DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Lecturer: Dr. Faramarz Safi
Islamic Azad University
Najafabad Branch

Chapter 7
Consistency and Replication

Consistency and Replication

An important issue in distributed systems is the replication of data. Data are generally replicated to enhance **reliability** or improve **performance**. One of the major problems is keeping replicas consistent. Informally, this means that when one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same.

Main questions here are “why replication is useful? and how it relates to scalability? what consistency actually means”.

First of all, we start with concentrating on managing replicas, which takes into account not only the placement of replica servers, but also how content is distributed to these servers.

The second issue is how replicas are kept consistent. In most cases, applications require a strong form of consistency. Informally, this means that updates are to be propagated more or less immediately between replicas.

Introduction
Reasons for Replication

There are two primary reasons for replicating data including reliability and performance:

1) Data are replicated to increase the reliability of a system.

If a file system has been replicated it may be possible to continue working after one replica crashes by simply switching to one of the other replicas. Also, by maintaining multiple copies, it becomes possible to provide better protection against corrupted data. For example, imagine there are three copies of a file and every read and write operation is performed on each copy. We can safeguard ourselves against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one.

2) Replication for performance

- Scaling in numbers: Replication for performance is important when the distributed system needs to scale in numbers and geographical area. Scaling in numbers. occurs, for example, when an increasing number of processes needs to access data that are managed by a single server. In that case, performance can be improved by replicating the server and subsequently dividing the work.
- Scaling in geographical area: The basic idea is that by placing a copy of data in the proximity of the process using them, the time to access the data decreases. As a consequence, the performance as perceived by that process increases.

the benefits of replication for performance may be hard to evaluate. Although a client process may perceive better performance, it may also be the case that more network bandwidth is now consumed keeping all replicas up to date.

Introduction
Weak points

If replication helps to improve reliability and performance, who could be against it? Unfortunately, there is a price to be paid when data are replicated. The problems with replication are:

- ❑ Having multiple copies may lead to consistency problems. Whenever a copy is modified, that copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency. Exactly when and how those modifications need to be carried out determines the price of replication.

- ❑ Cost of increased bandwidth for maintaining replication.

Introduction

An example

To understand the problem, consider improving access times to Web pages. If no special measures are taken, fetching a page from a remote Web server may sometimes even take seconds to complete.

To improve performance, Web browsers often locally store a copy of a previously fetched Web page (i.e., they cache a Web page). If a user requires that page again, the browser automatically returns the local copy. The access time as perceived by the user is excellent. The problem is that if the page has been modified in the meantime, modifications will not have been propagated to cached copies, making those copies out-of-date.

One solution to the problem of returning a stale copy to the user is to forbid the browser to keep local copies in the first place, effectively letting the server be fully in charge of replication. However, this solution may still lead to poor access times if no replica is placed near the user.

Another solution is to let the Web server invalidate or update each cached copy, but this requires that the server keeps track of all caches and sending them messages. This, in turn, may degrade the overall performance of the server.

Introduction

Replication as a scaling technique

Scalability issues generally appear in the form of performance problems. Placing copies of data close to the processes using them can improve performance through reduction of access time and thus solve scalability problems.

A possible trade-off that needs to be made is that keeping copies up to date may require more network bandwidth. Consider a process P that accesses a local replica N times per second, whereas the replica itself is updated M times per second. Assume that an update completely refreshes the previous version of the local replica. If N «M, that is, the access-to-update ratio is very low, we have the situation where many updated versions of the local replica will never be accessed by P, rendering the network communication for those versions useless. In this case, it may have been better not to install a local replica close to P, or to apply a different strategy for updating the replica.

Introduction
Replication as a scaling technique

A more serious problem, however, is that keeping multiple copies consistent may itself be subject to serious scalability problems. Intuitively, a collection of copies is consistent when the copies are always the same. This means that a read operation performed at any copy will always return the same result. Consequently, when an update operation is performed on one copy, the update should be propagated to all copies before a subsequent operation takes place, no matter at which copy that operation is initiated or performed. This type of consistency is sometimes informally referred to as tight consistency or synchronous replication.

The key idea is that an update is performed at all copies as a single atomic operation, or transaction. we need to synchronize all replicas. this means that all replicas first need to reach agreement on when exactly an update is to be performed locally.

