LECTURE 25

Consistency and Replication

Course/Slides Credits

Note: all course presentations are based on those developed by Andrew S. Tanenbaum and Maarten van Steen. They accompany their "Distributed Systems: Principles and Paradigms" textbook. http://www.prenhall.com/divisions/esm/app/author_tanenbaum/custom/dist_sys_1e/index.html

And additions made by Paul Barry in course CW046-4: Distributed Systems

http://glasnost.itcarlow.ie/~barryp/net4.html

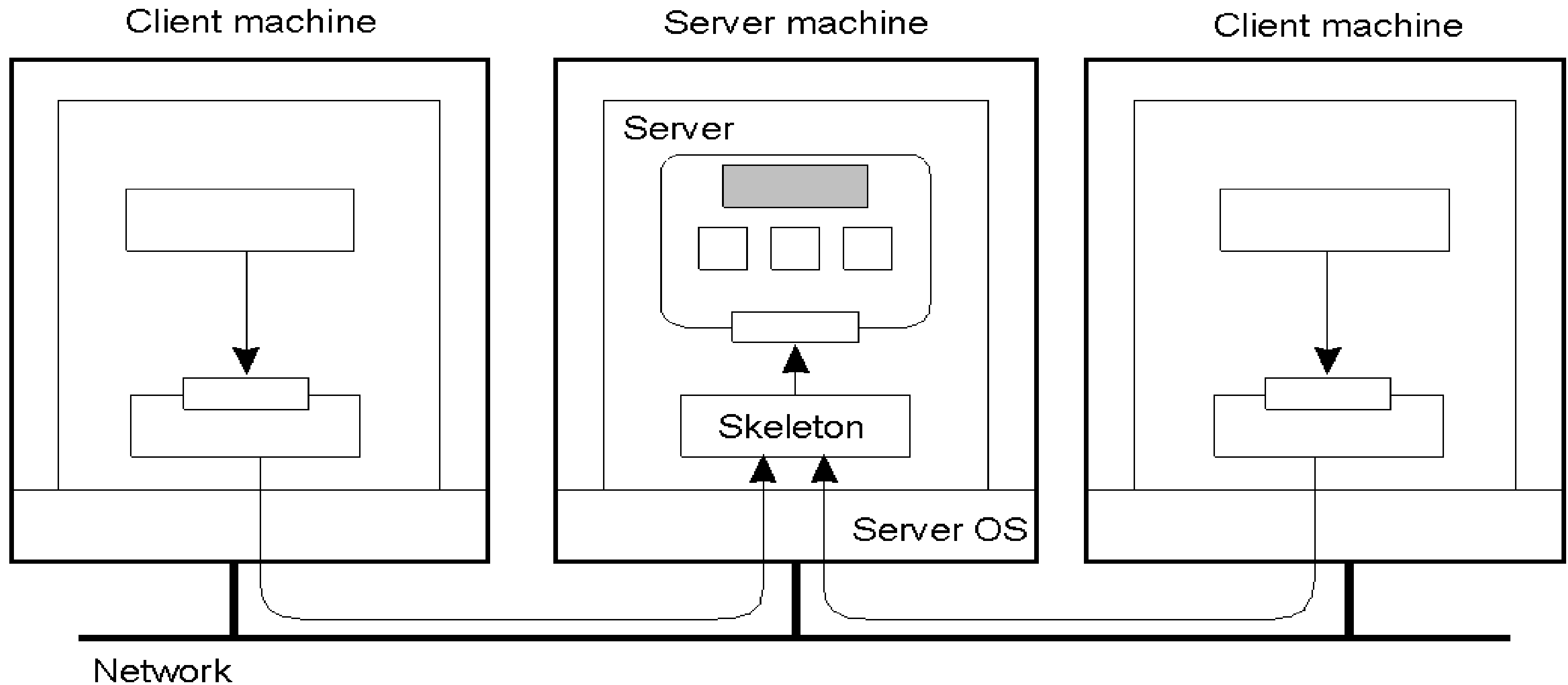
Why Replicate Data?

- Enhance reliability.
- Improve performance.
- But: if there are many replicas of the same thing, how do we keep all of them up-to-date? How do we keep the replicas *consistent*?
- Consistency can be achieved in a number of ways. We will study a number of *consistency models*, as well as *protocols* for implementing the models.

