CS865 – Distributed Software Development
Lecture 7

Tannenbaum and Van Steen – Chapter 7

Consistency and Replication

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

Introduction

Reasons for Replication

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 conflict**: 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

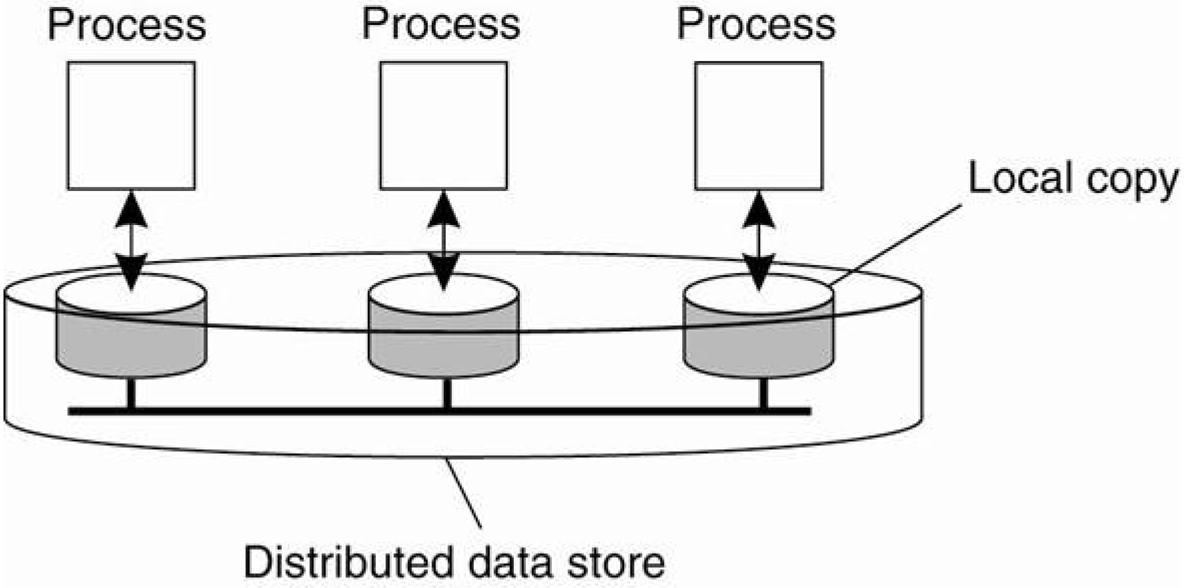
Different approaches to classifying consistency and replication can be found in Gray et al. (1996) and Wiesmann et al. (2000).

Data-Centric Consistency Models

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:

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



Continuous Consistency

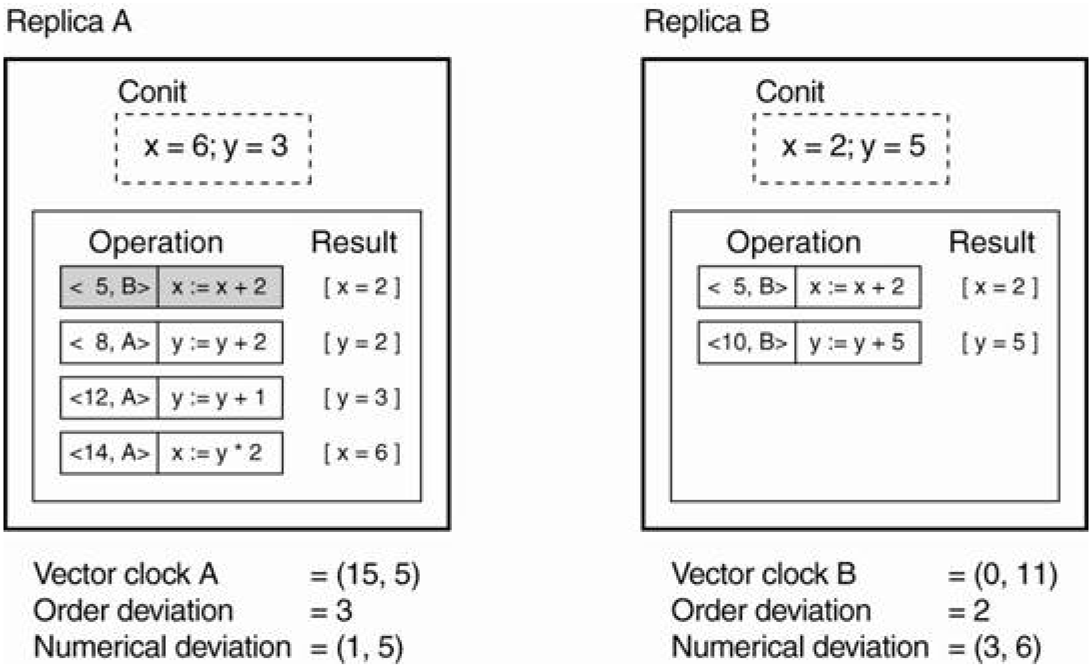
Observation: We can actually talk a about a **degree of consistency**: Yu and Vahdat (2002)

- replicas may differ in their **numerical value**
- replicas may differ in their relative **staleness**
- there may differences with respect to (number and order) of **performed update operations**

Conit: conistency unit) specifies the **data unit** over which consistency is to be measured.

- e.g. stock record, weather report, etc.

Conit example: numerical and ordering deviations



Operation		Result
< 5, B>	x := x + 2	[x = 2]
< 8, A>	y := y + 2	[y = 2]
<12, A>	y := y + 1	[y = 3]
<14, A>	x := y * 2	[x = 6]

Vector clock A = (15, 5)
Order deviation = 3
Numerical deviation = (1, 5)

Operation		Result
< 5, B>	x := x + 2	[x = 2]
<10, B>	y := y + 5	[y = 5]

Vector clock B = (0, 11)
Order deviation = 2
Numerical deviation = (3, 6)

Conit: contains the variables *x* and *y*:

- Each replica maintains a **vector clock**
- *B* sends *A* operation [**h**5,*B***i**: $x := x + 2$];
  A has made this operation **permanent** (cannot be rolled back)

- *A* has three **pending** operations) order deviation = 3
- *A* has missed **one** operation from *B*, yielding a max diff of 5 units) (1,5)

Strict Consistency

Any read on a data item 'x' returns a value corresponding to the result of the most recent write on 'x' (regardless of where the write occurred).

- With *Strict Consistency*, all writes are *instantaneously visible* to all processes and *absolute global time order* is maintained throughout the distributed system.

- This is the consistency model "Holy Grail" – not at all easy in the real world, and all but *impossible* within a DS.

Example:

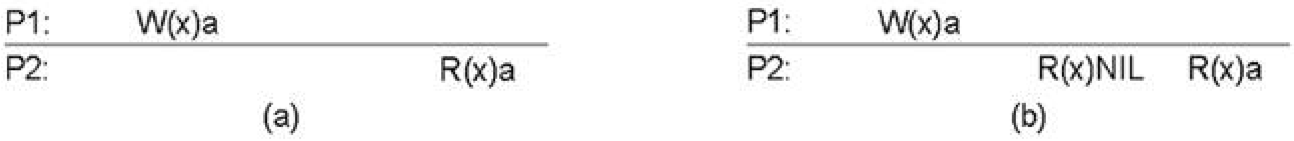
- Begin with:
  Wi(x)a –write by process Pi to data item x with the value a
  Ri(x)b - read by process Pi from data item x returning the value b
- Assume:
  - Time axis is drawn horizontally, with time increasing from left to right
  - Each data item is initially NIL.

P1 does a write to a data item x, modifying its value to a.

- Operation W1(x)a is first performed on a copy of the data store that is local to P1, and is then propagated to the other local copies.

P2 later reads the value NIL, and some time after that reads a (from its local copy of the store).

NOTE: it took some time to propagate the update of x to P2, which is perfectly acceptable.



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.

Sequential Consistency

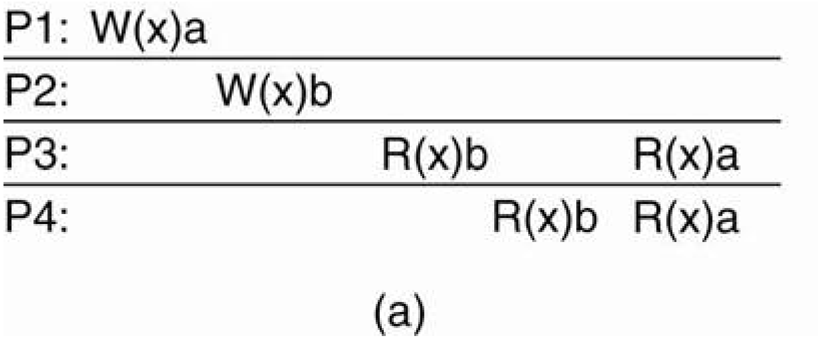
- A weaker consistency model, which represents a relaxation of the rules.
- It is also must easier (possible) to implement.