For example, replicas may need to decide on a global ordering of operations using Lamport timestamps, or let a coordinator assign such an order.

Introduction
Replication as a scaling technique

Replication and caching are widely used in scaling technique, but:

- Keeping replicas up to date needs networks use.
- Update needs to be atomic (transaction)
- Replicas need to be synchronized (time consuming)

Loose Consistency

In this case copies are not always the same everywhere.

Consistency Models

- DATA-CENTRIC CONSISTENCY MODELS
  - Continuous Consistency
  - Consistent Ordering of Operations

- CLIENT-CENTRIC CONSISTENCY MODELS
  - Eventual Consistency
  - Monotonic Reads
  - Monotonic Writes
  - Read Your Writes
  - Writes Follow Reads

Consistency Models

Traditionally, consistency has been discussed in the context of read and write operations on shared data, available by means of (distributed) shared memory. A (distributed) shared database, or a (distributed) file system.

Here, we use the broader term **data store**. A data store may be physically distributed across multiple machines. Each process that can access data from the store is assumed to have a local (or nearby) copy available of the entire store. Write operations are propagated to the other copies, as shown in Fig. 7-1. A data operation is classified as a write operation when it changes the data, and is otherwise classified as a read operation.

A **consistency model** is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly.

Normally, a process that performs a read operation on a data item, expects the operation to return a value that shows the results of the last write operation on that data. In the absence of a global clock, it is difficult to define precisely which write operation is the last one. As an alternative, we need to provide other definitions, leading to a range of consistency models. Each model effectively restricts the values that a read operation on a data item can return.

