



Chapter 6 - Consistency and Replication



Dr.T.GopiKrishna, Assistant Professor

Big Data, Machine Learning and data science Lab

Department of Computer Science and Engineering@SoEEC

Objectives of the Chapter

- **We discuss**
 - **Why replication is useful and its relation with scalability; in particular object-based replication**
 - **Consistency models**
 - **Data -Centric consistency Model**
 - **Client-Centric consistency models**
 - **How consistency and replication are implemented**

6.1 Reasons for Replication

- Two major reasons: **reliability** and **performance**

- ❖ **Reliability**

- if a file is replicated, we can switch to other replicas if there is a crash on our replica
- we can provide better protection against corrupted data; similar to **mirroring** in non-distributed systems

- ❖ **Performance**

- if the system has to scale in size and geographical area
place a copy of data in the proximity of the process using them, reducing the time of access and increasing its performance; for example a **Web server** is accessed by thousands of clients from all over the world

Replication as Scaling Technique

- **Replication and caching are widely applied as scaling techniques**
 - **processes can use local copies and limit access time and traffic. however, we need to keep the copies consistent; but this may**
 1. **Require more network bandwidth**
 - **if the copies are refreshed more often than used (low access-to-update ratio), the cost (bandwidth) is more expensive than the benefits;**

2. **Itself be subject to serious scalability**

problems

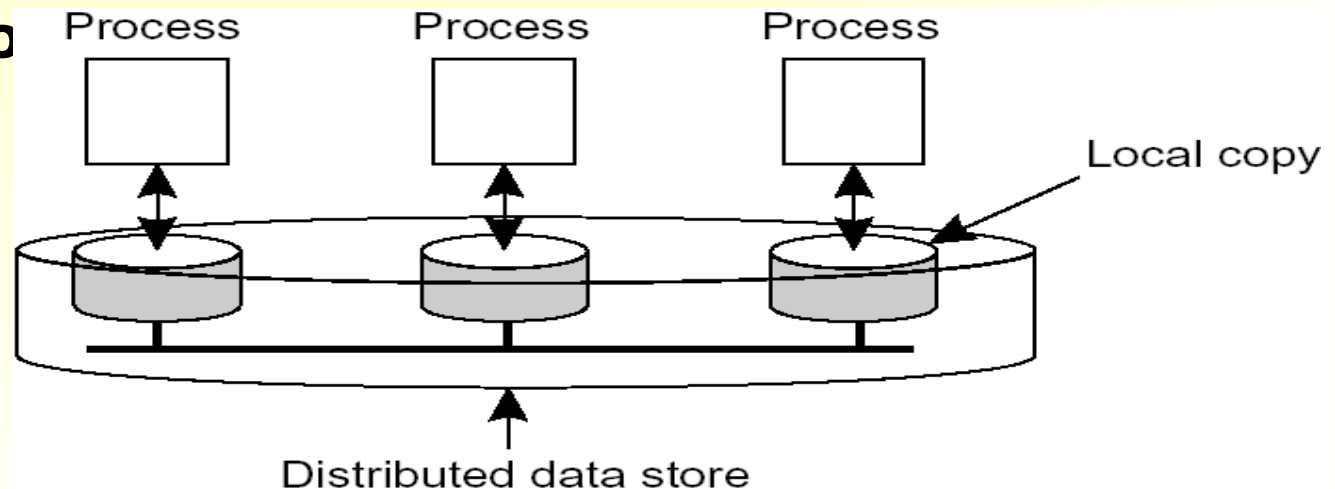
- **intuitively, a read operation made on any copy should return the same value (the copies are always the same)**
- **thus, when an update operation is performed on one copy, it should be propagated to all copies before a subsequent operation takes places**
- **this is sometimes called **tight consistency** (a write is performed at all copies in a single atomic operation or transaction)**
- **difficult to implement since it means that all replicas first need to reach agreement on when exactly an update is to be performed locally, say by deciding a global ordering of operations using **Lamport timestamps** and this takes a lot of communication time**

- **dilemma**

- **scalability problems can be alleviated by applying replication and caching, leading to a better performance**
- **but, keeping copies consistent requires global synchronization, which is generally costly in terms of performance**
- **solution: loosen the consistency constraints**
 - **updates do not need to be executed as atomic operations (no more instantaneous global synchronization); but copies may not be always the same everywhere**
 - **to what extent the consistency can be loosened depends on the specific application (the purpose of data as well as access and update patterns)**

6.2 Data-Centric Consistency Models

- Consistency has always been discussed
 - in terms of read and write operations on shared data available by means of (distributed) shared memory, a (distributed) shared database, or a (distributed) file system
 - we use the broader term **data store**, which may be physically distributed across multiple machines
 - assume also that each process has a local copy of the data store propagated



the general organization of a logical data store, physically distributed and replicated across multiple processes

- a **consistency model** is a contract between processes and the data store
 - processes agree to obey certain rules, then the data store promises to work correctly
- ideally, a process that reads a data item expects a value that shows the results of the last write operation on the data
- in a distributed system and in the absence of a global clock and with several copies, it is difficult to know which is the last write operation
- to simplify the implementation, each consistency model restricts what read operations return
- data-centric consistency models to be discussed
 1. strict consistency
 2. sequential consistency
 3. causal consistency
 4. weak consistency
 5. release consistency
 6. entry consistency

1. Strict Consistency

- the most stringent consistency model and is defined by the following condition:
Any read on a data item x returns a value corresponding to the result of the most recent write on x .
- this relies on absolute global time
- sometimes it is against nature
 - x is stored only on machine B
 - a process on machine A reads x at time $T1$, i.e., a message is sent to B
 - a process on machine B does a write on x at time $T2$ ($T1 < T2$)
 - if $T2 - T1$ is 1 nanosecond, and if the machines are 3 meters apart, the read request can reach B before the new write operation if the signal travels 10 times the speed of light
- ➔ the requirement is too stringent to demand

- the following notations and assumptions will be used
 - $W_i(x)_a$ means write by P_i to data item x with the value a has been done
 - $R_i(x)_b$ means a read by P_i to data item x returning the value b has been done
 - the index may be omitted when there is no confusion as to which process is accessing data
 - assume that initially each data item is *NIL*
- consider the following example; write operations

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

(a)

P1:	W(x)a	
P2:		R(x)NIL R(x)a

(b)

behavior of two processes operating on the same data item

- a) a strictly consistent data store*
- b) a data store that is not strictly consistent; P2's first read may be, for example, after 1 nanosecond of P1's write*
- the solution is to relax absolute time and consider time intervals

2. Sequential Consistency

- strict consistency is the ideal but impossible to implement
- fortunately, most programs do not need strict consistency
- sequential consistency is a slightly weaker consistency
- a data store is said to be sequentially consistent when it satisfies the following condition:
 - ***The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program***
- i.e., all processes see the same interleaving of operations
- time does not play a role; no reference to the “most recent” write operation

- example: four processes operating on the same data item x**

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

a sequentially consistent data store

- the write operation of P2 appears to have taken place before that of P1; but for all processes**

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

a data store that is not sequentially consistent

- to P3, it appears as if the data item has first been changed to b, and later to a; but P4, will conclude that the final value is b**
- not all processes see the same interleaving of write operations**

3. Causal Consistency

- it is a weakening of sequential consistency
- it distinguishes between events that are potentially causally related and those that are not
 - example: a write on y that follows a read on x; the writing of y may have depended on the value of x
- otherwise the two events are concurrent
 - two processes write two different variables
- if event B is caused or influenced by an earlier event, A, causality requires that everyone else must first see A, then B
- a data store is said to be causally consistent, if it obeys the following condition:
 - *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 on different machines.*

- **example**
 - **$W_2(x)b$ and $W_1(x)c$ are concurrent, not a requirement for processes to see them in the**

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

Diagram annotations: A red line labeled "CR" connects P1's W(x)a to P2's R(x)a. A red line labeled "Conc" connects P1's W(x)c to P2's W(x)b.

this sequence is allowed with a casually-consistent store, but not with sequentially or strictly consistent store

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

Diagram annotation: A red line labeled "CR" connects P1's W(x)a to P2's R(x)a.