Sequential Consistency:
The result of any execution is the same as if the (read and write) operations by all proceses on the data-store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

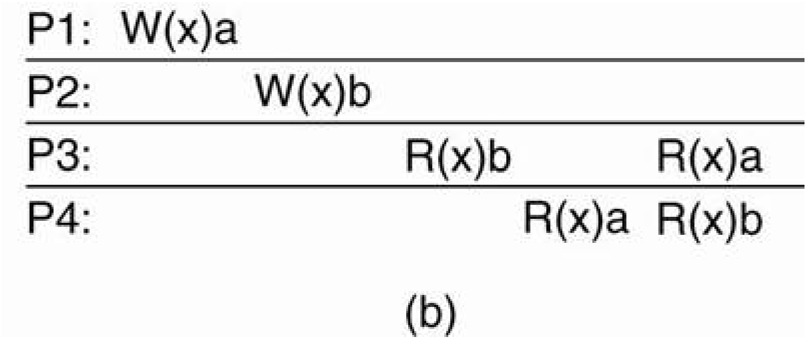
Example: Time independent process
Four processes operating on the same data item x.

Figure(a)	Figure(b)
• Process P1 first performs W(x)a to x. • Later (in absolute time), process P2 performs a write operation, by setting the value of x to b. • Both P3 and P4 first read value b, and later value a. • Write operation of process P2 appears to have taken place before that of P1.	• Violates sequential consistency - not all processes see the same interleaving of write operations. • To process 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.

(a) A sequentially consistent data store. (b) A data store that is not sequentially consistent.



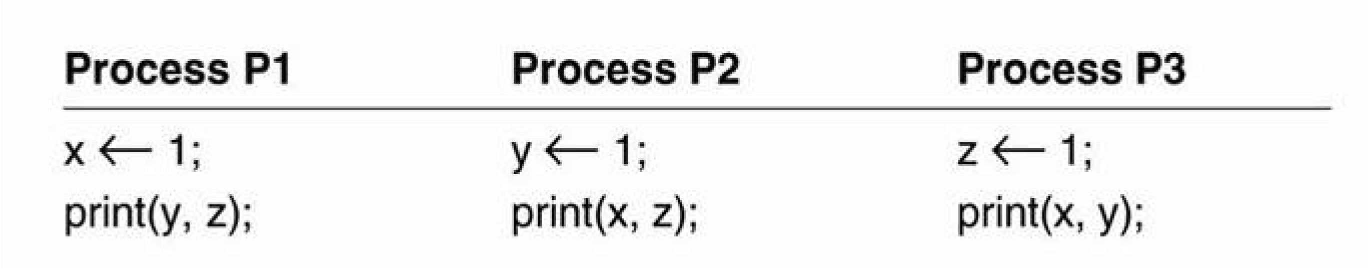
(a)



(b)

Example:

- Three concurrently-executing processes P1, P2, and P3 (Dubois et al., 1988).
- Three integer variables x, y, and z, which stored in a (possibly distributed) shared sequentially consistent data store.
- Assume that each variable is initialized to 0.
- An assignment corresponds to a write operation, whereas a print statement corresponds to a simultaneous read operation of its two arguments.
- All statements are assumed to be indivisible.

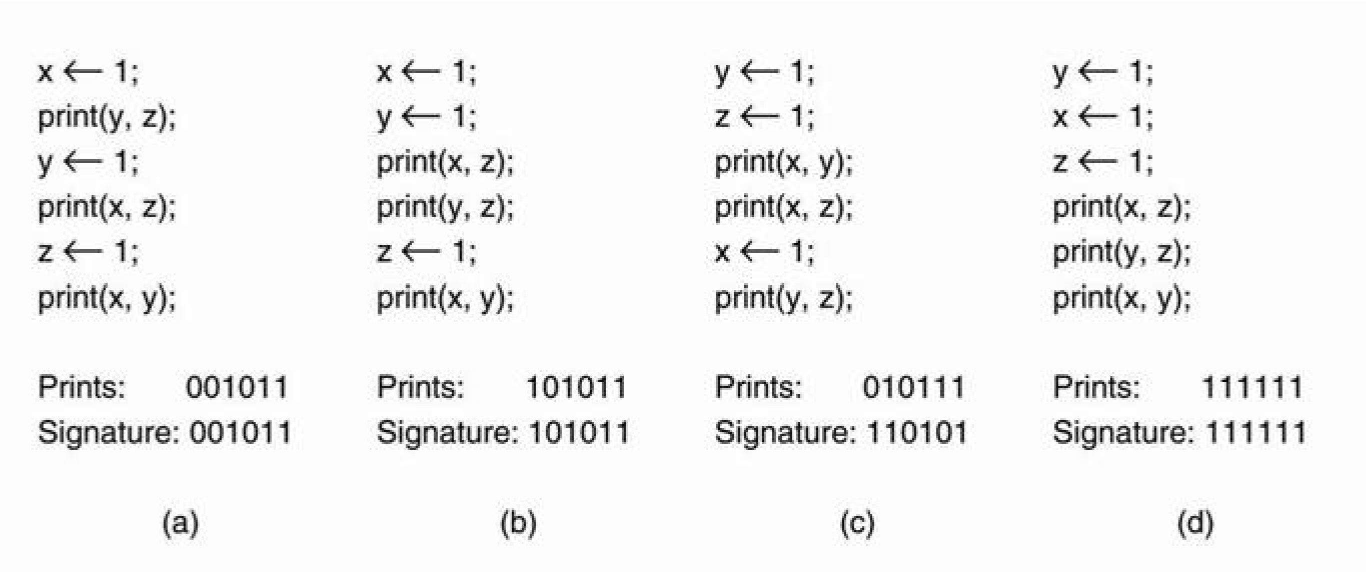


Process P1	Process P2	Process P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

- Various interleaved execution sequences are possible.
- With six independent statements, there are potentially 720 (6!) possible execution sequences
- Consider the 120 (5!) sequences that begin with x ←1.
  - Half of these have print (x,z) before y ←1 and thus violate program order.
  - Half have print (x,y) before z ←1 and also violate program order.
  - Only 1/4 of the 120 sequences, or 30, are valid.
  - Another 30 valid sequences are possible starting with y← 1

- Another 30 can begin with z ←1, for a total of 90 valid execution sequences.

Four valid execution sequences for the processes. The vertical axis is time.



```
x ← 1;               x ← 1;               y ← 1;               y ← 1;
print(y, z);         y ← 1;               z ← 1;               x ← 1;
y ← 1;               print(x, z);         print(x, y);         z ← 1;
print(x, z);         print(y, z);         print(x, z);         print(x, z);
z ← 1;               z ← 1;               x ← 1;               print(y, z);
print(x, y);         print(x, y);         print(y, z);         print(x, y);

Prints:    001011    Prints:    101011    Prints:    010111    Prints:    111111
Signature: 001011    Signature: 101011    Signature: 110101    Signature: 111111

     (a)                  (b)                  (c)                  (d)
```

- Figure (a) - The three processes are in order - P1, P2, P3.
- Each of the three processes prints two variables.
  - Since the only values each variable can take on are the initial value (0), or the assigned value (1), each process produces a 2-bit string.
  - The numbers after Prints are the actual outputs that appear on the output device.

- Output concatenation of P1, P2, and P3 in sequence produces a 6-bit string that characterizes a particular interleaving of statements.
  - This is the string listed as the Signature.

Examples of possible output:

- 000000 is not permitted - implies that the Print statements ran before the assignment statements, violating the requirement that statements are executed in program order.
- 001001 – is not permitted
  - First two bits 00 - y and z were both 0 when P1 did its printing.
    - This situation occurs only when P1 executes both statements before P2 or P3 start.
  - Second two bits 10 - P2 must run after P1 has started but before P3 has started.
  - Third two bits, 01- P3 must complete before P1 starts, but P1 execute first.

Causal Consistency

- *Writes that are potentially causally related must be seen by all processes in the same order.*
- *Concurrent writes (i.e. writes that are NOT causally related) may be seen in a different order by different processes.*

Example:

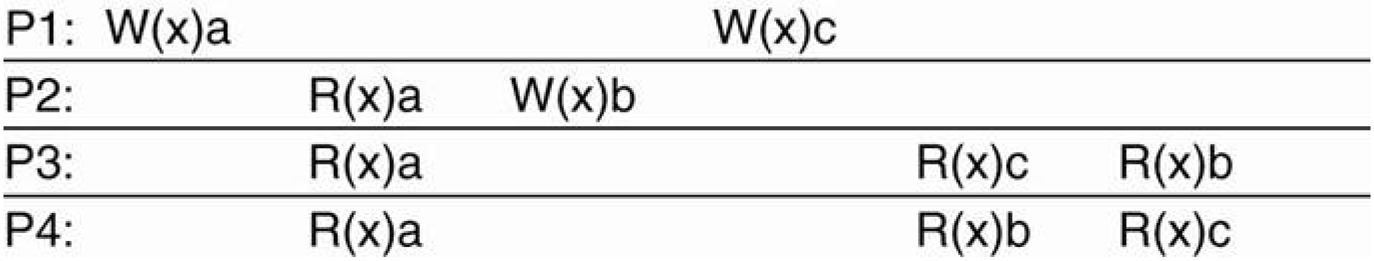
- Interaction through a distributed shared database.
- Process P1 writes data item x.
- Then P2 reads x and writes y.
- Reading of x and writing of y are potentially causally related because the computation of y may have depended on the value of x as read by P2 (i.e., the value written by P1).