Consistency Models

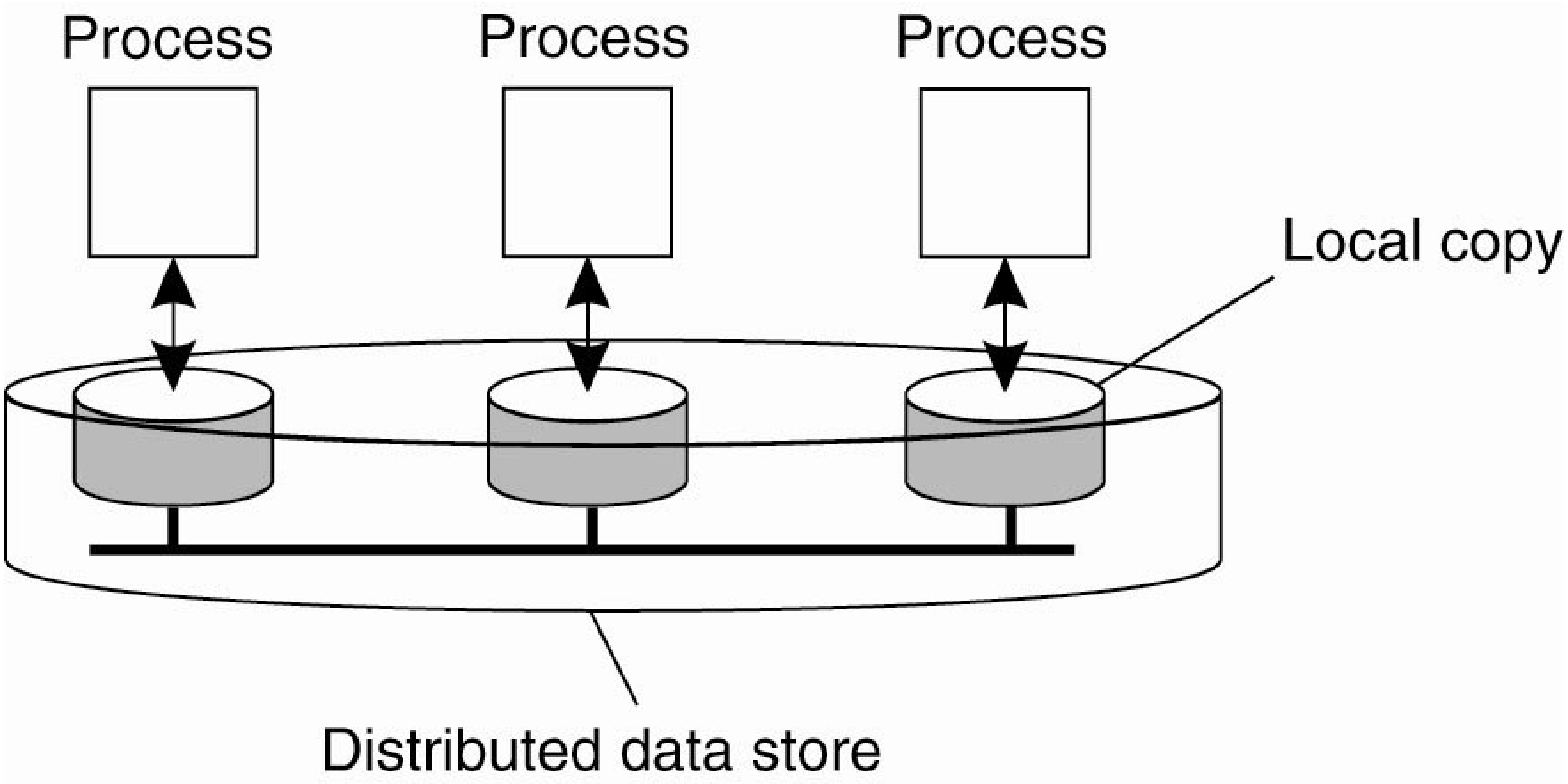


Figure 7-1. The general organization of a logical data store, physically distributed and replicated across multiple processes.

Data-centric Consistency Models

Replicating data poses consistency problems that cannot be solved efficiently in a general way. Only if we loosen consistency can there be hope for attaining efficient solutions. Unfortunately, there are also no general rules for loosening consistency: exactly what can be tolerated is highly dependent on applications.

There are different ways for applications to specify what inconsistencies they can tolerate. **Yu and Vahdat (2002)** take a general approach by distinguishing **three independent axes** for defining inconsistencies:

- deviation in numerical values between replicas,
- deviation in staleness between replicas,
- deviation with respect to the ordering of update operations.

These deviations are referred to as forming **continuous consistency** ranges.

Data-centric Consistency Models
Numerical Deviations

Measuring inconsistency in terms of numerical deviations can be used by applications for which the data have numerical semantics.

One obvious example is the replication of records containing stock market prices. In this case, an application may specify that two copies should not deviate more than $0.02, which would be an *absolute numerical deviation.*

Alternatively, a relative numerical deviation could be specified, stating that two copies should differ by no more than, for example, 0.5%.

In both cases, we would see that if a stock goes up (and one of the replicas is immediately updated) without violating the specified numerical deviations, replicas would still be considered to be mutually consistent.

Numerical deviation can also be understood in terms of the number of updates that have been applied to a given replica, but have not yet been seen by others. For example, a Web cache may not have seen a batch of operations carried out by a Web server. In this case, the associated deviation in the *value is also referred to* as its ***weight**.*

Data-centric Consistency Models
Staleness Deviations

Staleness deviations relate to the last time a replica was updated. For some applications, it can be tolerated that a replica provides old data as long as it is not *too old. For example, weather reports typically stay reasonably accurate over* some time, say a few hours. In such cases, a main server may receive timely updates, but may decide to propagate updates to the replicas only once in a while.

Data-centric Consistency Models
Ordering of Updates

Finally, there are classes of applications in which the ordering of updates are allowed to be different at the various replicas, as long as the differences remain bounded.

One way of looking at these updates is that they are applied tentatively to a local copy, awaiting global agreement from all replicas.

As a consequence, some updates may need to be rolled back and applied in a different order before becoming permanent.

Intuitively, ordering deviations are much harder to grasp than the other two consistency metrics.

The Notion of a Conit

Yu and Vahdat introduce a consistency unit, abbreviated to **Conit**. A conit specifies the unit over which consistency is to be measured.

For example, in our stock-exchange example, a conit could be defined as a record representing a single stock. Another example is an individual weather report.

The Notion of a Conit
An Example

To give an example of a Conit, and at the same time illustrate numerical and ordering deviations, consider the two replicas as shown in Fig. 7-2.

