   # increment the stack pointer
    OD;
  END;
  OPERATION  pop():integer;               # function returning an integer
  BEGIN
    GUARD top > 0 DO                      # suspend if the stack is empty
         top := top – 1;                  # decrement the stack pointer
        RETURN stack [top];               # return the top item
    OD;
  END;
BEGIN
  top := 0;                               # initialization
END;
```

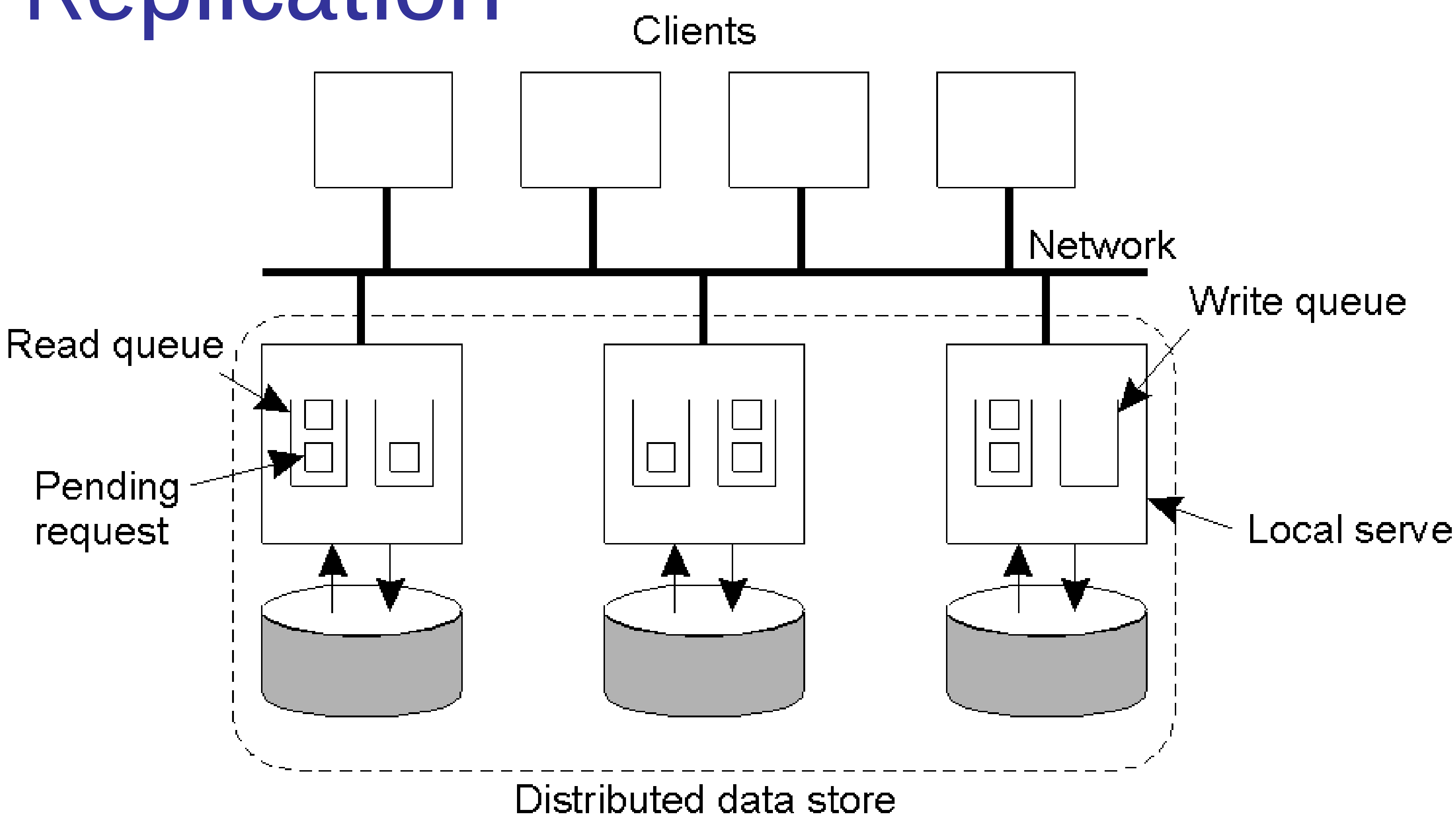
- Simplified stack object in Orca, with internal data and 2 operations.