- Conversly, if two processes spontaneously and simultaneously write two different data items, these are not causally related.
- Operations that are not causally related are said to be concurrent.

- For a data store to be considered causally consistent, it is necessary that the store 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 1: causal consistency

- This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.



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

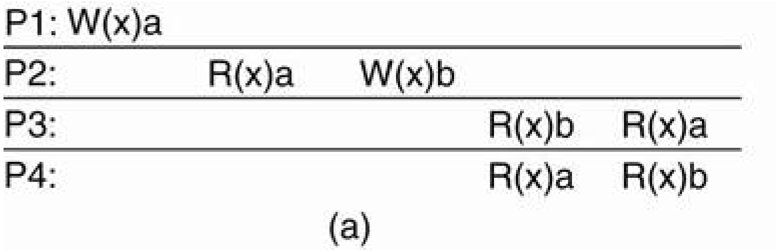
NOTE: The writes W2(x)b and W1(x)c are concurrent, so it is not required that all processes see them in the same order.

Example 2: causal consistency

• W2(x)b potentially depending on W1(x)a because b may result from a computation involving the value read by R2(x)a.	• Read has been removed, so W1(x)a and W2(x)b are now concurrent writes. • A causally-consistent store does not require concurrent writes to be globally

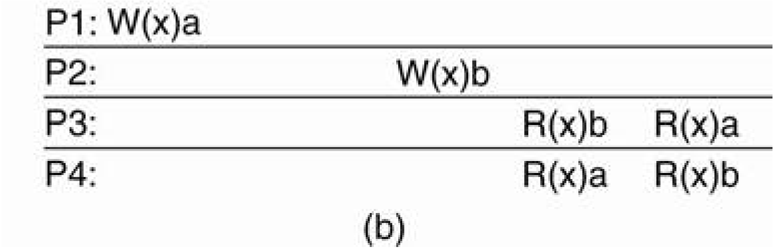
• The two writes are causally related, so all processes must see them in the same order. • It is incorrect.	ordered, • It is correct. • Note: situation that would not be acceptable for a sequentially consistent store.

(a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store.



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

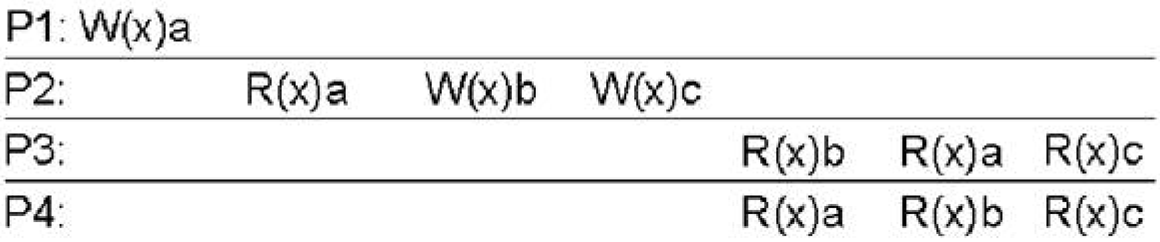
(b)

FIFO Consistency

Writes done by a single process are seen 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.

- Also called “PRAM Consistency” – Pipelined RAM.
- Easy to implement -There are no guarantees about the order in which different processes see writes – except that two or more writes from a single process must be seen in order.

Example:



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

- A valid sequence of FIFO consistency events.
- Note that none of the consistency models given so far would allow this sequence of events.

Weak Consistency

- Not all applications need to see all writes, let alone seeing them in the same order.
- Leads to Weak Consistency (which is primarily designed to work with *distributed critical sections*).
- This model introduces the notion of a synchronization variable", which is used update all copies of the data-store.

Properties Weak Consistency:

1. Accesses to synchronization variables associated with a data-store are *sequentially consistent*.
2. No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.
3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

Meaning

By doing a sync.

- a process can *force* the just written value out to all the other replicas.
- a process can be *sure* it's getting the most recently written value before it reads.

Essence: the weak consistency models enforce consistency on a *group of operations*, as opposed to individual reads and writes (as is the case with strict, sequential, causal and FIFO consistency).

Grouping Operations

- *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.

Convention: when a process enters its critical section it should acquire the relevant synchronization variables, and likewise when it leaves the critical section, it releases these variables.

Critical section: a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

Synchronization variables: are synchronization primitives that are used to coordinate the execution of processes based on asynchronous events.

- When allocated, synchronization variables serve as points upon which one or more processes can block until an event occurs.
- Then one or all of the processes can be unblocked. at the same time.
- Each synchronization variable has a current owner, namely, the process that last acquired it.
  - The owner may enter and exit critical sections repeatedly without having to send any messages on the network.
  - A process not currently owning a synchronization variable but wanting to acquire it has to send a message to the current owner asking for ownership and the current values of the data associated with that synchronization variable.
  - It is also possible for several processes to simultaneously own a synchronization variable in nonexclusive mode, meaning that they can read, but not write, the associated data.

Note: that the data in a process' critical section may be associated to different synchronization variables.

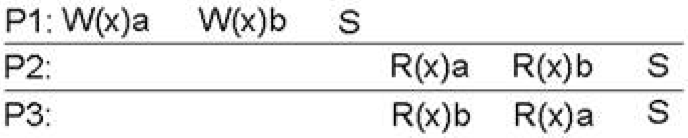
The following criteria must be met (Bershad et al., 1993)

1. 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.

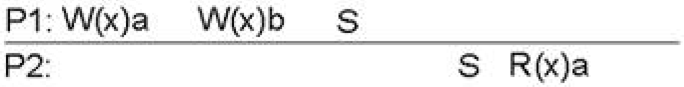
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.

3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Example:



(a)



(b)

a)A valid sequence of events for weak consistency.

- This is because P2 and P3 have yet to synchronize, so there’s no guarantees about the value in ‘x’.

b)An invalid sequence for weak consistency.

- P2 has synchronized, so it cannot read ‘a’ from ‘x’ – it should be getting ‘b’.

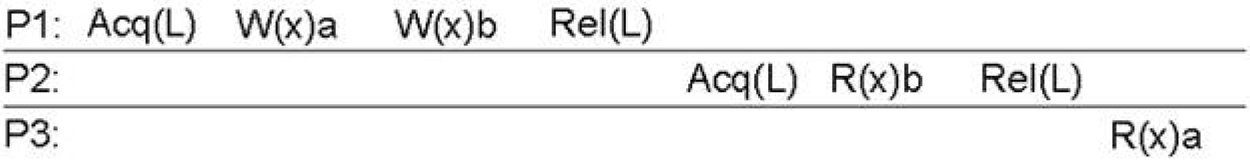
Release Consistency

- Question: how does a weakly consistent data-store know that the sync is the result of a read or a write?
  - Answer: It doesn’t!
- It is possible to implement efficiencies if the data-store is able to determine whether the sync is a read or write.
- Two sync variables can be used, acquire and release, and their use leads to the Release Consistency model.

Release Consistency

- *When a process does an acquire, the data-store will ensure that all the local copies of the protected data are brought up to date to be consistent with the remote ones if needs be.*
- *When a release is done, protected data that have been changed are propagated out to the local copies of the data-store.*

Example:



A valid event sequence for release consistency.

- Process P3 has not performed an *acquire*, so there are no guarantees that the read of ‘x’ is consistent.
- The data-store is simply not obligated to provide the correct answer.
- P2 does perform an *acquire*, so its read of ‘x’ is consistent.

Release Consistency Rules

A distributed data-store is “Release Consistent” if it obeys the following rules:

1. Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
2. Before a release is allowed to be performed, all previous reads and writes by the process must have completed.
3. Accesses to synchronization variables are *FIFO consistent* (sequential consistency is not required).

Entry consistency

- Acquire and release are still used, and the data-store meets 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.
- 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.

So:

- At an *acquire*, all remote changes to guarded data must be brought up to date.
- Before a write to a data item, a process must ensure that no other process is trying to write *at the same time*.

Locks are associates with individual data items, as opposed to the entire data-store.

- a **lock** is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution.

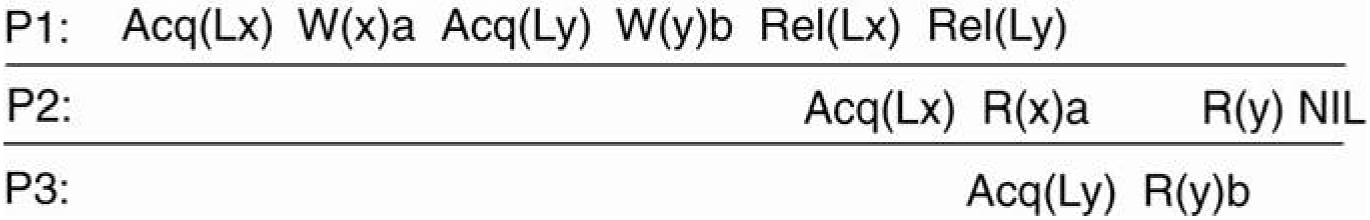
Example:

- P1 does an acquire for x, changes x once, after which it also does an acquire for y.

- Process P2 does an acquire for x but not for y, so that it will read value a for x, but may read NIL for y.
- Because process P3 first does an acquire for y, it will read the value b when y is released by P1.