In this example we see two replicas that operate on a conit containing the data items *x and y. Both variables are assumed to have been initialized to 0.*

Replica A received the operation *5,B : x := x +2* from replica B and has made it permanent (i.e., the operation has been committed at A and cannot be rolled back). Replica A has three tentative update operations: 8,A, 12,A, and 14,A, which brings its **ordering deviation to 3**. Also note that due to the last operation *14,A, A's vector clock becomes (15,5).*

The Notion of a Conit
An Example

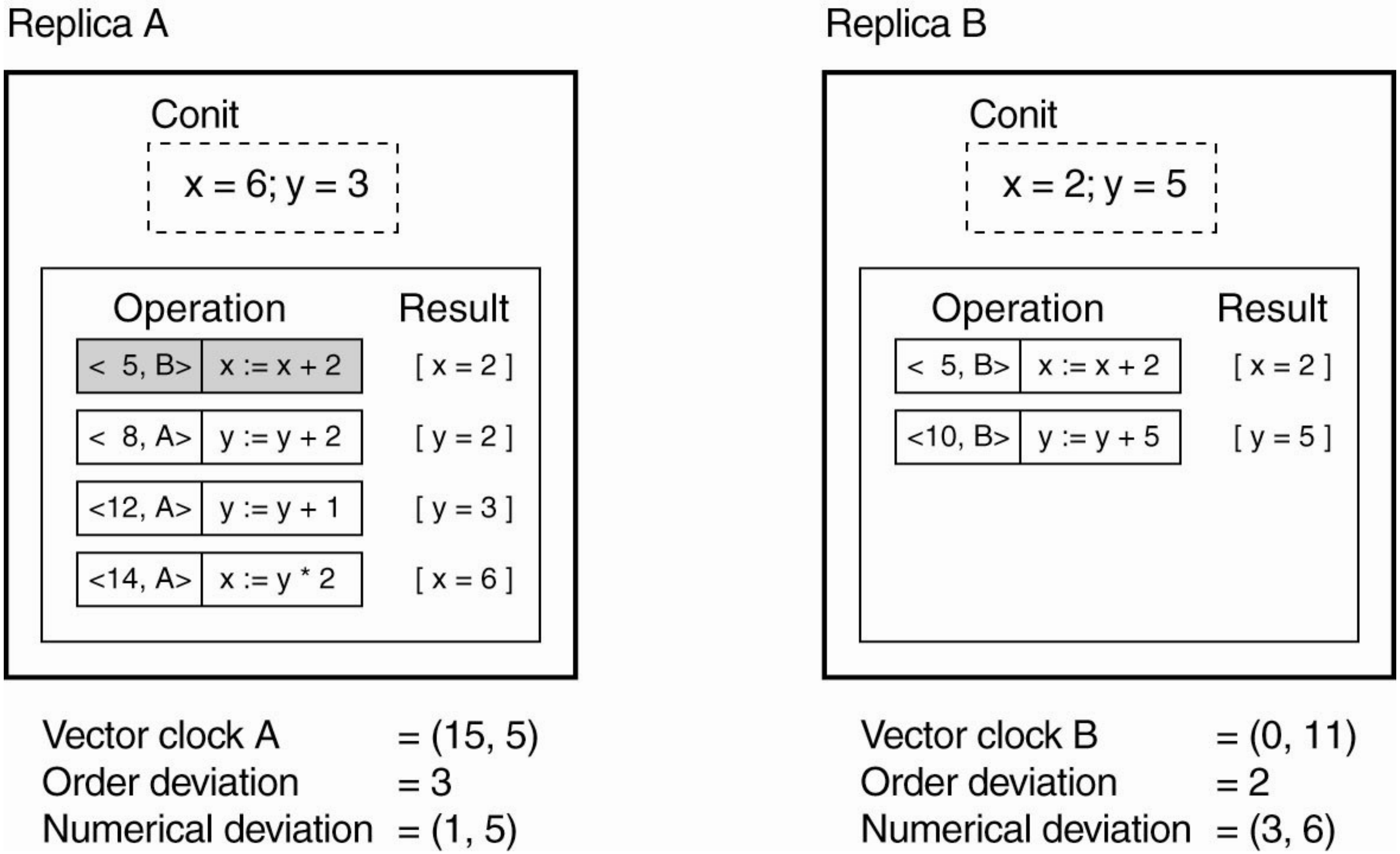
The only operation from B that A has not yet seen is 10,B, bringing its **numerical deviation with respect to operations to 1**.