(a)

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

Diagram annotation: A red line labeled "Conc" connects P1's W(x)a to P2's W(x)b.

(b)

- a) *a violation of a causally-consistent store*
- b) *a correct sequence of events in a causally-consistent store*

- **Models with synchronization operations**

4. Weak Consistency

- **FIFO consistency is still unnecessarily restrictive for many applications; it requires that writes originating in a single process be seen everywhere in order**
- **not all applications require even seeing all writes, let alone seeing them in order**
- **for example, there is no need to worry about intermediate results in a critical section since other processes will not see the data until it leaves the critical section; only the final result need to be seen by other processes**
- **this can be done by a **synchronization variable, S**, that has only a single associated operation **synchronize(S)**, which synchronizes all local copies of the data store**
- **a process performs operations only on its locally available copy of the store**
- **when the data store is synchronized, all local**

- this leads to **weak consistency models** which have three properties
 1. ***Accesses to synchronization variables associated with a data store are sequentially consistent*** (all processes see all operations on synchronization variables in the same order)
 2. ***No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere*** (synchronization flushes the pipeline: all partially completed - or in progress - writes are guaranteed to be completed when the synchronization is done)
 3. ***No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed*** (when a process accesses a data item (for reading or writing) all previous synchronization will have been completed; by

- **weak consistency enforces consistency on a group of operations, not on individual reads and writes**
- **e.g., S stands for synchronizes; it means that a local copy of a data store is brought up to date**

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)

- a) a valid sequence of events for weak consistency*
- b) an invalid sequence for weak consistency; P2 should get **b***

6. Entry Consistency

- like release consistency, it requires an acquire and release to be used at the start and end of a critical section
- however, it requires that each ordinary shared data item to be associated with some synchronization variable such as a lock
- if it is desired that elements of an array be accessed independently in parallel, then different array elements may be associated with different locks
- synchronization variable ownership
 - each synchronization variable has a **current owner**, the process that acquired it last
 - the owner may enter and exit critical sections repeatedly without sending messages
 - other processes must send a message to the current owner asking for ownership and the current values of the data associated with that synchronization variable

- a data store exhibits **entry consistency** if it meets all the following conditions:
 - *An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process. (at an acquire, all remote changes to the guarded data must be made visible)*
 - *Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.*
 - *After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner. (it must first fetch the most*

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)
P2:			Acq(Lx)	R(x)a		R(y)NIL
P3:				Acq(Ly)		R(y)b

*a valid event sequence for entry
consistency*

- **when an acquire is done only those variables guarded by that synchronization variable are made consistent**
- **therefore, a few shared data items have to be synchronized when there is a release**

Summary of Data-Centric Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses
Linearizability	All processes see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	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

(a)

Consistency	Description
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

(b)

a) consistency models not using synchronization operations

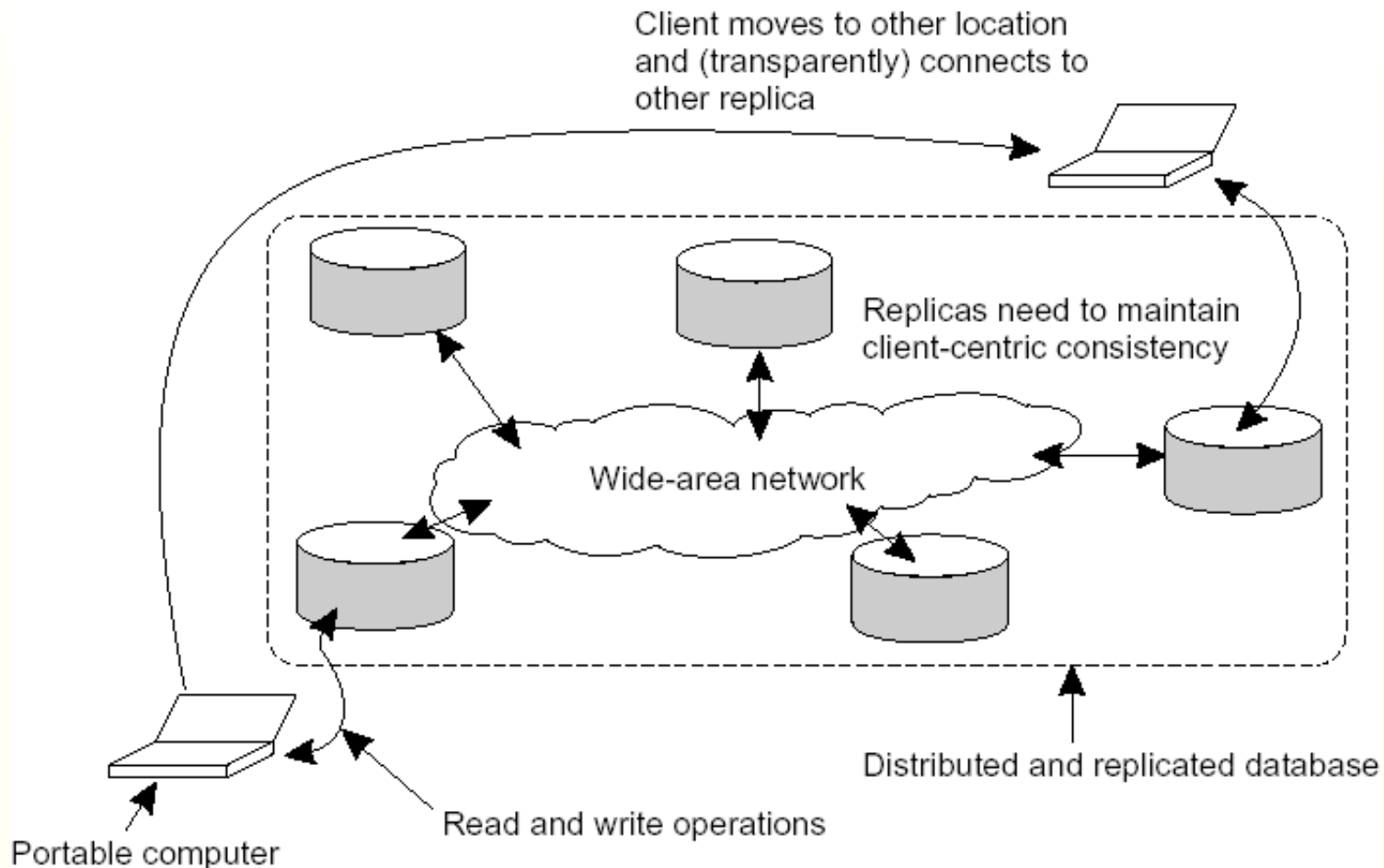
b) models with synchronization operations