Note: P2's read on 'y' returns NIL as no locks have been requested.

A valid event sequence for entry consistency.


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

Summary of Consistency Models

Consistency	Description: Consistency models that do not use synchronization operations.
Strict	**Absolute time ordering of all shared accesses matters**
Linearizability	**All processes must 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 used. Writes from different processes may not always be seen in that order.**

Consistency	Description: Models that do use synchronization operations
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.**

Consistency versus Coherence

- A number of processes execute read and write operations on a set of data items.
- A consistency model describes what can be expected with respect to that set when multiple processes concurrently operate on that data.
- The set is then said to be consistent if it adheres to the rules described by the model.

- Coherence models describe what can be expected to only a single data item (Cantin et al., 2005).
- Sequential consistency model - applied to only a single data item.

Client-Centric Consistency Models

- Above consistency models - maintaining a consistent (globally accessible) data-store in the presence of concurrent read/write operations.
- Another class of distributed datastore - characterized by *the lack of simultaneous updates*.
    - Here, the emphasis is more on maintaining a consistent view of things *for the individual client process* that is currently operating on the data-store.

Question: How fast should updates (writes) be made available to read-only processes?

- Most database systems: *mainly read*.
- DNS: *write-write conflicts* do no occur.
- WWW: as with DNS, except that heavy use of client-side caching is present: *even the return of stale pages is acceptable to most users*.

NOTE: all exhibit a high degree of acceptable inconsistency … with the *replicas* gradually become consistent over time.

Eventual Consistency

Special class of distributed data stores:

- Lack of simultaneous updates
- When updates occur they can easily be resolved.
- Most operations involve reading data.

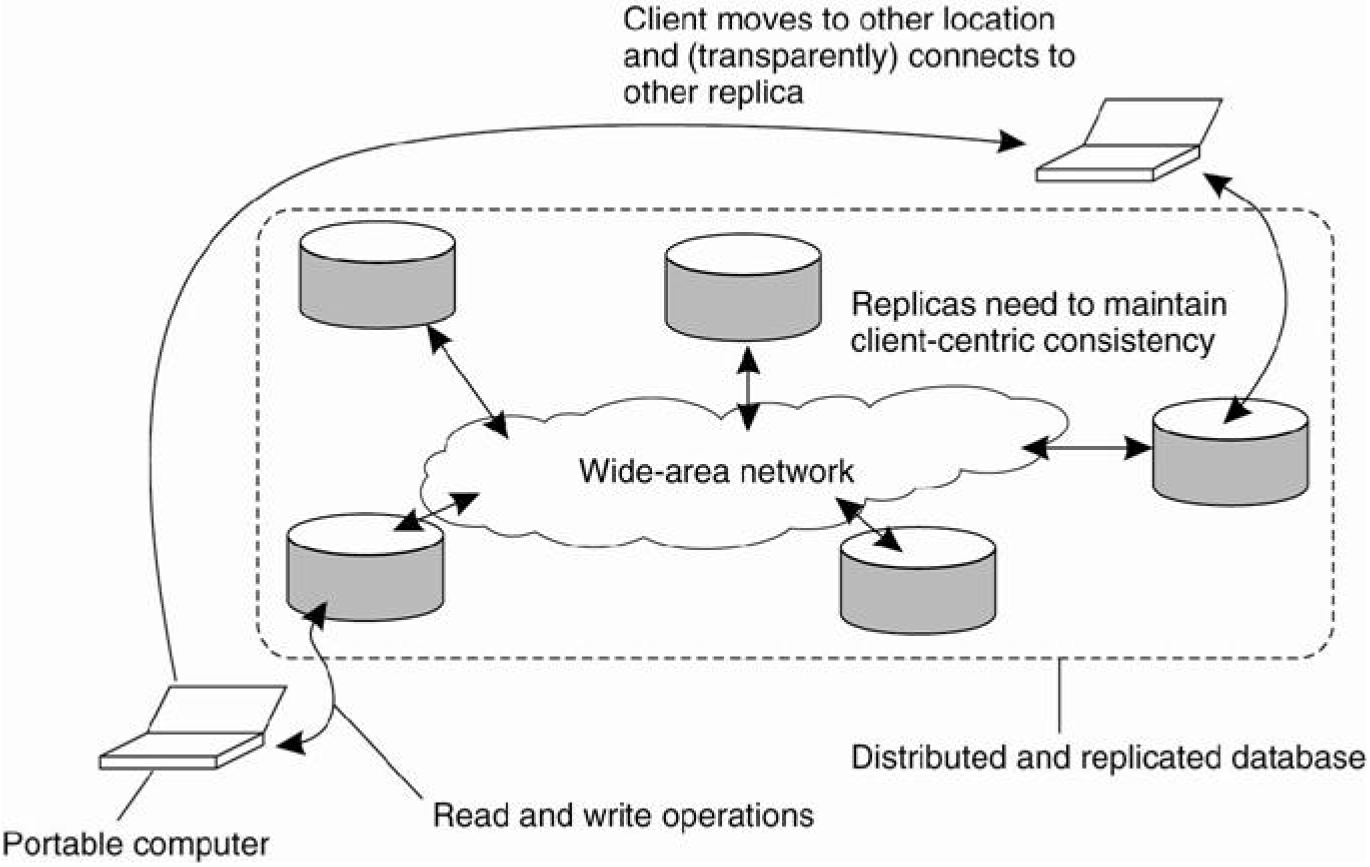
- These data stores offer a very weak consistency model, called eventual consistency.

The eventual consistency model states that, when no updates occur for a long period of time, eventually all updates will propagate through the system and all the replicas will be consistent.

Example: Consistency for Mobile Users

- 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.


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

- Client-centric consistency models originate from the work on Bayou [see, for example Terry et al. (1994) and Terry et al., (1998)].
- Bayou is a database system developed for mobile computing, where it is assumed that network connectivity is unreliable and subject to various performance problems.
    - Wireless networks and networks that span large areas, such as the Internet, fall into this category.
- Bayou distinguishes four different consistency models:
    1. monotonic reads

2. monotonic writes
3. read your writes
4. writes follow reads

Notation:

- $x_i[t]$ denotes the version of data item x at local copy L_i at time t.
- $WS\ x_i[t]$ *is the set of write operations at* L_i *that lead to* version x_i of x (at time *t*);
- If operations in $WS\ x_i[t1]$ have also been performed at local copy L_j at a later time $t2$, we write $WS\ (x_i[t1], x_j[t2])$.
- If the ordering of operations or the timing is clear from the context, the time index will be omitted.

Monotonic Reads

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.

- Monotonic-read consistency guarantees that if a process has seen a value of x at time t, it will never see an older version of x at a later time.

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.

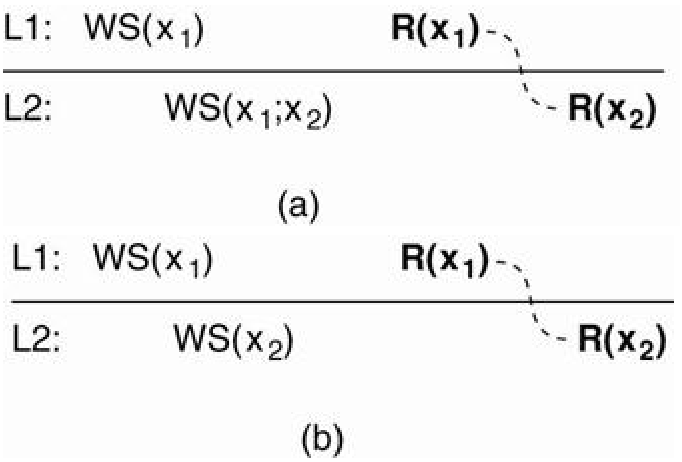
Example: Reading (not modifying) incoming mail while you are on the move.

- Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

Example:

- The read operations performed by a single process P at two different local copies of the same data store.
- Vertical axis - two different local copies of the data store are shown - L1 and L2
- Time is shown along the horizontal axis
- Operations carried out by a single process P in boldface are connected by a dashed line representing the order in which they are carried out.

(a) A monotonic-read consistent data store. (b) A data store that does not provide monotonic reads.



(a)	(b)
○ Process P first performs a read operation on x at L1, returning the value of x1 (at that time). ○ This value results from the write operations in WS (x1) performed at L1. ○ Later, P performs a read operation on x at L2, shown as R (x2). ○ To guarantee monotonic-read consistency, all operations in WS (x1) should have been propagated to L2 before the second read operation takes place.	○ Situation in which monotonic-read consistency is not guaranteed. ○ After process P has read x1 at L1, it later performs the operation R (x2) at L2 . ○ But, only the write operations in WS (x2) have been performed at L2 . ○ No guarantees are given that this set also contains all operations contained in WS (x1).