In this example, the **weight** of this deviation can be expressed as the maximum difference between the (committed) values of x and y at A, and the result from operations at B not seen by A. The committed value at A is (x,y) = (2,0), whereas the-for A unseen-operation at B yields a difference of y = 5. **weight = max[|2-2|,|5-0|]=5**

A similar reasoning shows that B has two tentative update operations: 5,B and 10,B , which means it has an **ordering deviation of 2**. Because B has not yet seen a single operation from A, its vector clock becomes (0, 11). The **numerical deviation is 3** with a total **weight of 6**. This last value comes from the fact B's committed value is (x,y) = (0,0), whereas the tentative operations at A will already bring x to 6.

weight=max[|0-6|,|0-3|]=6.

Continuous Consistency an example



Replica A

Conit

x = 6; y = 3

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)

Replica B

Conit

x = 2; y = 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)

Figure 7-2. An example of keeping track of consistency deviations [adapted from (Yu and Vahdat, 2002)].

Continuous Consistency
Coarse granularity

Note that there is a trade-off between maintaining fine-grained and coarse grained conits. If a conit represents a lot of data, such as a complete database, then updates are aggregated for all the data in the conit. As a consequence, this may bring replicas sooner in an inconsistent state.

For example, assume that in Fig.7-3 two replicas may differ in no more than one outstanding update. In that case, when the data items in Fig. 7-3(a) have each been updated once at the first replica, the second one will need to be updated as well.

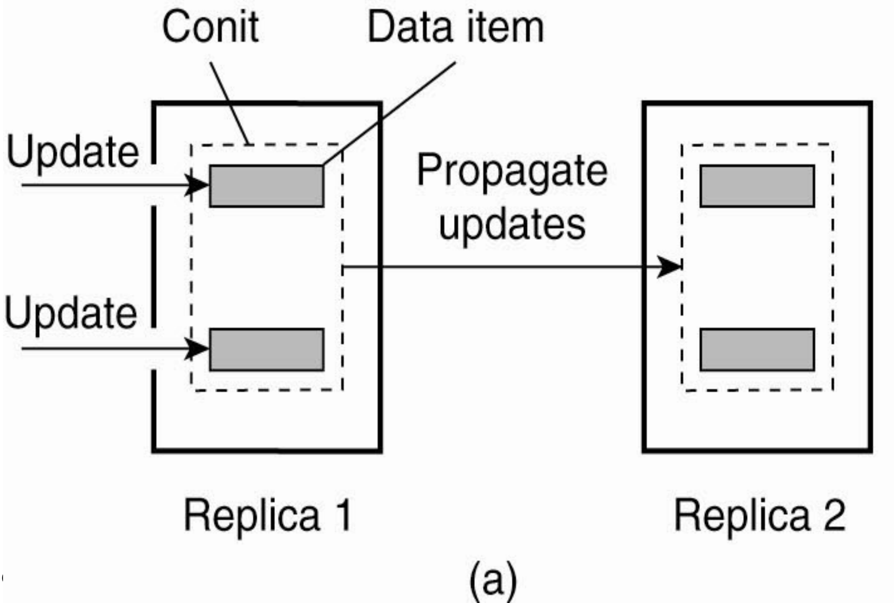


Figure 7-3. Choosing the appropriate granularity for a conit. (a) Two updates lead to update propagation.

Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2

Continuous Consistency
Fine granularity

This is not the case when choosing a smaller conit, as shown in Fig. 7-3(b). There, the replicas are still considered to be up to date. This problem is particularly important when the data items contained in a conit are used completely independently, in which case they are said to falsely share the conit.