- consistency models differ
 - in complexity of implementation
 - ease of programming
 - performance
- **strict consistency**: most restrictive; never implemented, implementation in a distributed system is impossible
- **linearizability**: hardly ever used; but facilitates reasoning about the correctness of parallel programs
- **sequential consistency**: widely used, but poor performance; so relax conditions by having **causal consistency** and **FIFO consistency**
- **weak consistency**, **release consistency**, and **entry consistency**: require additional programming constructs; allow programmers to pretend that a data store is sequentially consistent when in fact it is not; may provide the best performance

6.3 Client-Centric Consistency Models

- with many applications, updates happen very rarely
- for these applications, data-centric models where high importance is given for updates are not suitable
- very weak consistency is generally sufficient for such systems
- **Eventual Consistency**
 - there are many applications where few processes (or a single process) update the data while many read it and there are no **write-write** conflicts; we need to handle only **read-write** conflicts; e.g., DNS server, Web site
 - for such applications, it is even acceptable for readers to see old versions of the data (e.g., cached versions of a Web page) until the new version is propagated
 - with eventual consistency, it is only required

- **data stores that are eventually consistent have the property that in the absence of updates, all replicas converge toward identical copies of each other**
- **write-write conflicts are rare and are implemented separately**
- **the problem with eventual consistency is when different replicas are accessed, e.g., a mobile client accessing a distributed database may acquire an older version of data when it uses a new replica as a result of changing location**



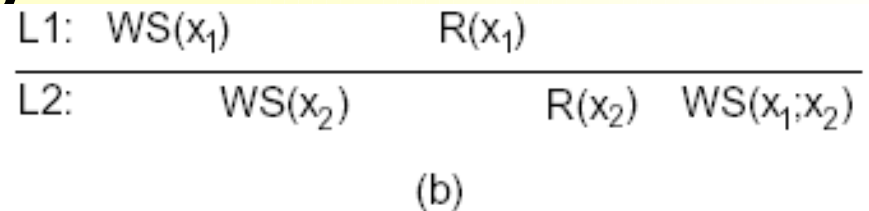
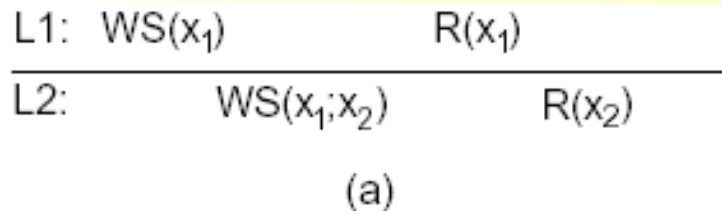
the principle of a mobile user accessing different replicas of a distributed database

- the solution is to introduce **client-centric consistency**
- it provides guarantees for a **single client** concerning the consistency of accesses to a data store by that client; no guarantees are given

- there are four client-centric consistency models
- consider a data store that is physically distributed across multiple machines
- a process reads and writes to a locally available copy and updates are propagated
- assume that data items have an associated owner, the only process permitted to modify that item, hence write-write conflicts are avoided
- the following notations are used
 - $x_i[t]$ denotes the version of the data item x at local copy L_i at time t
 - version $x_i[t]$ is the result of a **series** of write operations at L_i that took place since initialization; denote this set by **$WS(x_i[t])$**
 - if operations in $WS(x_i[t_1])$ have also been performed at local copy L_j at a later time t_2 , we write **$WS(x_i[t_1]; x_j[t_2])$** ; it means that $WS(x_i[t_1])$ is part of $WS(x_j[t_2])$
 - the time index may be omitted if ordering of

1. Monotonic Reads

- a data store is said to provide **monotonic-read consistency** if the following condition holds:
 - *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 value or a more recent value*
- **i.e., a process never sees a version of data older than what it has already seen**



the read operations performed by a single process P at two different local copies of the same data store

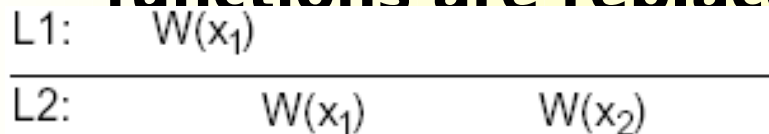
a) a monotonic-read consistent data store

b) a data store that does not provide monotonic reads; there is no guaranty that when $R(x_2)$ is executed $WS(x_2)$ also contains $WS(x_1)$

2. Monotonic Writes

- it may be required that write operations propagate in the correct order to all copies of the data store
- in a **monotonic-write consistent** data store the following condition holds:
 - *A write operation by a process on a data item x is completed before any successive write operation on x by the same process*
- completing a write operation means that the copy on which a successive operation is performed reflects the effect of a previous write operation by the same process, no matter where that operation was initiated
- monotonic-write consistency resembles data-centric FIFO consistency; here we consider consistency only for a single process (instead of for a collection of concurrent processes)

- **may not be necessary if a later write operation completely overwrites the present**
- $x = 78;$**
- $x = 90;$**
- **no need to make sure that x has been first changed to 78**
- **it is important only if part of the state of the data item changes**
- **e.g., a software library, where one or more functions are replaced. leading to a new version**



(a)



(b)

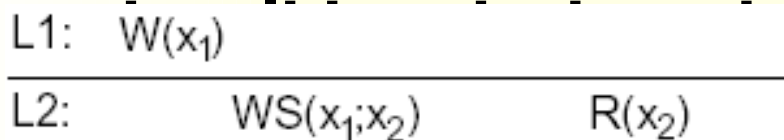
the write operations performed by a single process P at two different local copies of the same data store

a) a monotonic-write consistent data store

b) a data store that does not provide monotonic-write consistency

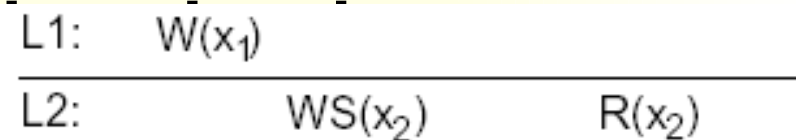
3. Read Your Writes

- a data store is said to provide **read-your-writes consistency**, if the following condition holds:
 - *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*
- i.e., a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place
- the absence of read-your-writes consistency is often experienced when a Web page is modified using an editor and the modification is not seen on the browser due to caching; read-your-writes consistency guarantees that the cache is



(a)

a) a data store that provides read-your-writes consistency



(b)