More on Replication

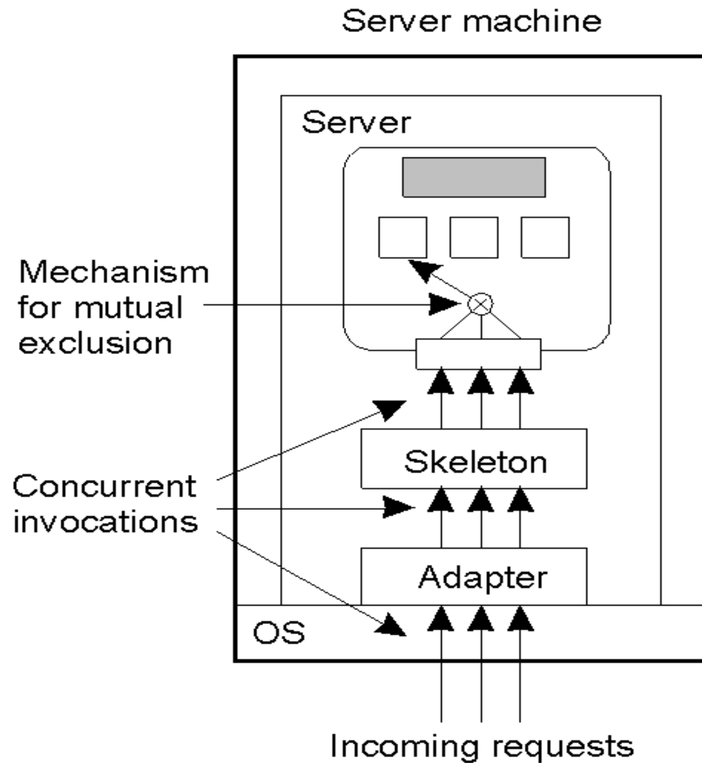
- Replicas allows remote sites to continue working in the event of local failures.
- It is also possible to protect against data corruption.
- Replicas allow data to reside close to where it is used.
- This directly supports the distributed systems goal of enhanced *scalability*.
- Even a large number of replicated "local" systems can improve performance: think of clusters.
- So, what's the catch?
  - It is not easy to keep all these replicas *consistent*.

Concurrent Object Access: Problem

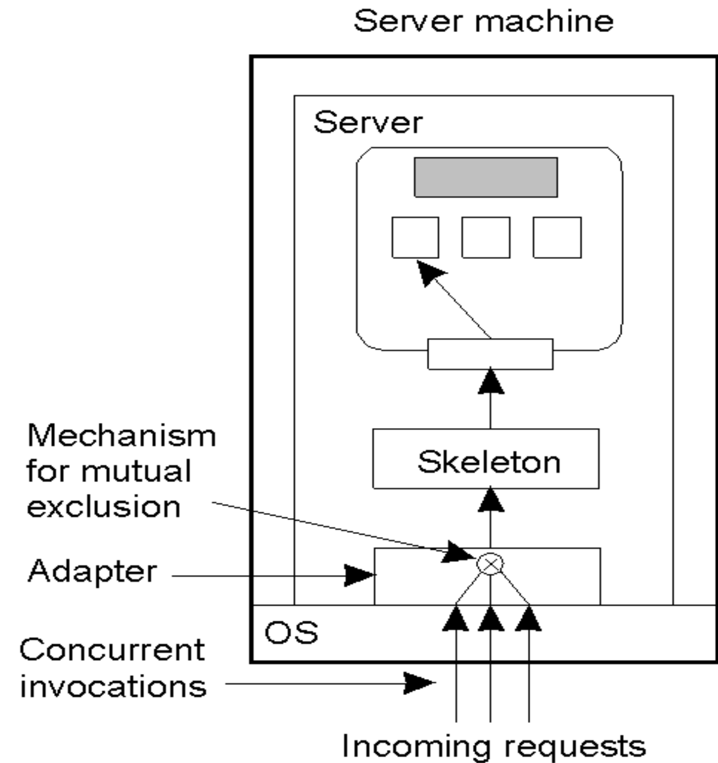


Organization of a distributed remote object shared by two different clients. But, how do we protect the object in the presence of multiple simultaneous access?

Concurrent Object Access: Solutions



(a)