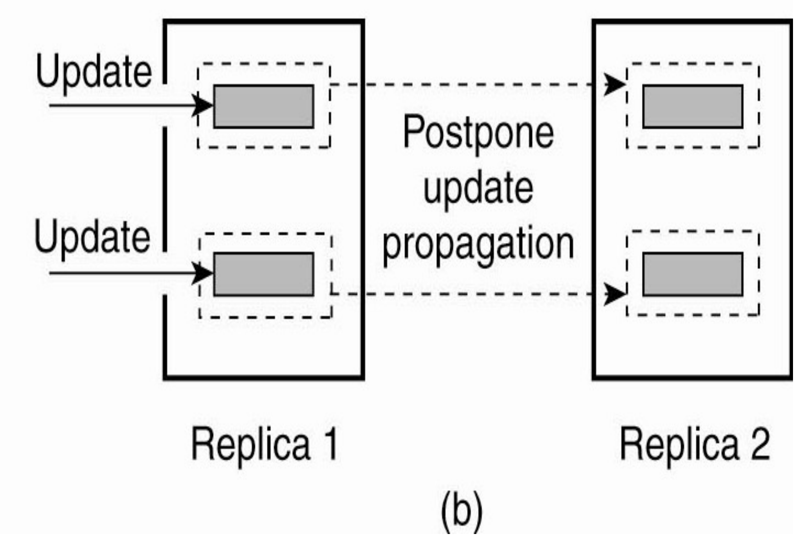


Figure 7-3. Choosing the appropriate granularity for a conit.

(b) No update propagation is needed (yet).

Continues Consistency
Coarse and Fine Granularity

Unfortunately, making conits very small is not a good idea:

1) the total number of conits that need to be managed grows as well.
2) This overhead, in turn, may adversely affect overall performance.

There are two important issues that need to be dealt with:
First, in order to enforce consistency we need to have protocols.
Second, program developers must specify the consistency requirements for their applications.

Practice indicates that obtaining such requirements may be extremely difficult. Programmers are generally not used to handling replication, let alone understanding what it means to provide detailed information on consistency. Therefore, it is mandatory that there are simple and easy-to-understand programming interfaces.

Continues Consistency
As a toolkit

Continuous consistency can be implemented as a toolkit which appears to programmers as just another library that they link with their applications. A conit is simply declared alongside an update of a data item. For example, the fragment of pseudocode:

```
DependsOnConit(ConitQ, 4, 0, 60);
read message m from head of queue Q;
```

In this case, the call to DependsOnConitO specifies that the numerical deviation, ordering deviation, and staleness should be limited to the values 4, 0, and 60 (seconds), respectively. This can be interpreted as that there should be at most 4 unseen update operations at other replicas, there should be no tentative local updates, and the local copy of Q should have been checked for staleness no more than 60 seconds ago.

If these requirements are not fulfilled, the underlying middle ware will attempt to bring the local copy of Q to a state such that the read operation can be carried out

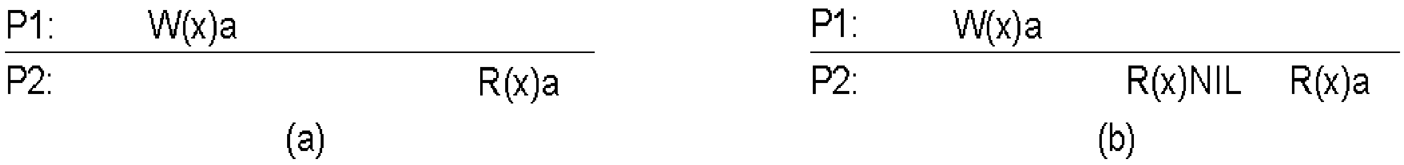
Consistent Ordering of Operations

Besides continuous consistency, there is a huge body of work on data-centric consistency models from the past decades. An important class of models comes from the field of concurrent programming.

Researchers have sought to express the semantics of concurrent accesses when shared resources are replicated. This has led to at least one important consistency model that is widely used. In the following, we concentrate on what is known as **Strict consistency**, **sequential consistency**, **Linearizability**, **causal consistency, FIFO, Weak, Release, and Entry**.

Consistent Ordering of Operations
Strict Consistency

*Any **read** on a **data item x** returns a value corresponding to the result of the **most recent write**. T*wo operations in the same time interval are said to be in conflict if they operate on the same data and one of them is a write operation.



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.

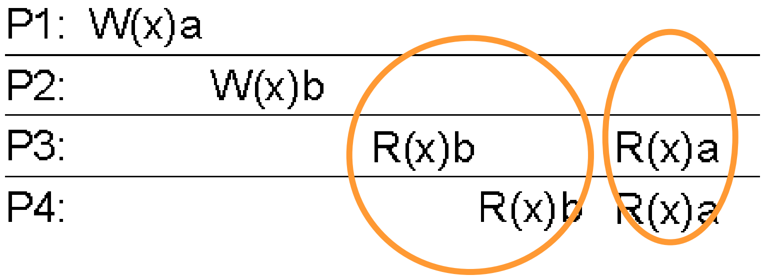
a) A strictly consistent store.

b) A store that is not strictly consistent.