4. **Writes Follow Reads**

- updates are propagated as the result of previous read operations
- a data store is said to provide **writes-follow-reads consistency**, if the following condition holds:
 - *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*
- i.e., any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process
- this guaranties, for example, that users of a newsgroup see a posting of a reaction to an article only after they have seen the original article; if B is a response to message A, writes-follow-reads consistency guarantees that B will be²¹

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)

- a) a writes-follow-reads consistent data store
- b) a data store that does not provide writes-follow-reads consistency

Implementation of Client-Centric Consistency

- a naive implementation

- each **write** operation is given a globally unique identifier, assigned by the server that accepts the operation for the first time
- then for each **client**, keep track of two sets of identifiers:
 - the **read set** consists of the **write identifiers** relevant for the read operations performed by a client
 - the **write set** consists of the **identifiers** of the **writes** performed by the client

- **monotonic-read consistency** is implemented as follows
 - when a client performs a **read** operation at a server, the server is handed the client's **read set** to check if all the identified writes have taken place locally
 - if not, the server contacts the other servers to ensure that it is brought up to date before carrying out the read operation (or the read operation is forwarded to a server where the write operations took place)
 - after the read operation, the relevant write operations that have taken place at the selected servers are added to the client's **read set**
- **monotonic-write consistency** is implemented as follows
 - when a client initiates a new **write** operation to a server, the server is handed the client's **write set**
 - it then ensures that the identified write

- **read-your-writes consistency** is implemented as follows
 - it requires that the server where the read operation is performed has seen all the write operations in the client's **write set**
 - the writes can be fetched from the other servers before the read operation is performed (may result with a poor response time)
 - alternatively, the client-side software can search for a server where the identified write operations in the client's write set have already been performed
- **writes-follow-reads consistency** is implemented as follows
 - first bring the selected server up to date with the write operations in the client's **read set**
 - then add the identifier of the write operation to the **write set**, along with the identifiers in the **read set** (which have now become relevant for the write

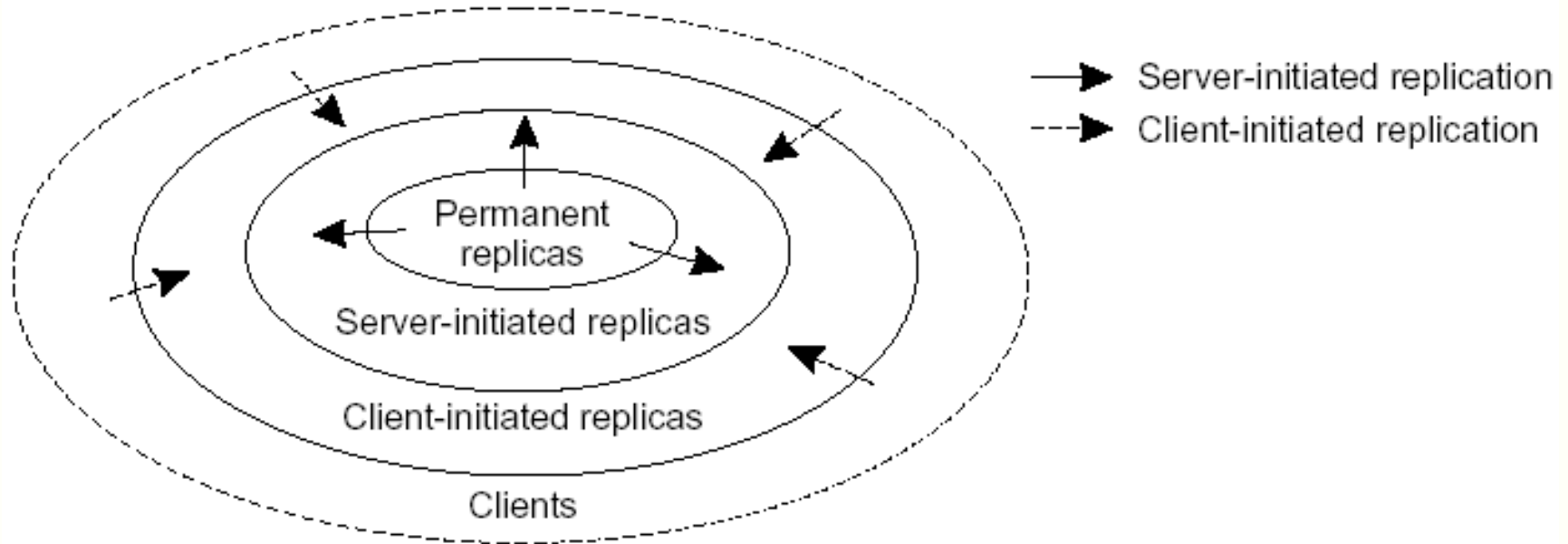
- **problem: in naive implementation, the read and write sets can become very large**
- **to improve efficiency, read and write operations can be grouped into sessions, clearing the sets when the session ends**

6.4 Distribution Protocols

- there are different ways of propagating, i.e., distributing updates to replicas, independent of the consistency model
- we will discuss
 - replica placement
 - update propagation
 - epidemic protocols

a. Replica Placement

- a major design issue for distributed data stores is deciding **where**, **when**, and by **whom** copies of the data store are to be placed
- three types of copies:
 - permanent replicas
 - server-initiated replicas
 - client-initiated replicas



the logical organization of different kinds of copies of a data store into three concentric rings

1. Permanent Replicas

- the initial set of replicas that constitute a distributed data store; normally a small number of replicas
- e.g., a Web site: two forms
 - the files that constitute a site are replicated across a limited number of servers on a LAN; a request is forwarded to one of the servers
 - **mirroring**: a Web site is copied to a limited number of servers, called **mirror sites**, which are geographically spread across the Internet; clients choose one of the mirror sites

2. Server-Initiated Replicas (push caches)

- Web Hosting companies dynamically create replicas to improve performance (e.g., create

3. Client-Initiated Replicas (client caches or simply caches)

- **to improve access time**
- **a cache is a local storage facility used by a client to temporarily store a copy of the data it has just received**
- **placed on the same machine as its client or on a machine shared by clients on a LAN**
- **managing the cache is left entirely to the client; the data store from which the data have been fetched has nothing to do with keeping cached data consistent**

b. Update Propagation

- updates are initiated at a client, forwarded to one of the copies, and propagated to the replicas ensuring consistency
- some design issues in propagating updates
 - **state versus operations**
 - **pull versus push protocols**
 - **unicasting versus multicasting**

1. State versus Operations

- what is actually to be propagated? three possibilities
 - send notification of update only (for **invalidation protocols** - useful when read/write ratio is small); use of little bandwidth
 - transfer the modified data (useful when read/write ratio is high)
 - transfer the update operation (also called **active replication**); it assumes that each machine knows how to do the operation; use of little bandwidth, but more processing power

2. Pull versus Push Protocols

- **push-based approach** (also called **server-based protocols**): propagate updates to other replicas without those replicas even asking for the updates (used when high degree of consistency is required and there is a high read/write ratio)
- **pull-based approach** (also called **client-based protocols**): often used by client caches; a client or a server requests for updates from the server whenever needed (used when the read/write ratio is low)
- a comparison between push-based and pull-based protocols; for simplicity assume multiple clients and a single server