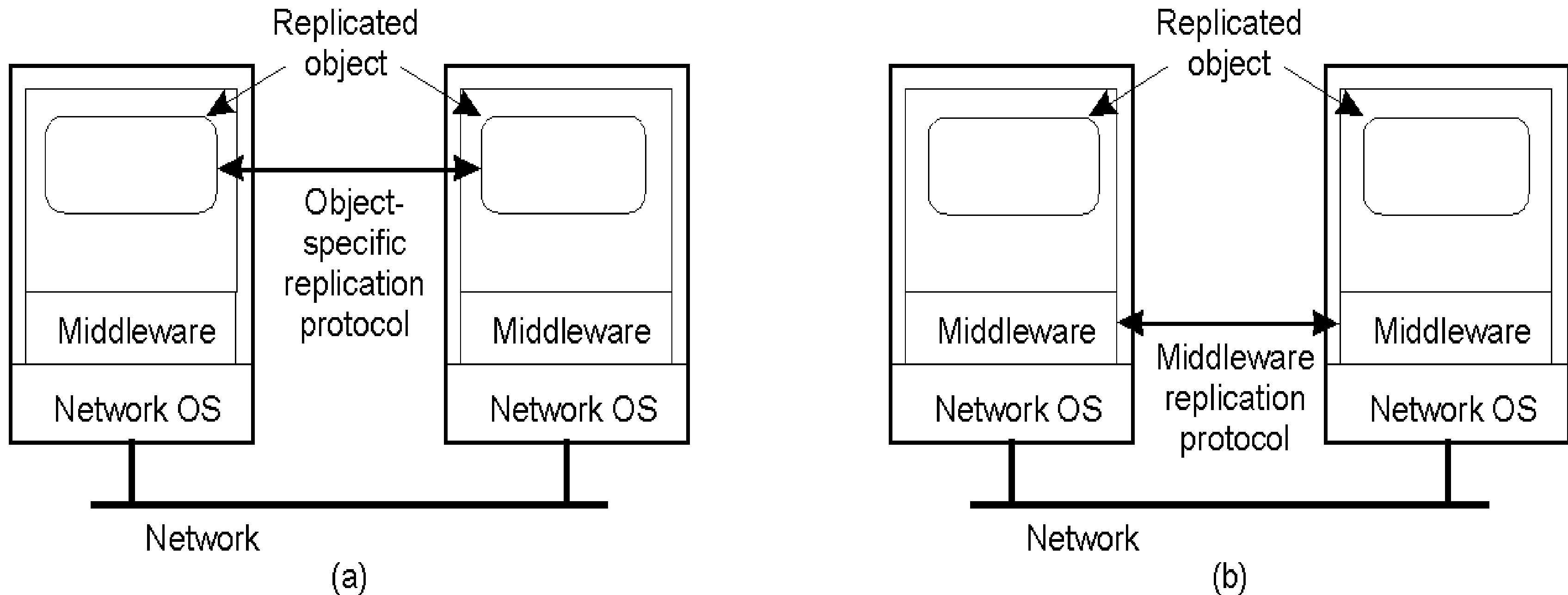


(b)

a) A remote object capable of handling concurrent invocations on its own.

b) A remote object for which an object adapter is required to handle

Object Replication: Solutions



a) A distributed system for replication-aware distributed objects – the object itself is "aware" that it is replicated. This is a very flexible set-up, but can be costly in that the DS developer has to concern themselves with replication/consistency.

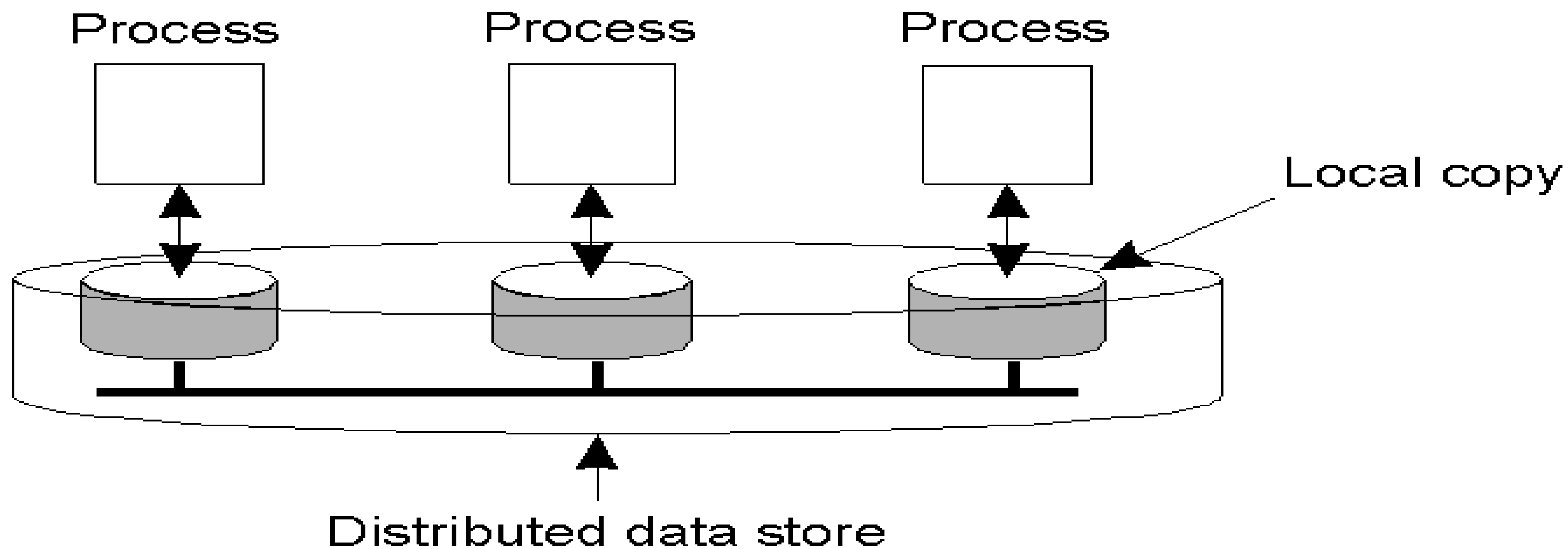
b) A distributed system responsible for replica management – less flexible, but removes burden from the DS developer. The *most*

Replication and Scalability

- Replication is a widely-used scalability technique: think of Web clients and Web proxies.
- When systems scale, the first problems to surface are those associated with performance – as the systems get bigger
(e.g., more users), they get often slower.
- Replicating the data and moving it closer to where it is needed helps to solve this scalability problem.
- A problem remains: how to efficiently synchronize all of the replicas created to solve the scalability issue?
- Dilemma: adding replicas improves scalability, but incurs the (oftentimes considerable) overhead of keeping the replicas
up-to-date!!!
- As we shall see, the solution often results in a relaxation

Data-Centric Consistency Models

- A data-store can be read from or written to by any process in a DS.
- A local copy of the data-store (replica) can support "fast reads".
- However, a write to a local replica needs to be propagated to *all* remote replicas.



Various consistency models help to understand the various mechanisms used to achieve and enable this.