Monotonic Writes

In a monotonic-write consistent store, the following condition holds:

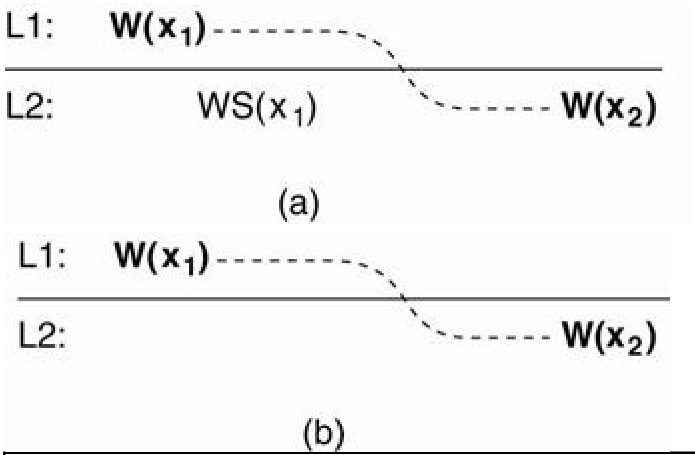
> A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

Hence: A write operation on a copy of item x is performed only if that copy has been brought up to date by means of any preceding write operation, which may have taken place on other copies of x. If need be, the new write must wait for old ones to finish.

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).

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.



(a)	(b)
○ Process P performs a write operation on x at local copy L1, presented as the operation W(x1). ○ Later, P performs another write operation on x, but this time at L2, shown as W (x2). ○ To ensure monotonic-write consistency, the previous write operation at L1 must have been propagated to L2. ○ This explains operation W (x1) at L2, and why it takes place before W (x2).	○ Situation in which monotonic-write consistency is not guaranteed. ○ Missing is the propagation of W(x1) to copy L2. ○ No guarantees can be given that the copy of x on which the second write is being performed has the same or more recent value at the time W(x1) completed at L1.

Read Your Writes

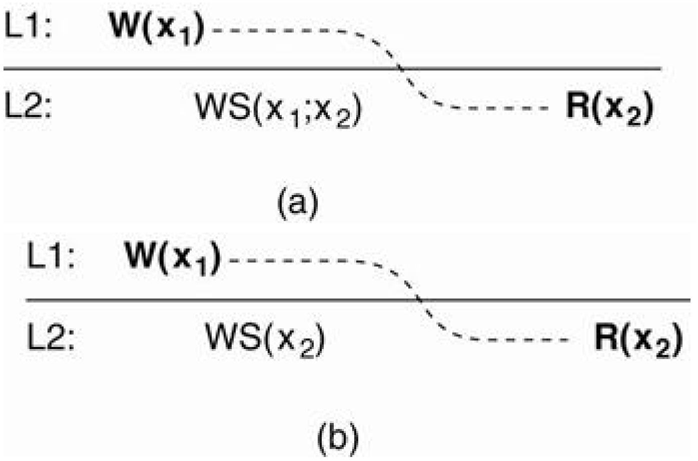
○ A client-centric consistency model that is closely related to monotonic reads is as follows. A data store is said to provide read-your-writes consistency, if the following condition holds:

> 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.

Hence: a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

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

Example:
(a) A data store that provides read-your-writes consistency. (b) A data store that does not.



(a)

(b)

(a)	(b)
○ Process P performed a write operation W(x1) and later a read operation at a different local copy. ○ Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation. ○ This is expressed by WS (x1;x2), which states that W (x1) is part of WS (x2).	○ W (x1) has been left out of WS (x2), meaning that the effects of the previous write operation by process P have not been propagated to L2.

Writes Follow Reads

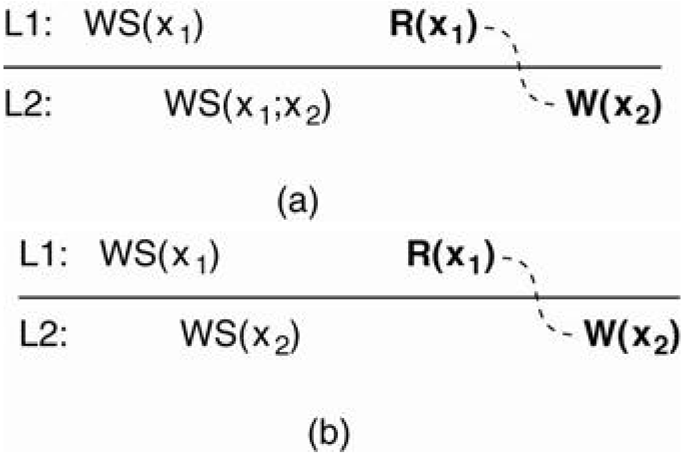
A data store is said to provide writes-follow-reads consistency, if the following holds:

> 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.

Hence: 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.

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

Example:
(a) A writes-follow-reads consistent data store. (b) A data store that does not



(a)	(b)
○ A process reads x at local copy L1. ○ The write operations that led to the value just read, also appear in the write set at L2, where the same process later performs a write operation. ○ (Note that other processes at L2 see those write operations as well.)	○ No guarantees are given that the operation performed at L2, ○ They are performed on a copy that is consistent with the one just read at L1.

Replica Management

Key issues:

- ○ Decide where, when, and by whom replicas should be placed
- ○ Which mechanisms to use for keeping the replicas consistent.

Placement problem:

- ○ Placing replica servers
  - Replica-server placement is concerned with finding the best locations to place a server that can host (part of) a data store.
- ○ Placing content.
  - Content placement deals with finding the best servers for placing content.

Replica-Server Placement

Essence: Figure out what the best *K* places are out of *N* possible locations.

1. Select best location out of *N* - *k* for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location

minimizes the average distance to all clients.) Computationally expensive. (Qiu et al. 2001)

2. Select the *k*-th largest autonomous system and place a server at the best-connected host. Computationally expensive. (Radoslavov et al. 2001)
    - An autonomous system (AS) can best be viewed as a network in which the nodes all run the same routing protocol and which is managed by a single organization.

3. Position nodes in a *d*-dimensional geometric space, where distance reflects latency. Identify the *K* regions with highest density and place a server in every one. Computationally cheap. (Szymaniak et al. 2006)

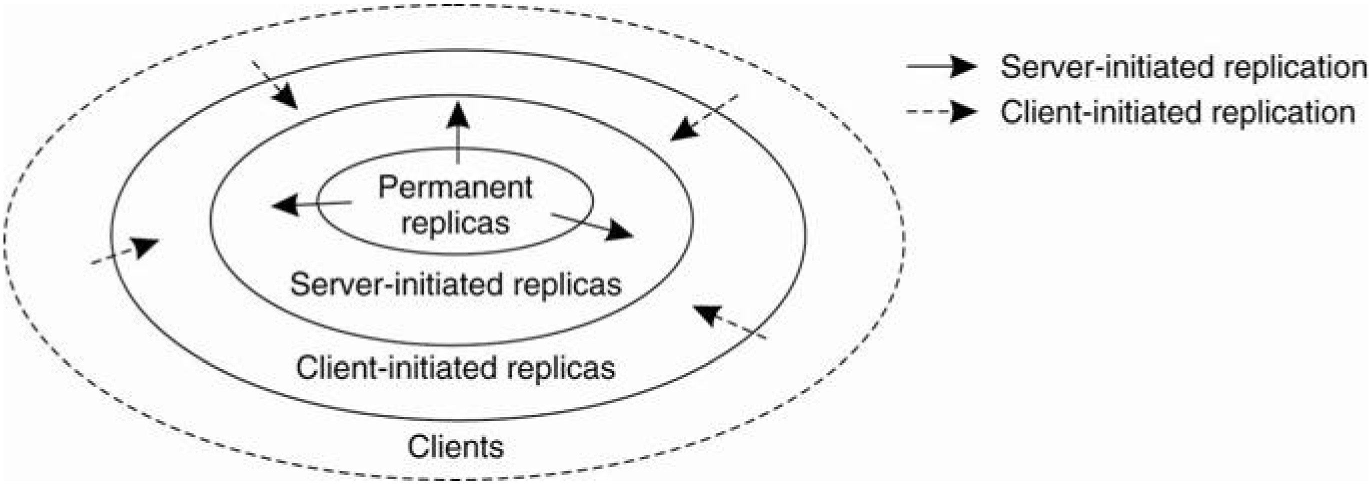
Content Replication and Placement

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)

- The logical organization of different kinds of copies of a data store into three concentric rings.


Server-initiated replication
Client-initiated replication
Permanent replicas
Server-initiated replicas
Client-initiated replicas
Clients

Examples:

- The initial set of replicas that constitute a distributed data store.
- The number of permanent replicas is small.
- The files that constitute a site are replicated across a limited number of servers at a single location.
    - Whenever a request comes in, it is forwarded to one of the servers, for instance, using a round-robin strategy.
- Alternatively, a database is distributed and possibly replicated across a number of geographically dispersed sites.
    - This architecture is generally deployed in federated databases (Sheth and Larson, 1990).