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

3. Unicasting versus Multicasting

- multicasting can be combined with push-based approach; the underlying network takes care of sending a message to multiple receivers
- unicasting is the only possibility for pull-based approach; the server sends separate messages to each receiver

c. Epidemic Protocols

- update propagation in **eventual consistency** is often implemented by a class of algorithms known as **epidemic protocols**
- updates are aggregated into a single message and then exchanged between two servers

6.5 Consistency Protocols

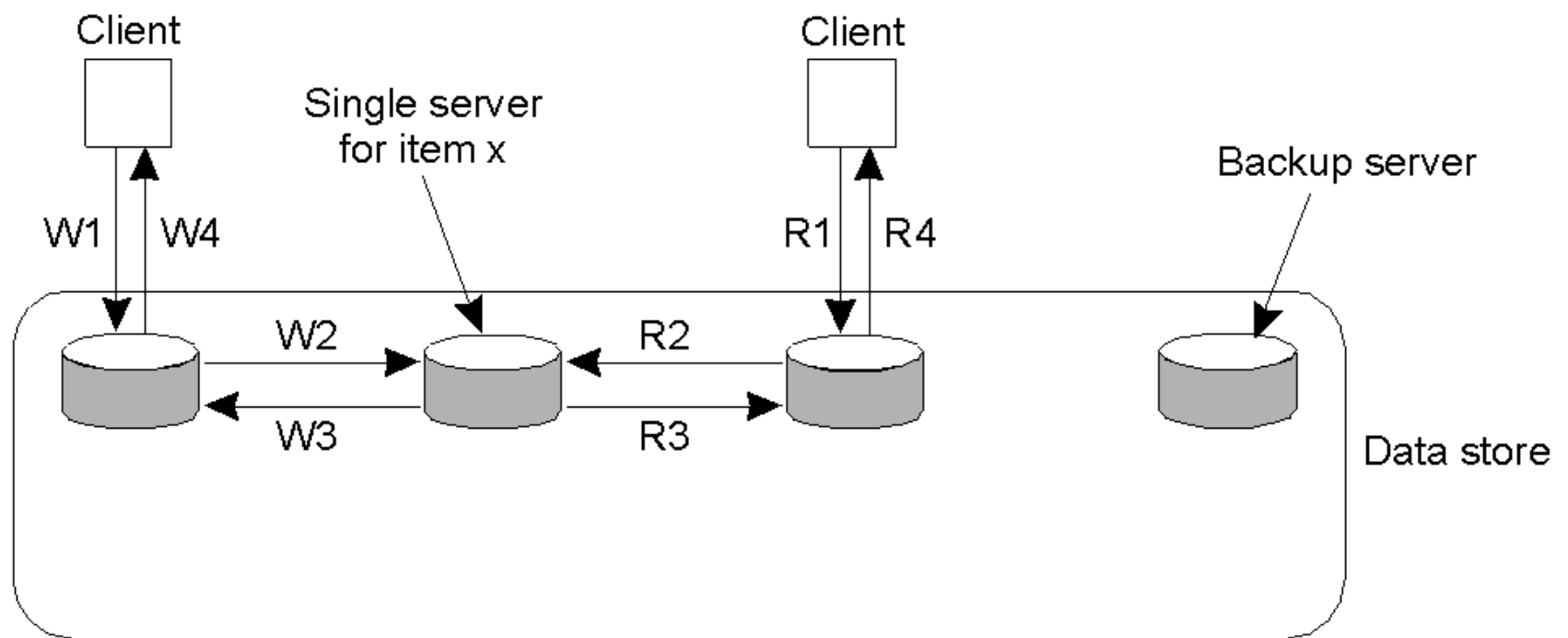
- so far we have concentrated on various consistency models and general design issues
- **consistency protocols** describe an implementation of a specific consistency model
- there are three types
 - **primary-based protocols**
 - remote-write protocols
 - local-write protocols
 - **replicated-write protocols**
 - active replication
 - quorum-based protocols
 - **cache-coherence protocols**

1. Primary-Based Protocols

- each data item x in the data store has an associated primary, which is responsible for coordinating write operations on x
- two approaches: **remote-write protocols**, and **local-write protocols**

a. **Remote-Write Protocols**

- all read and write operations are carried out at a (**remote**) single server; in effect, data are not replicated; traditionally used in client-server systems, where the server may possibly be distributed



W1. Write request

W2. Forward request to server for x

W3. Acknowledge write completed

W4. Acknowledge write completed

R1. Read request

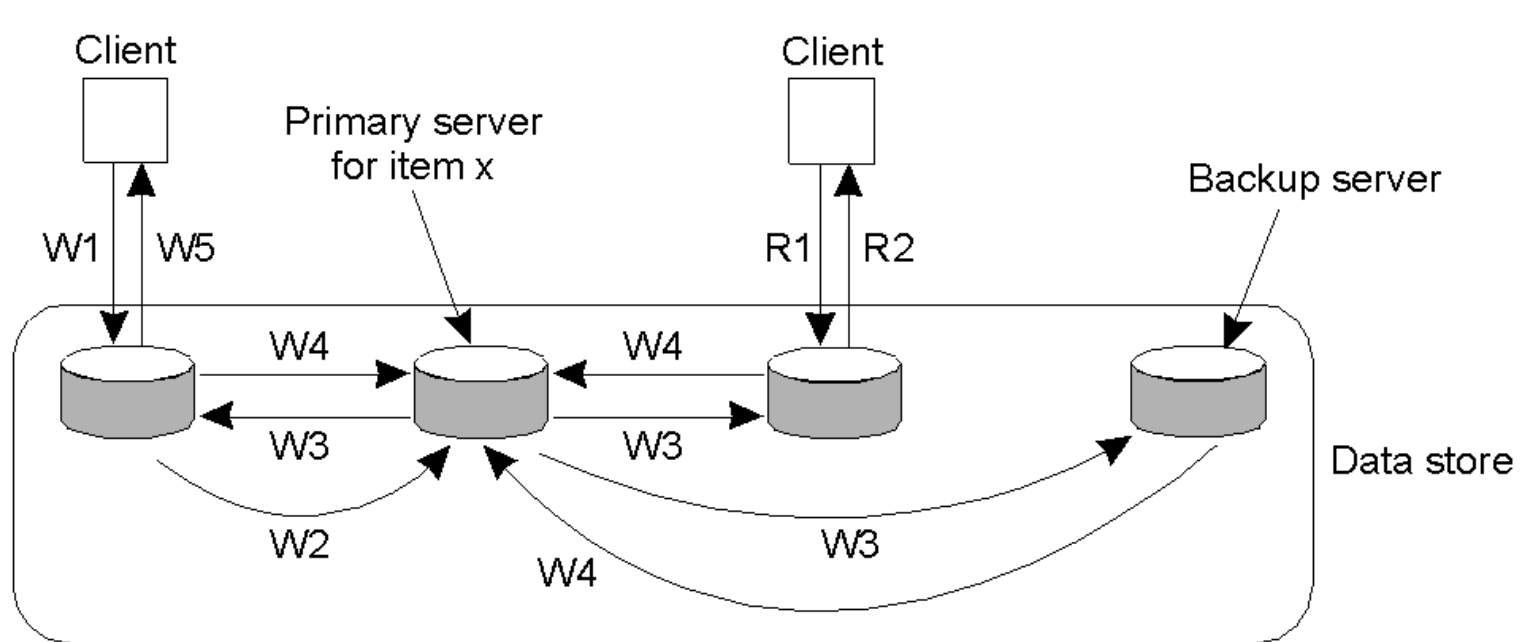
R2. Forward request to server for x

R3. Return response

R4. Return response

primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded

- another approach is **primary-backup protocols** where reads can be made from local **backup servers** while writes should be made directly on the **primary server**
- the backup servers are updated each time the