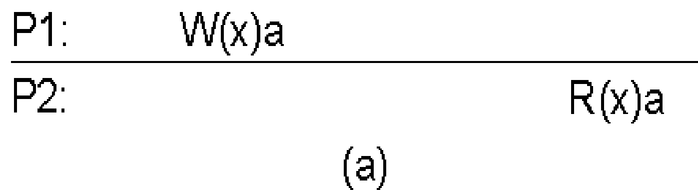
What is a Consistency Model?

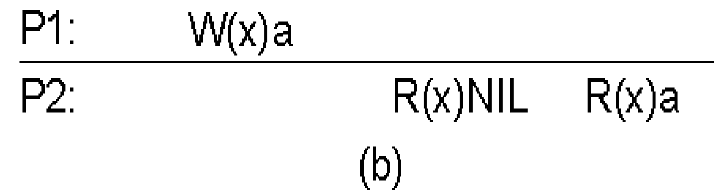
- A "consistency model" is a CONTRACT between a DS data-store and its processes.
- If the processes agree to the rules, the data-store will perform properly and as advertised.
- We start with *Strict Consistency*, which is defined as:
  - *Any read on a data item 'x' returns a value corresponding to the result of the most recent write on 'x' (regardless of where the write occurred).*

Consistency Model Diagram Notation

- $W_i(x)a$ – a write by process 'i' to item 'x' with a value of 'a'. That is, 'x' is set to 'a'.
- (Note: The process is often shown as 'P_i').
- $R_i(x)b$ – a read by process 'i' from item 'x' producing the value 'b'. That is, reading 'x' returns 'b'.
- Time moves from left to right in all diagrams.

Strict Consistency Diagrams





- 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.

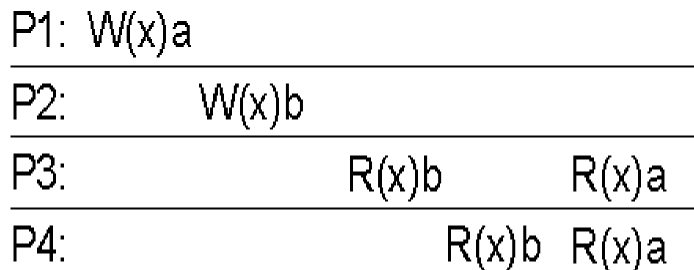
- With *Strict Consistency*, all writes are *instantaneously visible* to all processes and *absolute global time order* is maintained throughout the DS. This is the consistency model "Holy Grail" – not at all easy in the real world, and all but *impossible* within a DS.

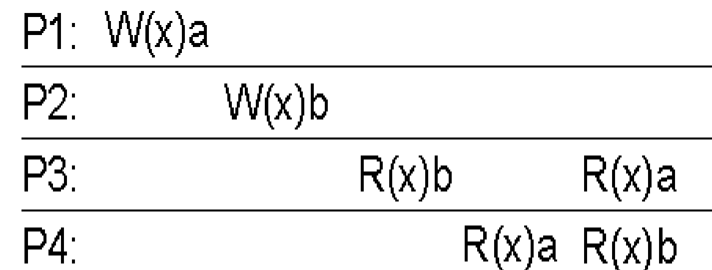
Sequential Consistency

- A weaker consistency model, which represents a relaxation of the rules.
- It is also much easier (possible) to implement.
- Definition of “Sequential Consistency”:
    - *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 the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*

Sequential Consistency Diagrams (1)

In other words: all processes see the same interleaving set of operations, regardless of what that interleaving is.





a) A sequentially consistent data-store – the “first” write occurred *after* the “second” on all replicas.
b) A data-store that is not sequentially consistent – it appears the writes have occurred in a non-sequential order, and this is NOT allowed.

Sequential Consistency Diagrams (2)

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

Three concurrently executing processes.

Sequential Consistency Diagrams (3)

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

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

Problem with Sequential Consistency

- With this consistency model, adjusting the protocol to favor reads over writes (or vice-versa) can have a devastating impact on performance (refer to the textbook for the gory details).
- For this reason, other weaker consistency models have been proposed and developed.
- Again, a relaxation of the rules allows for these weaker models to make sense.

Causal Consistency

- This model distinguishes between events that are "causally related" and those that are not.
- *If event B is caused or influenced by an earlier event A, then causal consistency requires that every other process see event A, then event B.*
- Operations that are not causally related are said to be *concurrent*.

More on Causal Consistency

- A causally consistent data-store obeys this 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 (i.e., by different processes).*

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

- This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store. Note: it is assumed that $W_2(x)b$ and $W_1(x)c$ are concurrent.

Another Causal Consistency Example

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)

a) Violation of causal-consistency – P2's write is related to P1's write due to the read on 'x' giving 'a' (all processes must see them in the same order).

b) A causally-consistent data-store: the read has been removed, so the 2 writes are now *concurrent*. The reads by P3 and P4 are now OK.

FIFO Consistency

- Defined as follows:
  - *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.*
- This is also called “PRAM Consistency” – Pipelined RAM.
- The attractive characteristic of FIFO is that is it 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.

FIFO Consistency Example (1)

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 studied so far would allow this sequence of events.

FIFO Consistency Example (2)

(a)	(b)	(c)
x = 1;	x = 1;	y = 1;
print (y, z);	y = 1;	print (x, z);
y = 1;	**print (x, z);**	z = 1;
print(x, z);	print (y, z);	**print (x, y);**
z = 1;	z = 1;	x = 1;
print (x, y);	print (x, y);	print (y, z);
Prints: 00	Prints: 10	Prints: 01

Statement execution as seen by the three processes from the similar previous slide. The statements in bold are the ones that generate the output shown.