Management of Shared Objects



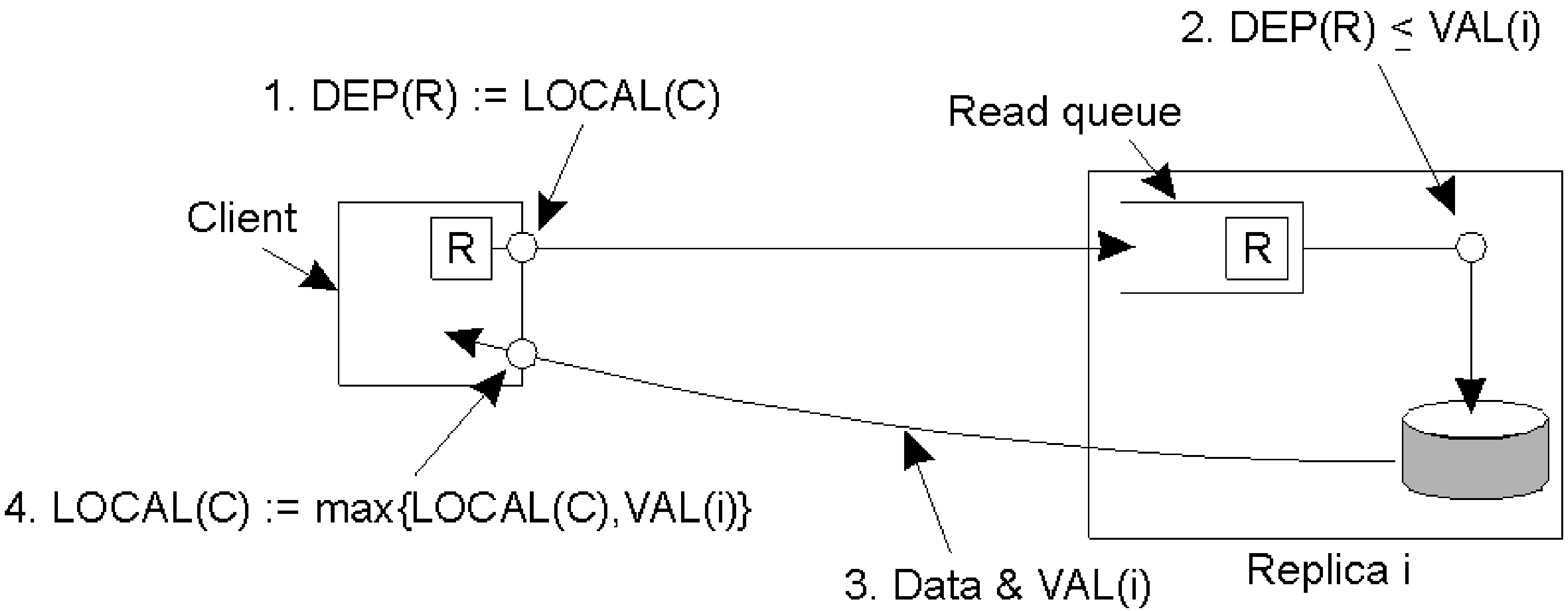
- 4 cases of a process *P* operating on an object *O* in Orca.

Causal-Consistent Lazy Replication



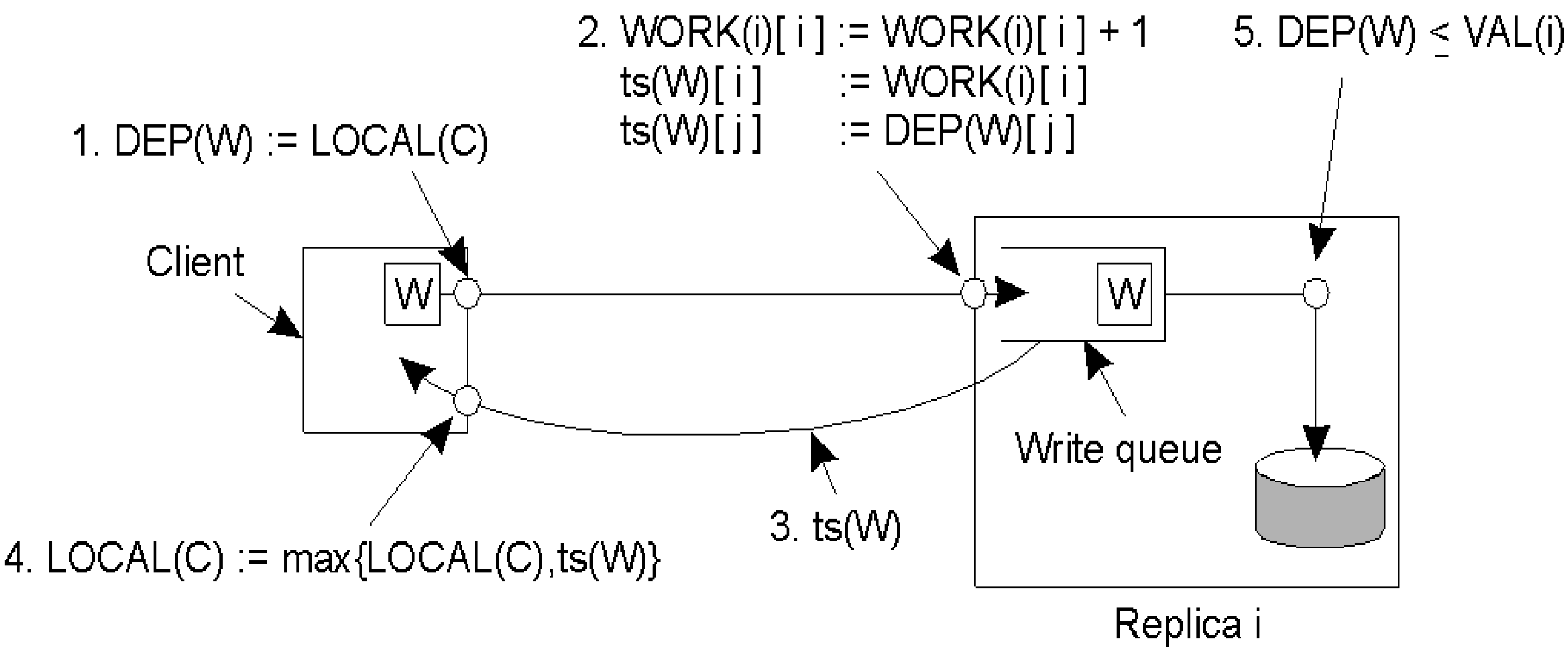
- General organization of a distributed data store.
  Clients also handle consistency-related communication.

Processing Read Operations



- Performing a read operation at a local copy.

Processing Write Operations



- Performing a write operation at a local copy.