Server-Initiated Replicas

Copies of a data store are created to enhance performance

Example:

- Dynamic placement of replica servers in Web hosting services.
- Sivasubramanian et al. (2004) provide an overview of replication in Web hosting services

Example:

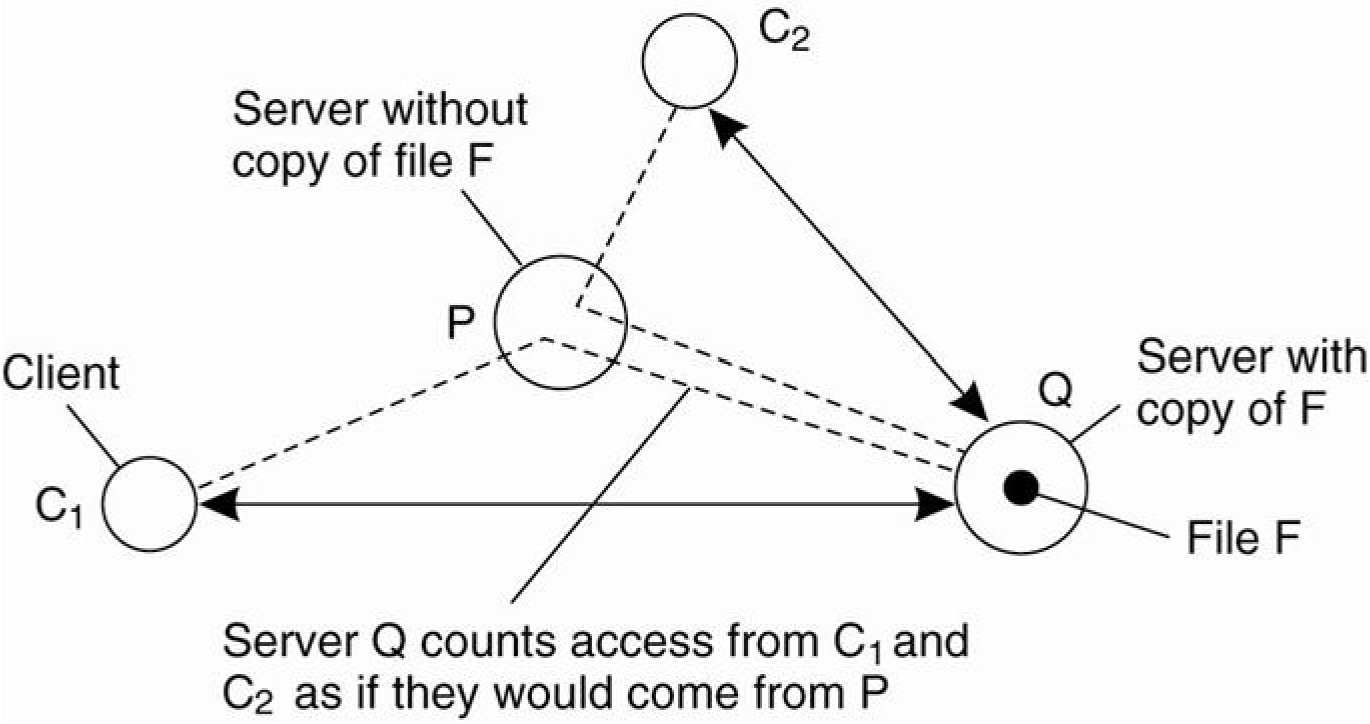
- Placement of content with replica servers in place
- Dynamic replication of files for Web hosting servicea is described in Rabinovich et al. (1999).
- Algorithm is designed to support Web pages for which reason it assumes that updates are relatively rare compared to read requests.
    - Algorithm considers:
        1. Replication take used to reduce the load on a server.
        2. Specific files on a server can be migrated or replicated to servers placed in the proximity of clients that issue many requests for those files.

- Each server keeps track of:
    - access counts per file
    - where access requests originate

- Given a client C, each server can determine which of the servers in the Web hosting service is closest to C.
- If client C1 and client C2 share the same "closest" server P, all access requests for file F at server Q from C1 and C2 are jointly registered at Q as a single access count cntQ(P,F).

- Removed from S - when the number of requests for a file F at server S drops below a deletion threshold del (S,F)
- Replicate F – when replication threshold rep (S,F) is surpassed
- Migrate F – when the number of requests lies between the deletion and replication thresholds

Counting access requests from different clients.


C_2
Server without copy of file F
P
Client
C_1
Q
Server with copy of F
File F
Server Q counts access from C_1 and C_2 as if they would come from P

Client-Initiated Replicas

- Client-initiated replicas aka (client) caches.
- Cache is a local storage facility that is used by a client to temporarily store a copy of the data it has just requested.
- Managing cache is left to the client.
- Used to improve access times to data.

Approaches to Cache placement:

- Traditional file systems: data files are rarely shared at all (see, e.g., Muntz and Honeyman, 1992; and Blaze, 1993) rendering a shared cache useless.
- LAN caches : machine shared by clients on the same local-area network.
- WAN caches: place (cache) servers at specific points in a wide-area network and let a client locate the nearest server.
    - When the server is located, it can be requested to hold copies of the data the client was previously fetching from somewhere else, as described in Noble et al. (1999).

Content Distribution

State versus Operations

What is to be propagated:

1. Propagate only a notification of an update.
    - Performed by an invalidation protocol
    - In an invalidation protocol, other copies are informed that an update has taken place and that the data they contain are no longer valid.
        i. Adv. - Use less bandwidth

2. Transfer data from one copy to another.
    - Useful when the read-to-write ratio is relatively high
    - Modifications:
        i. Log the changes and transfer only those logs to save bandwidth.
        ii. Aggregate transfers so that multiple modifications are packed into a single message to save communication overhead.

3. Propagate the update operation to other copies - active replication
    - Do not to transfer any data modifications at all
    - Tell each replica which update operation it should perform and send only the parameter values that those operations need
    - Assumes that each replica is represented by a process capable of "actively" keeping its associated data up to date by performing operations (Schneider, 1990)
    - Adv. –
        - updates can be propagated at minimal bandwidth costs, provided the size of the parameters associated with an operation are relatively small
        - the operations can be of arbitrary complexity, which may allow further improvements in keeping replicas consistent
    - Disadv. -
        - more processing power may be required by each replica

Pull versus Push Protocols

Should updates be pulled or pushed?

- **Push-based** - server-based protocols - updates are propagated to other replicas without those replicas requesting updates.
    - Used between permanent and server-initiated replicas and can used to push updates to client caches.
    - Used when replicas need to maintain a relatively high degree of consistency

- **Pull-based** - client-based protocols - a server or client requests another server to send it updates it has at that moment.
    - Used by client caches.

Comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems.

Issue	*Push-based*	*Pull-based*
State at server	Must keep 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

- **Trade-offs have lead to a hybrid form of update propagation based on - leases.**
    - A lease is a promise by the server that it will push updates to the client for a specified time.
    - When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary.
        - Or a client requests a new lease for pushing updates when the previous lease expires.
    - Introduced by Gray and Cheriton (1989).
    - Dynamically switches between push-based and pull-based strategy.

Issue: Make lease expiration time dependent on system's behavior (adaptive leases): Duvvuri et al. (2003)

- **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

Epidemic Protocols

- Used to implement *Eventual Consistency* (note: these protocols are used in Bayou).
- Main concern is the propagation of updates to all the replicas in *as few a number of messages as possible*.
- Idea is to "infect" as many replicas as quickly as possible.

Infective replica: a server that holds an update that can be spread to other replicas.
Susceptible replica: a yet to be updated server.
Removed replica: an updated server that will not (or cannot) spread the update to any other replicas.

- The trick is to get all susceptible servers to either infective or removed states as quickly as possible without leaving any replicas out.

The Anti-Entropy Protocol

- Entropy: "a measure of the degradation or disorganization of the universe".
- Server P picks Q at random and exchanges updates, using one of three approaches:
  1. P only pushes to Q.
  2. P only pulls from Q.
  3. P and Q push and pull from each other.
- Sooner or later, all the servers in the system will be infected (updated). Works well.

The Gossiping Protocol

This variant is referred to as "gossiping" or "rumour spreading", as works as follows:

1. P has just been updated for item 'x'.
2. It immediately pushes the update of 'x' to Q.
3. If Q already knows about 'x', P becomes disinterested in spreading any more updates (rumours) and is removed.
4. Otherwise P gossips to another server, as does Q.

This approach is good, but can be shown not to guarantee the propagation of all updates to all servers.

The Best of Both Worlds

A mix of anti-entropy and gossiping is regarded as the best approach to rapidly infecting systems with updates.
BUT... what about ***removing*** data?

- ○ Updates are easy, deletion is much, much harder!
- ○ Under certain circumstances, after a deletion, an "old" reference to the deleted item may appear at some replica and cause the deleted item to be *reactivated*!
- ○ One solution is to issue "Death Certificates" for data items – these are a special type of update.
- ○ Only problem remaining is the eventual removal of "old" death certificates (with which timeouts can help).

Consistency Protocols

A consistency protocol describes an implementation of a specific consistency model.

The most widely implemented models are:
1. Sequential Consistency.
Those in which operations can be grouped through locking or transactions
2. Weak Consistency (with sync variables).
3. Atomic Transactions

Continuous Consistency

Numerical Errors (Yu and Vahdat 2000)

Principle: consider a data item x and let $weight(W)$ denote the numerical change in its value after a write operation W.

- Assume that $\forall W : weight(W) > 0$.
- W is initially forwarded to one of the N replicas, denoted as $origin(W)$.
- $TW[i, j]$ are the writes executed by server S_i that originated from S_j:

$$TW[i,j] = \sum \{weight(W) \mid origin(W) = S_j \ \& \ W \in L_i\}$$

Note: Actual value $v(t)$ of x:

$$v(t) = v(0) + \sum_{k=1}^{N} TW[k,k]$$

value v_i of x at replica i:

$$v_i = v(0) + \sum_{k=1}^{N} TW[i,k]$$

Problem: We need to ensure that $v(t)$ - $v_i < \delta_i$ for every server S_i.