FIFO Consistency Example (3)

Process P1	Process P2
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);

Two concurrent processes.

Introducing Weak Consistency

- Not all applications need to see all writes, let alone seeing them in the same order.
- This 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 to update all copies of the data-store.

Weak Consistency Properties

- The three properties of 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.

Weak Consistency: What It Means

- So …
- By doing a sync., a process can *force* the just written value out to all the other replicas.
- Also, by doing a sync., a process can be *sure* it's getting the most recently written value before it reads.
- In 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).

Weak Consistency Examples

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. 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'.

Introducing 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

- Defined as follows:
  - *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 need be.*
  - *When a “release” is done, protected data that have been changed are propagated out to the local copies of the data-store.*

Release Consistency Example

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

- 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).

Introducing Entry Consistency

- A different twist on things is “Entry Consistency”. Acquire and release are still used, and the data-store meets the following conditions:

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's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Entry Consistency: What It Means

- 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*

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	

- Locks associate with individual data items, as opposed to the entire data-store. Note: P2's read on 'y' returns NIL as no locks have been requested.

Summary of Consistency Models

Consistency	Description
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 same order. Accesses 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.

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after 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 it is entered.

(b)

a) Consistency models that do not use synchronization operations.

b) Models that do use synchronization operations. (These require additional programming constructs, and allow programmers to treat the data-store *as if it is sequentially consistent*, when in fact it is not. They "should" also offer the best performance).

Client-Centric Consistency Models

- The previously studied consistency models concern themselves with maintaining a consistent (globally accessible) data-store in the presence of concurrent read/write operations
- Another class of distributed data-store is that which is 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.

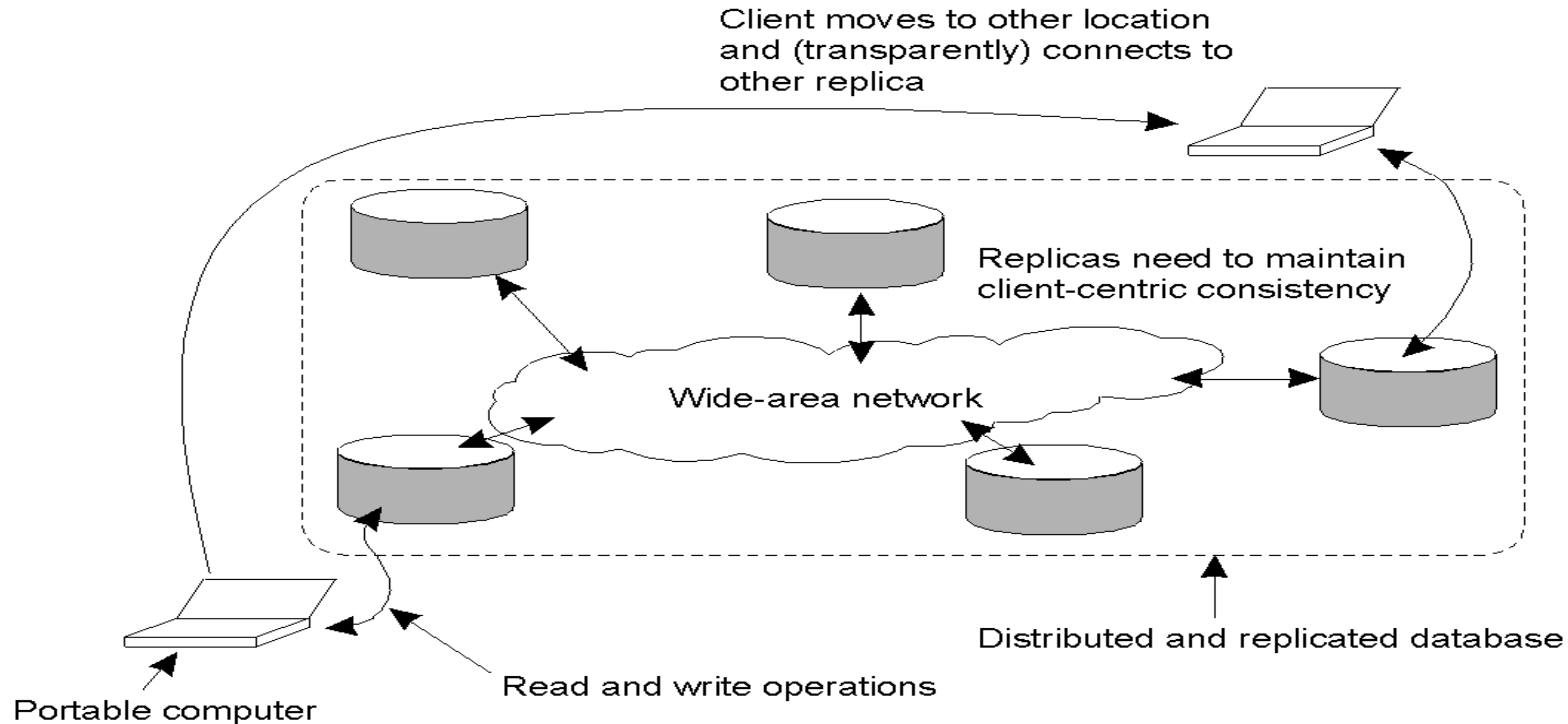
More Client-Centric Consistency

- How fast should updates (writes) be made available to read-only processes?
  - Think of most database systems: *mainly read*.
  - Think of the DNS: *write-write conflicts* do no occur.
  - Think of 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*.
- These systems all exhibit a high degree of acceptable inconsistency … with the *replicas* gradually becoming consistent