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

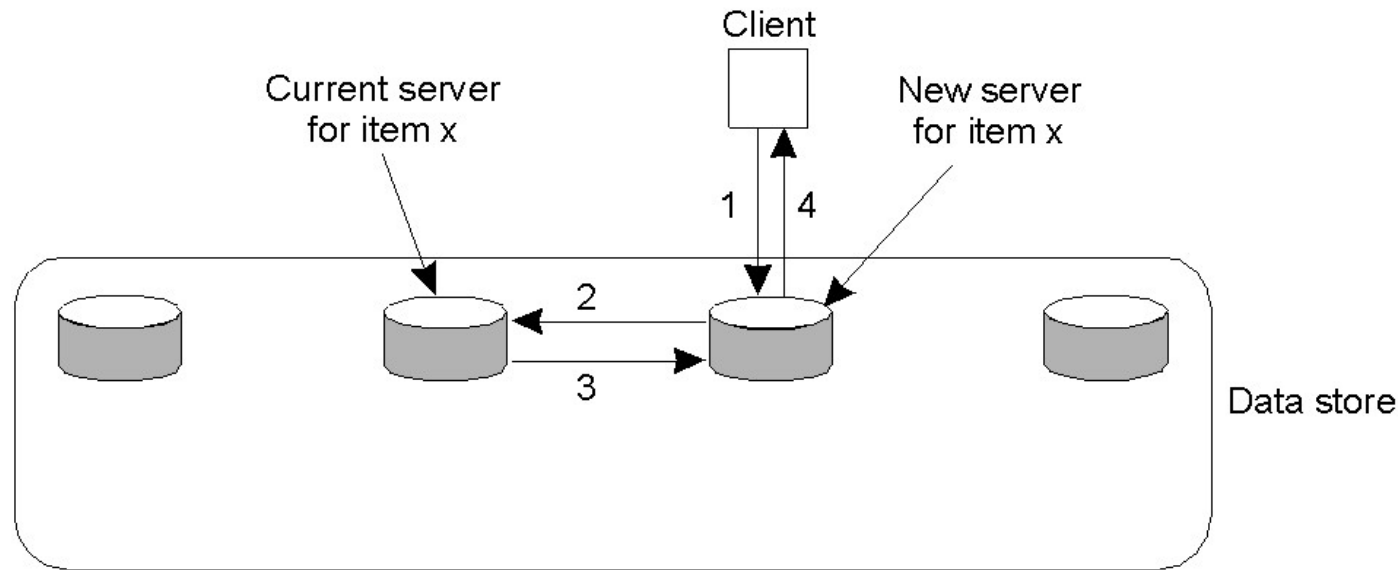
R1. Read request
 R2. Response to read

the principle of primary-backup protocol

- may lead to performance problems since it may take time before the process that initiated the write operation is allowed to continue - updates are blocking
- primary-backup protocols provide straightforward implementation of **sequential consistency**; the **primary** can order all incoming writes

b. Local-Write Protocols

- two approaches
- i. there is a single copy; no replicas**
 - when a process wants to perform an operation on some data item, the single copy of the data item is transferred to the process, after which the operation is performed



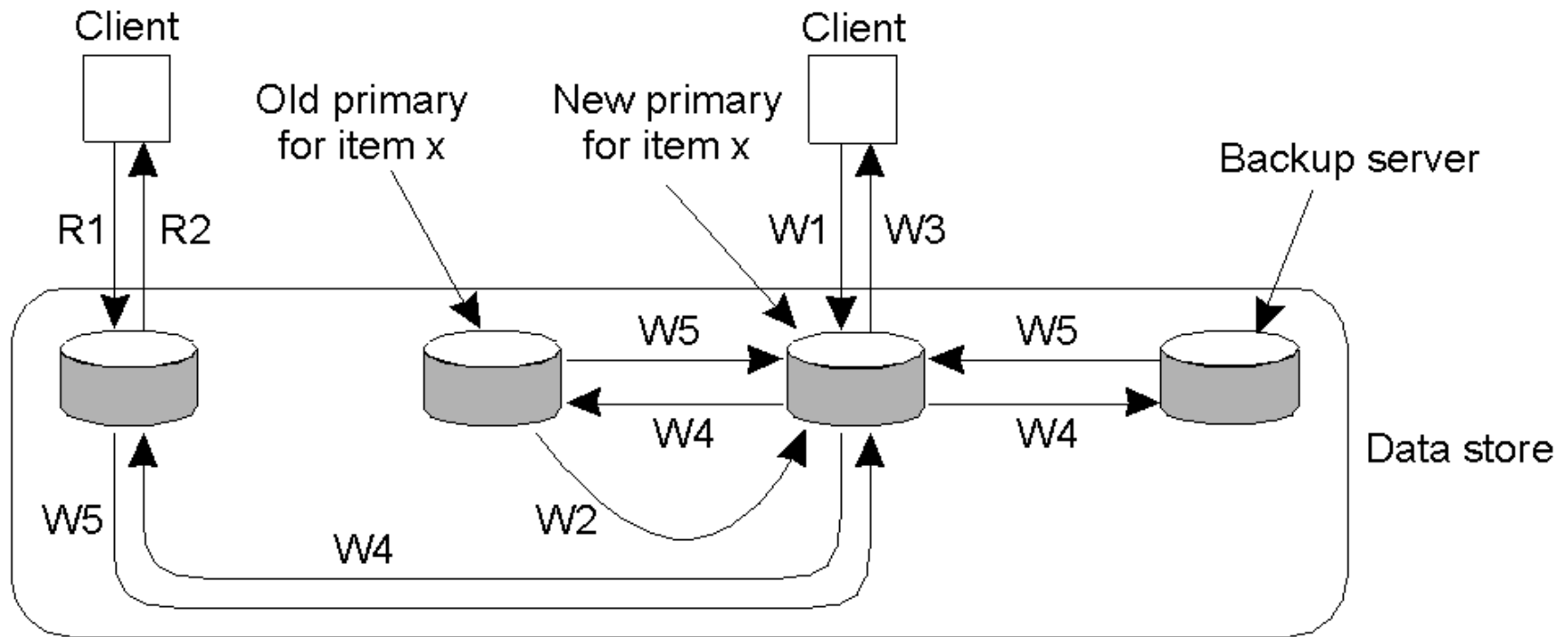
1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

primary-based local-write protocol in which a single copy is migrated between processes