Approach: Let every server S_k maintain a **view** $TW_k[i, j]$ of what it believes is the value of $TW[i, j]$. This information can be gossiped when an update is propagated.

Note: $0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$.

Solution: S_k sends operations from its log to S_i when it sees that $TW_k[i, j]$ is getting too far from $TW[i, j]$

in particular, when $TW[k, k] - TW_k[i, k] > \delta I /(N - 1)$.

Primary-Based Protocols

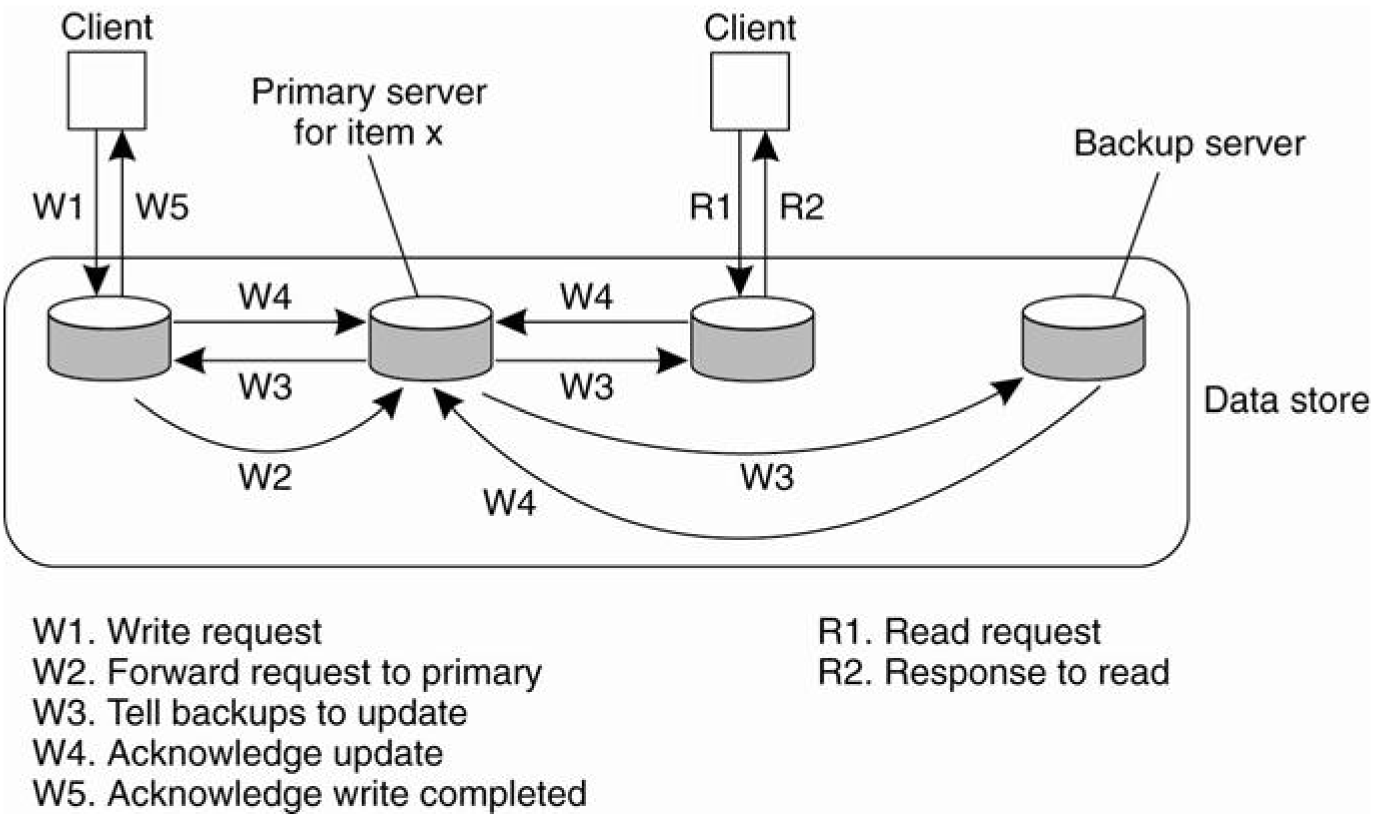
- Use for sequential consistency
- Each data item is associated with a “primary” replica.
- The primary is responsible for coordinating writes to the data item.
- There are two types of Primary-Based Protocol:
  1. Remote-Write.
  2. Local-Write.

Remote-Write Protocols

- AKA primary backup protocols (Budhiraja et al., 1993)
- All writes are performed at a single (remote) server.
- Read operations can be carried out locally.
- This model is typically associated with traditional client/server systems.

Example:

1. A process wanting to perform a write operation on data item x, forwards that operation to the primary server for x.
2. The primary performs the update on its local copy of x, and forwards the update to the backup servers.
3. Each backup server performs the update as well, and sends an acknowledgment back to the primary.
4. When all backups have updated their local copy, the primary sends an acknowledgment back to the initial process.



The Bad and Good of Primary-Backup

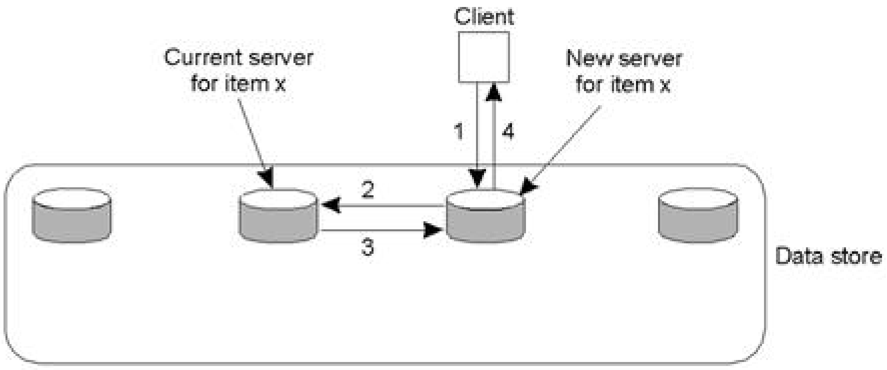
- Bad: Performance!
  - *All of those writes can take a long time (especially when a "blocking write protocol" is used).*
  - Using a non-blocking write protocol to handle the updates can lead to fault tolerant problems (which is our next topic).
- Good: as the *primary* is in control, all writes can be sent to each backup replica *IN THE SAME ORDER*, making it easy to implement *sequential consistency*.

Local-Write Protocols

- AkA fully migrating approach
- A single copy of the data item is still maintained.
- Upon a write, the data item gets transferred to the replica that is writing.
  - the status of *primary* for a data item is *transferrable*.

Process: whenever a process wants to update data item x, it locates the primary copy of x, and moves it to its own location.

Example:
Primary-based local-write protocol in which a single copy is *migrated* between processes (prior to the read/write).



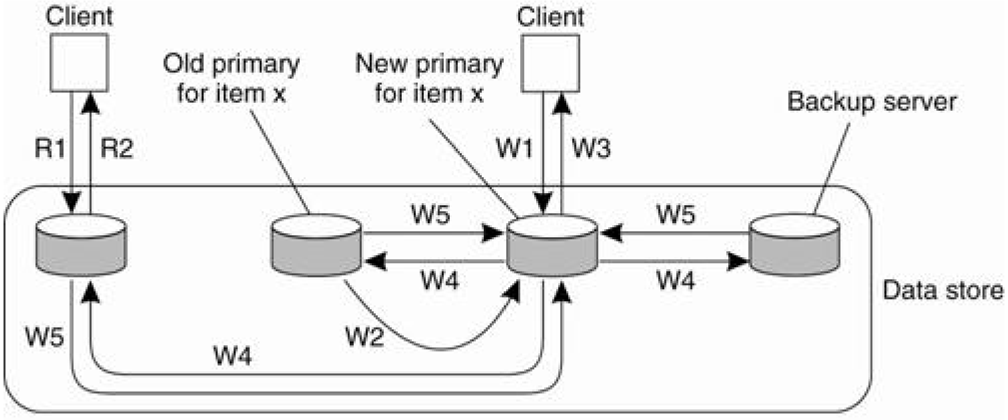
1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Local-Write Issues

- Question to be answered by any process about to read from or write to the data item is:

  "Where is the data item right now?"
- Processes can spend more time actually locating a data item than using it!

Primary-backup protocol in which the primary migrates to the process wanting to perform an update.



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

Advantage:

- Multiple, successive write operations can be carried out locally, while reading processes can still access their local copy.
- Can be achieved only if a nonblocking protocol is followed by which updates are propagated to the replicas after the primary has finished with locally performing the updates.