Toward Eventual Consistency

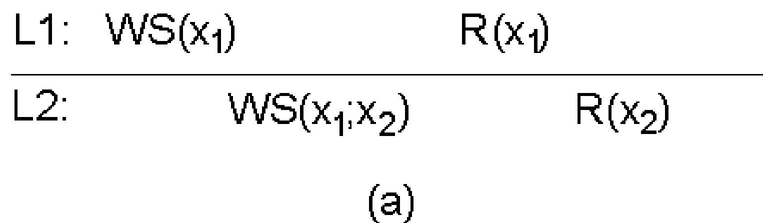
- The only requirement is that all replicas will *eventually* be the same.
- All updates must be guaranteed to propagate to all replicas … *eventually*!
- This works well if every client always updates the same replica.
- Things are a little difficult if the clients are *mobile*.

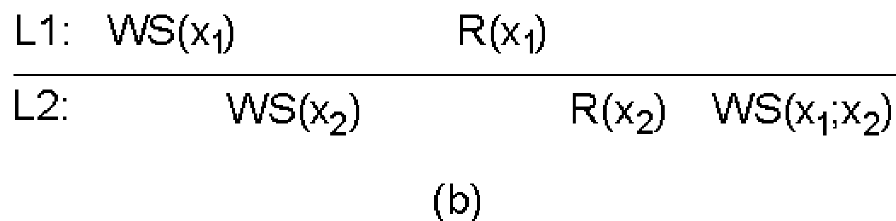
Eventual Consistency: Mobile Problems



- The principle of a mobile user accessing different replicas of a distributed database.
- When the system can guarantee that a single client sees accesses to the data-store in a consistent way, we then say that "client-centric consistency" holds.

Monotonic Reads



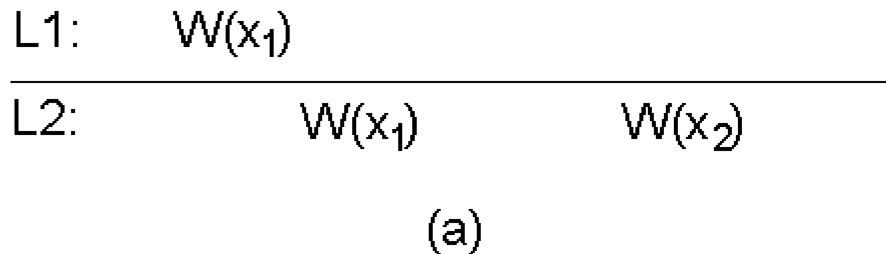


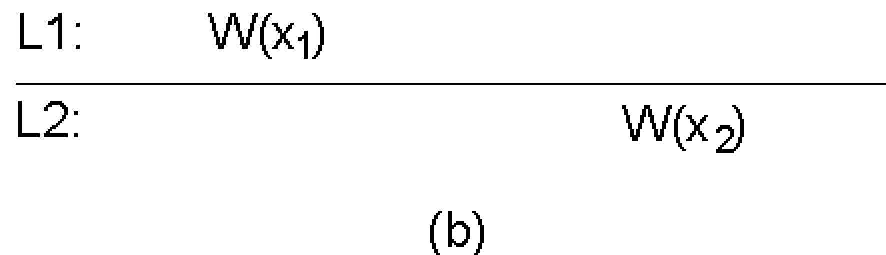
- The read operations performed by a single process *P* at two different local copies of the same data store.

a) A monotonic-read consistent data store

b) A data store that does not provide monotonic

Monotonic Writes



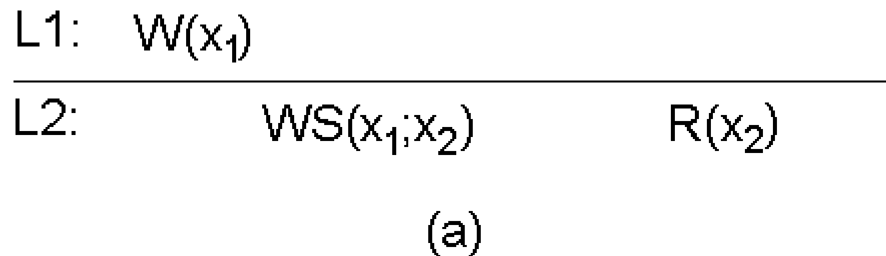


- The write operations performed by a single process *P* at two different local copies of the same data store

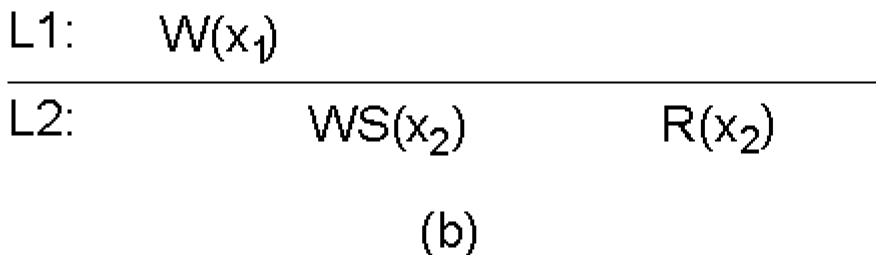
a) A monotonic-write consistent data store.

b) A data store that does not provide

Read Your Writes



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



b) A data store that does not.