- **consistency is straight forward**
- **keeping track of the current location of each data item is a major problem**

ii. primary-backup local-write protocol

- **the primary migrates between processes that wish to perform a write operation**
- **multiple, successive write operations can be carried out locally, while (other) reading processes can still access their local copy**
- **such improvement is possible only if a nonblocking protocol is followed**



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 migrates to the process wanting to perform an update

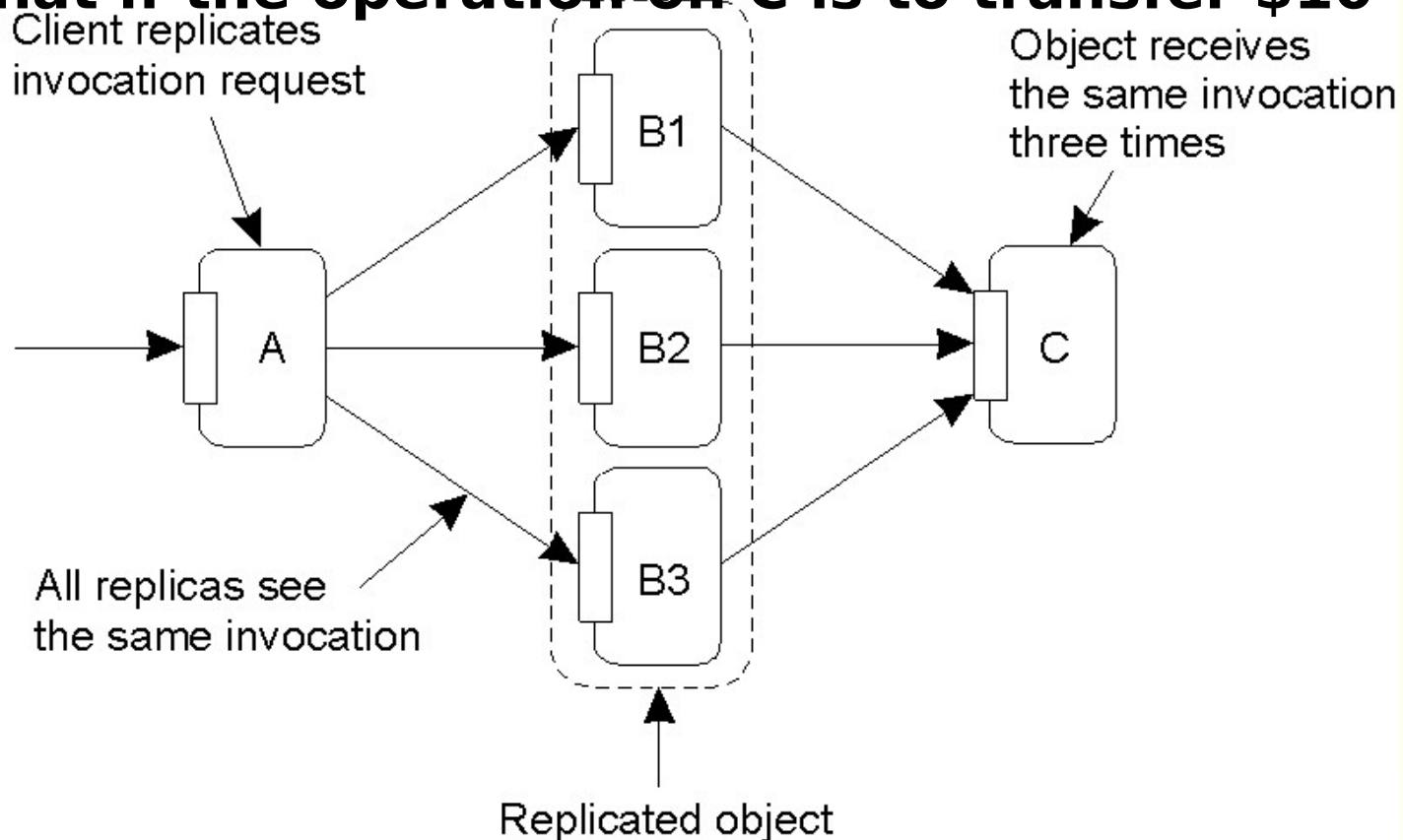
2. Replicated-Write Protocols

- unlike primary-based protocols, write operations can be carried out at multiple replicas; two approaches: **Active Replication** and **Quorum-Based Protocols**

a. **Active Replication**

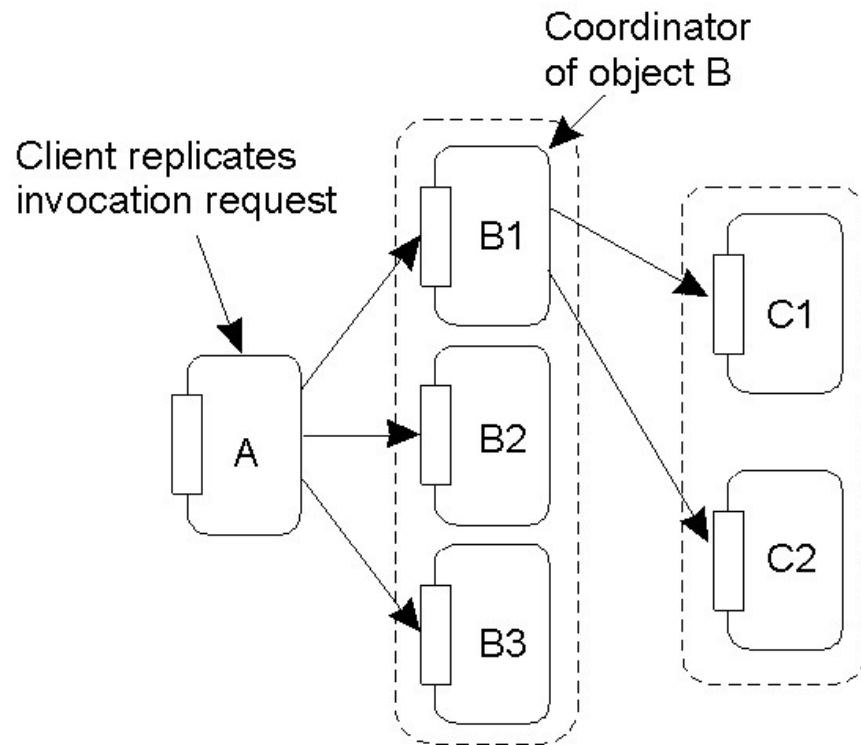
- each replica has an associated process that carries out update operations
- updates are generally propagated by means of write operations (the operation is propagated); also possible to send the update
- the operations need to be done in the same order everywhere; **totally-ordered multicast**
- two possibilities to ensure that the order is followed
 - Lamport's timestamps, or
 - use of a central **sequencer** that assigns a unique sequence number for each operation; the operation is first sent to the sequencer

- a problem is replicated invocations
 - suppose object A invokes B, and B invokes C; if object B is replicated, each replica of B will invoke C independently
 - this may create inconsistency and other effects; what if the operation on C is to transfer \$10