Replicated-Write Protocols

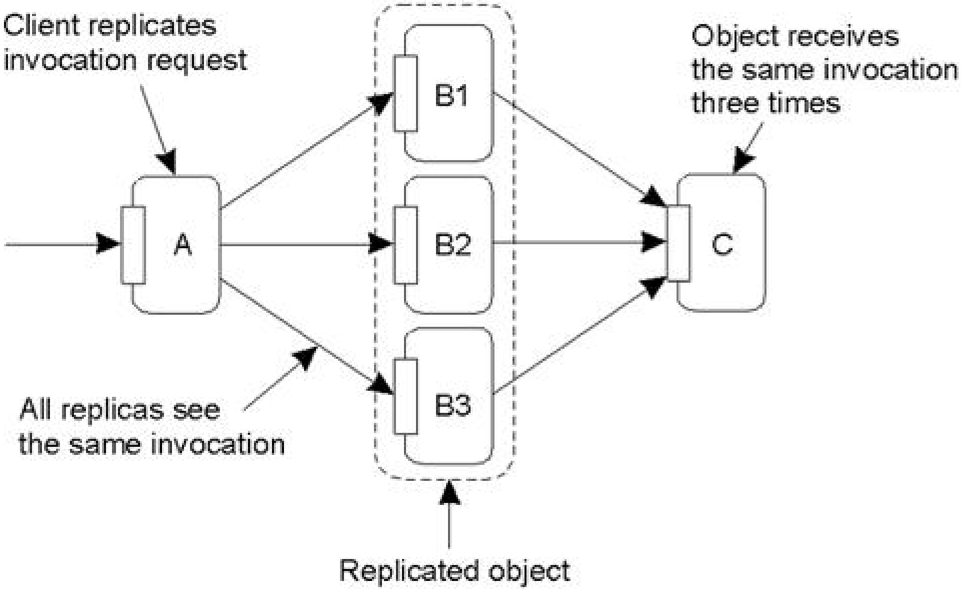
- Writes can be carried out at *any* replica.
- There are two types:
  1. Active Replication.
  2. Majority Voting (Quorums).

A

ctive Replication

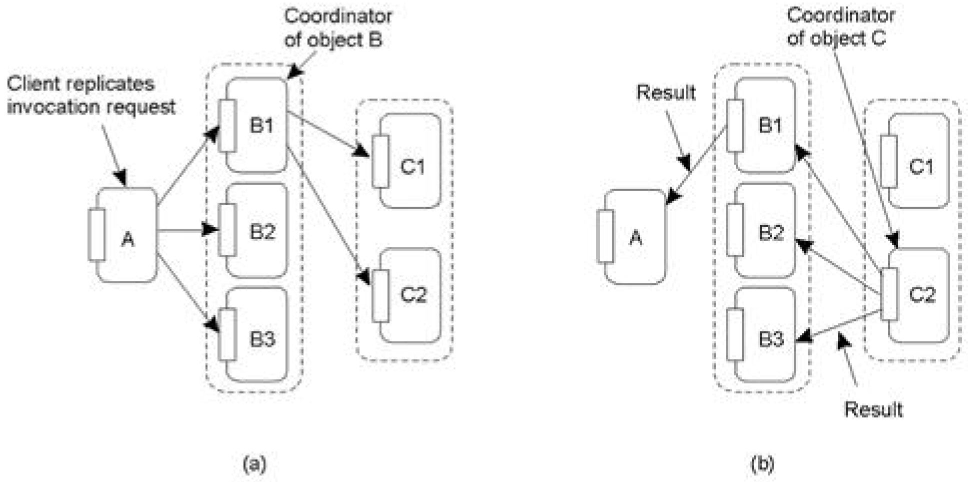
- A special process carries out the update operations at each replica.
- Lamport's timsestamps can be used to achieve total ordering, but this does not scale well within Distributed Systems.
- An alternative/variation is to use a *sequencer*, which is a process that assigns a unique ID# to each update, which is then propagated to all replicas.

Issue: *replicated invocations*


Client replicates invocation request
Object receives the same invocation three times
A
B1
B2
B3
C
All replicas see the same invocation
Replicated object

The problem of replicated invocations –

- 'B' is a replicated object (which itself calls 'C').
- When 'A' calls 'B', how do we ensure 'C' isn't invoked three times?
-



a)Using a coordinator for 'B', which is responsible for forwarding an invocation request from the replicated object to 'C'.

b)Returning results from 'C' using the same idea: a coordinator is responsible for returning the result to all 'B's. Note the single result returned to 'A'.

Quorum-Based Protocols

- Clients must request and acquire permissions from multiple replicas before either reading/writing a replicated data item.
- Methods devised by Thomas (1979) and generalized by Gifford (1979).

Example:

- A file is replicated within a distributed file system.
- To update a file, a process must get approval from a majority of the replicas to perform a write.
  - The replicas need to agree *to also perform the write*.
- After the update, the file has a new version # associated with it (and it is set at all the updated replicas).
- To read, a process contacts a majority of the replicas and asks for the version # of the files.
  - If the version # is the same, then the file must be the most recent version, and the read can proceed.

Gifford's method

- To read a file of which N replicas exist a client needs to assemble a read quorum, an arbitrary collection of any NR servers, or more.
- To modify a file, a write quorum of at least NW servers is required.
- The values of NR and NW are subject to the following two constraints:

$$NR + NW > N$$

$$NW > N/2$$

- First constraint prevents read-write conflicts
- Second constraint prevents write-write conflicts.

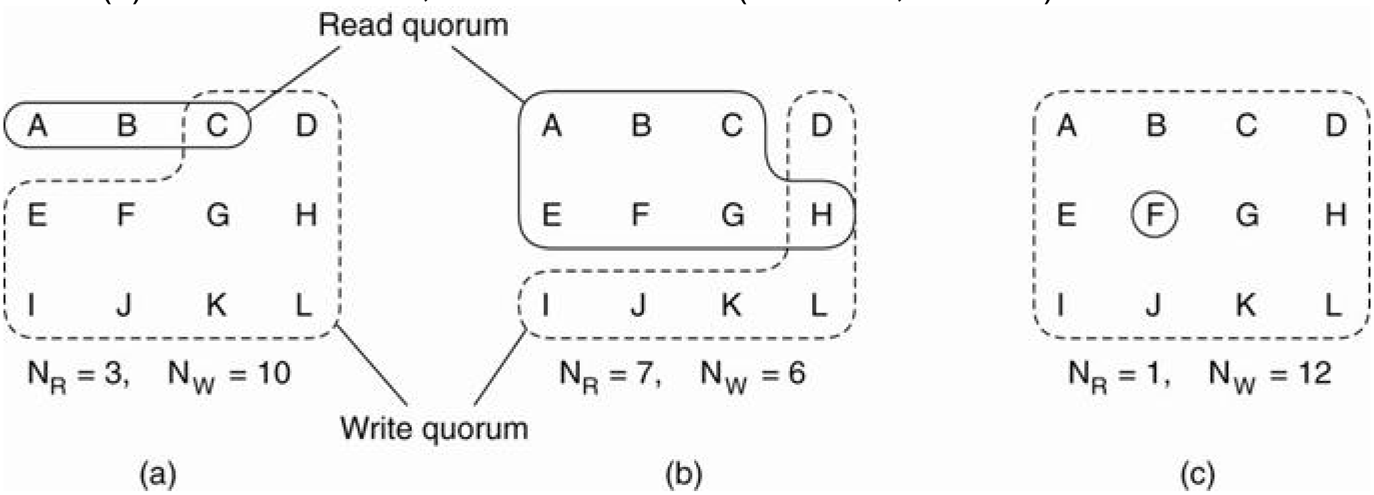
Only after the appropriate number of servers has agreed to participate can a file be read or written.

Example:

- NR = 3 and NW = 10
- Most recent write quorum consisted of the 10 servers C through L.
- All get the new version and the new version number.
- Any subsequent read quorum of three servers will have to contain at least one member of this set.
- When the client looks at the version numbers, it will know which is most recent and take that one.

Three examples of the voting algorithm:

(a) A correct choice of read and write set.
(b) A choice that may lead to write-write conflicts.
(c) A correct choice, known as ROWA (read one, write all).



(b) a write-write conflict may occur because NW ≤ N/2.

- If one 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, then the two updates will both be accepted without detecting that they actually conflict.

(c) NR = 1, making it possible to read a replicated file by finding any copy and using it.

- poor performance, becausewrite updates need to acquire all copies. (aka Read-One, Write-All (ROWA)). T

Cache-Coherence Protocols

These are a special case, as the cache is typically controlled by the client *not* the server.

Coherence Detection Strategy:

- When are inconsistencies actually detected?
- Statically at compile time: extra instructions inserted.
- Dynamically at runtime: code to check with the server.

Coherence Enforcement Strategy

- How are caches kept consistent?
- Server Sent: invalidation messages.
- Update propagation techniques.
- Combinations are possible.

See these papers of middle-ware solutions for further discussion (Min and Baer, 1992; Lilja, 1993; and Tartalja and Milutinovic, 1997).

What about Writes to the Cache?

Read-only Cache: updates are performed by the server (ie, pushed) or by the client (ie, pulled whenever the client notices that the cache is *stale*).
Write-Through Cache: the client modifies the cache, then sends the updates to the server.
Write-Back Cache: delay the propagation of updates, allowing multiple updates to be made locally, then sends the most recent to the server (this can have a dramatic *positive* impact on performance).