Writes Follow Reads

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

An Example: The Bayou System

- The Bayou System implements 4 models of *Client-Centric Consistency*:

1. Monotonic-Read Consistency
2. Monotonic-Write Consistency
3. Read-Your-Writes Consistency
4. Writes-Follow-Reads Consistency

More on Bayou (1)

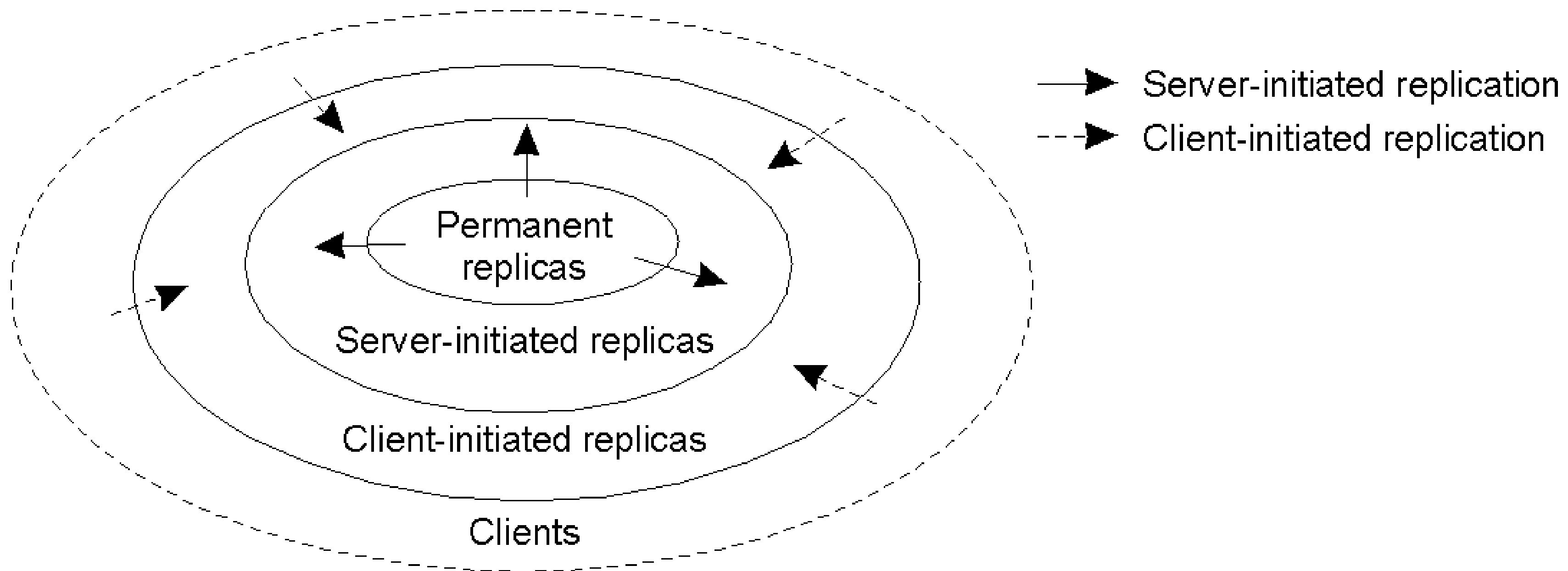
- Monotonic Reads: *if a process reads the value of a data item ‘x’, any successive read operation on ‘x’ by that process will always return that same value or a more recent value.*
- Monotonic Writes: *A write operation by a process on a data item ‘x’ is completed before any successive write operation on ‘x’ by the same process.*

More on Bayou (2)

- Read Your Writes: *The effect of a write operation by a process on data item ‘x’ will always be seen by a successive read operation on ‘x’ by the same process.*
- Writes Follow Reads: *A write operation by a process on a data item ‘x’ following a previous read operation on ‘x’ by the same process, is guaranteed to take place on the same or a more recent value of ‘x’ that was read.*

Distribution Protocols

- *Regardless of which consistency model is chosen, we need to decide* ***where***, ***when*** and ***by whom*** copies of the data-store are