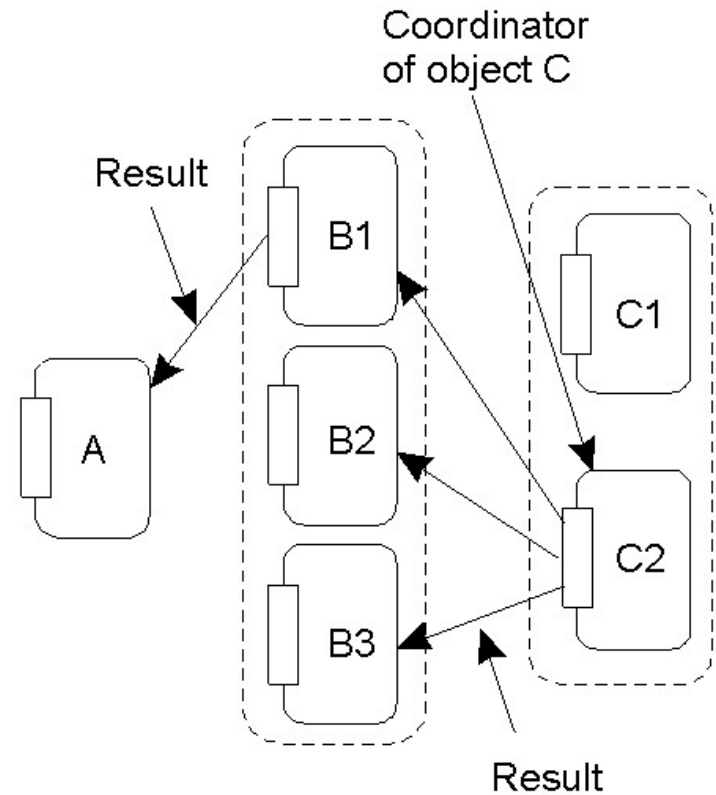


the problem of replicated invocations

- **one solution is to have a replication-aware communication layer that avoids the same invocation being sent more than once**
- **when a replicated object B invokes another replicated object C, the invocation request is first assigned the same, unique identifier by each replica of B**
- **a coordinator of the replicas of B forwards its request to all replicas of object C; the other replicas of object B hold back; hence only a single request is sent to each replica of C**
- **the same mechanism is used to ensure that only a single reply message is returned to the replicas of B**



(a)



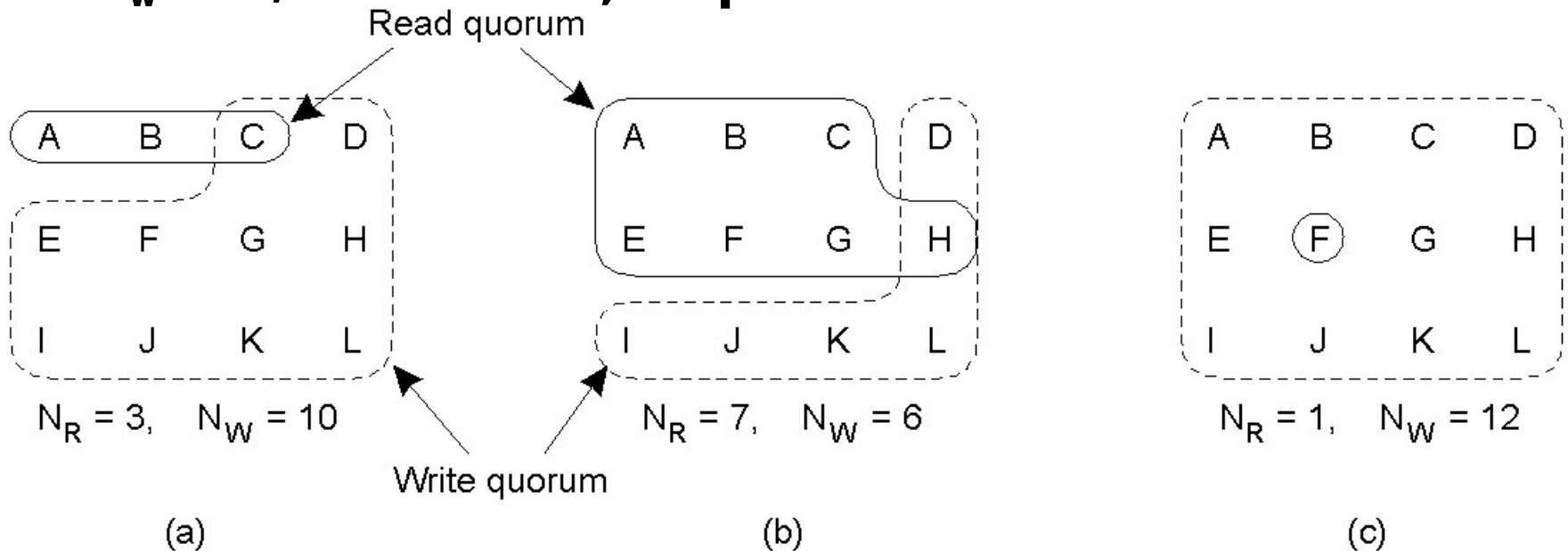
(b)

- a) forwarding an invocation request from a replicated object*
- b) returning a reply to a replicated object*

b. Quorum-Based Protocols

- use of **voting**: clients are required to request and acquire the permission of multiple servers before either reading or writing a replicated data item
- e.g., assume a distributed file system where a file is replicated on N servers
 - a client must first contact at least $\text{half} + 1$ (majority) servers and get them to agree to do an **update**
 - the new update will be done and the file will be given a new version number
 - to **read** a file, a client must also first contact at least $\text{half} + 1$ and ask them to send version numbers; if all version numbers agree, this must be the most recent version
 - a more general approach is to arrange a **read quorum** (a collection of any N_R servers, or more) for reading and a **write quorum** (of at least N_W servers) for updating

- the values of N_R and N_W are subject to the following two constraints
 - $N_R + N_W > N$; to prevent read-write conflicts
 - $N_W > N/2$; to prevent write-write conflicts



three examples of the voting algorithm ($N = 12$)

- a correct choice of read and write sets; any subsequent read quorum of three servers will have to contain at least one member of the write set which has a higher version number
- a choice that may lead to write-write conflicts; if a client chooses $\{A, B, C, E, F, G\}$ as its write set and another client chooses $\{D, H, I, J, K, L\}$ as its write set, the two updates will both be accepted without detecting that they actually conflict

3. Cache-Coherence Protocols

- caches form a special case of replication as they are controlled by clients instead of servers
- cache-coherence protocols ensure that a cache is consistent with the server-initiated replicas
- two design issues in implementing caches:
coherence detection and **coherence enforcement**
 - **coherence detection strategy**: **when** inconsistencies are actually detected
 - **static** solution: prior to execution, a compiler performs the analysis to determine which data may lead to inconsistencies if cached and inserts instructions that avoid inconsistencies
 - **dynamic** solution: at runtime, a check is made with the server to see whether a cached data have been modified since they were cached

- **coherence enforcement strategy: how** caches are kept consistent with the copies stored at the servers
 - simplest solution: do not allow shared data to be cached; suffers from performance improvement
 - allow caching shared data and
 - let a server send an invalidation to all caches whenever a data item is modified or
 - propagate the update