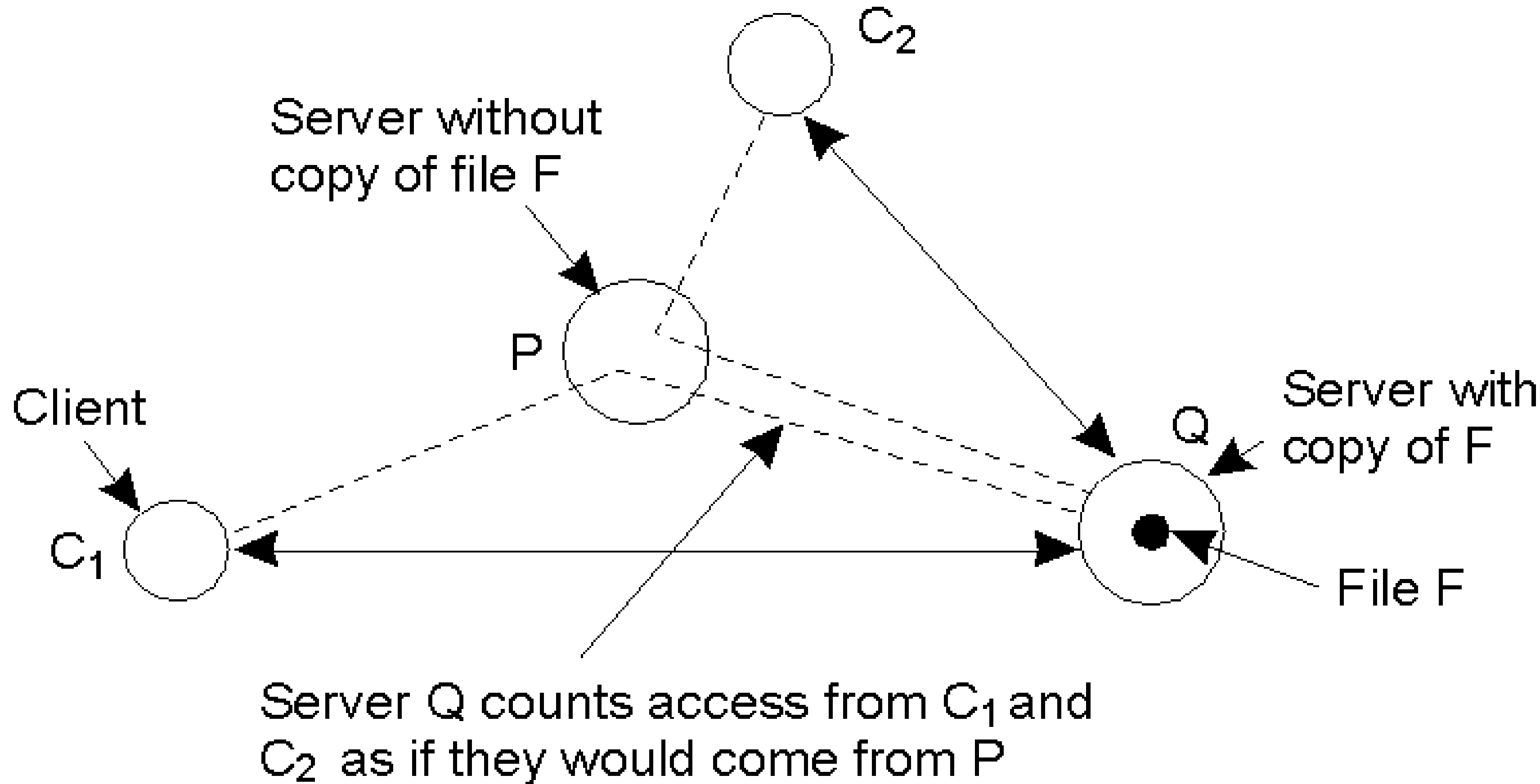


Replica Placement Types

- There are three types of replica:

1. ***Permanent replicas***: tend to be small in number, organized as COWs (Clusters of Workstations) or mirrored systems.
2. ***Server-initiated replicas***: used to enhance performance at the initiation of the owner of the data-store. Typically used by web hosting companies to geographically locate replicas close
to where they are needed most. (Often referred to
as "push caches").
3. ***Client-initiated replicas***: created as a result of client requests – think of browser caches. Works well assuming, of course, that the

Server-Initiated Replicas



Counting access requests from different clients.

Update Propagation

- When a client initiates an update to a distributed data-store, what gets propagated?
- There are three possibilities:

1. Propagate *notification* of the update to the other replicas – this is an “invalidation protocol” which indicates that the replica’s data is no longer up-to-date. Can work well when there’s many writes.
2. Transfer the *data* from one replica to another – works well when there’s many reads.
3. Propagate the *update* to the other replicas – this is “active replication”, and shifts the workload to each of the replicas upon an “initial

Push vs. Pull Protocols

- Another design issue relates to whether or not the updates are *pushed* or *pulled*?

1. ***Push-based/Server-based Approach***: sent "automatically" by server, the client does *not* request the update. This approach is useful when a high degree of consistency is needed. Often used between permanent and server-initiated replicas.
2. ***Pull-based/Client-based Approach***: used by client caches (e.g., browsers), updates are requested by the client from the server. No request, no update!

Push vs. Pull Protocols: Trade Offs

Issue	Push-based	Pull-based
State on 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.

- A comparison between push-based and pull-based protocols in the case of *multiple client, single server systems*.
- Hybrid schemes are possible: e.g., “leases” – a promise from a server to push updates to a client for a period of time. Once the lease expires, the client reverts to a pull-based approach (until another lease is issued).

Epidemic Protocols

- This is an interesting class of protocols that can be used to implement *Eventual Consistency* (note: these protocols are used in Bayou).
- The main concern is the propagation of updates to all the replicas in *as few a number of messages as possible*.
- Of course, here we are spreading updates, not diseases!
- With this "update propagation model", the idea is to "infect" as many replicas as quickly as possible.

Epidemic Protocols: Terminology

- ***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

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

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. Oh dear.

ASSIGNMENT

- Q: Explain all consistency protocols in detail.