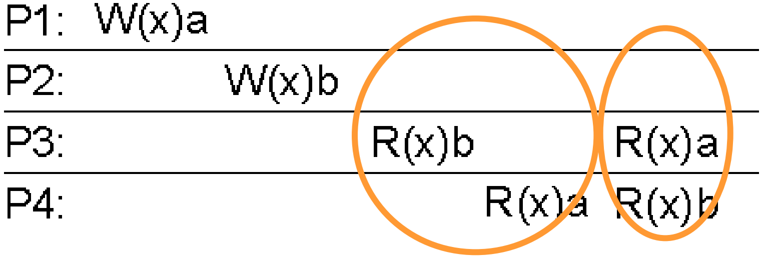
Strict consistency is the ideal model but it is impossible to implement in a distributed system. It is based on *absolute global time or a global agreement on commitment of changes*..

Consistent Ordering of Operations
Sequential Consistency

Sequential Consistency it is a weaker consistency model than strict 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 some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. All processes see the same interleaving of operations



(a)



(b)

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

No reference to the timing of the operations.

Consistent Ordering of Operations
Linearizability Consistency

Linearizability is weaker than strict consistency but stronger than 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 some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. In addition, if $ts_{OP1}(x) < ts_{OP2}(y)$, then operation OP1(x) should precede OP2(y) in this sequence. Operations receive a timestamp using a global clock, but with finite precision.

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

Example : three concurrently executing processes; (x, y, z) are data store items

Various (90) interleaved execution sequences are possible

Linearizability and Sequential Consistency

▲ 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)

Not all signature pattern are allowed : 000000 not permitted, 001001 not permitted.

Constraints:

- ❑ Program order must be maintained.
- ❑ Data coherence must be respected.

Data coherence : any read must return the most recently written value of the data (relatively to the single data item, without regard to other data)

Causal Consistency (1)

When there is a read *followed* by a write, the two events are *potentially* causally related. Operation not causally related are said concurrent.

The same idea in multi-processors:

writes that are potentially causally related must be seen by all processors in the same order. Concurrent writes may be seen in a different order on different machines:

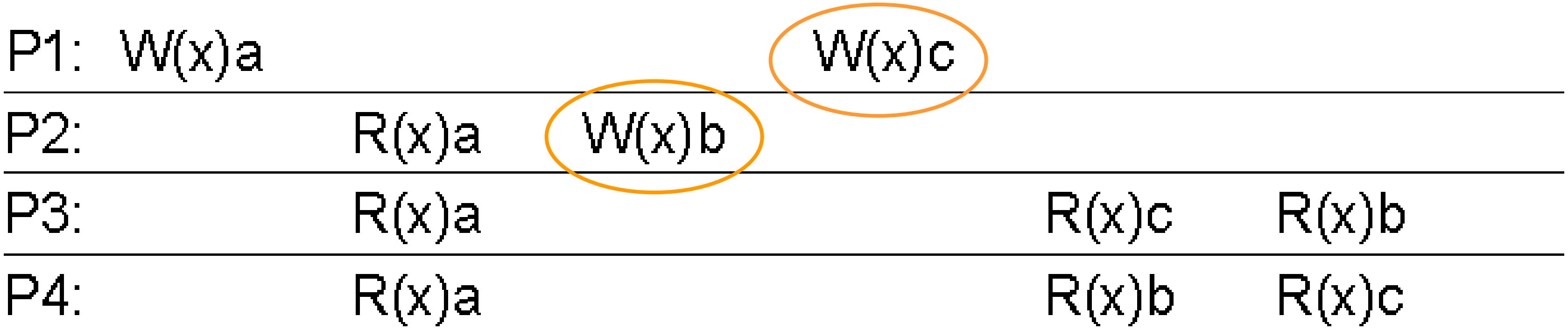
- causally related writes: the write comes after a read that returned the value of the other write

Necessary 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.

We already came across causality when discussing vector timestamps in the previous chapter. If event b is caused or influenced by an earlier event a, causality requires that everyone else first see a, then see b.

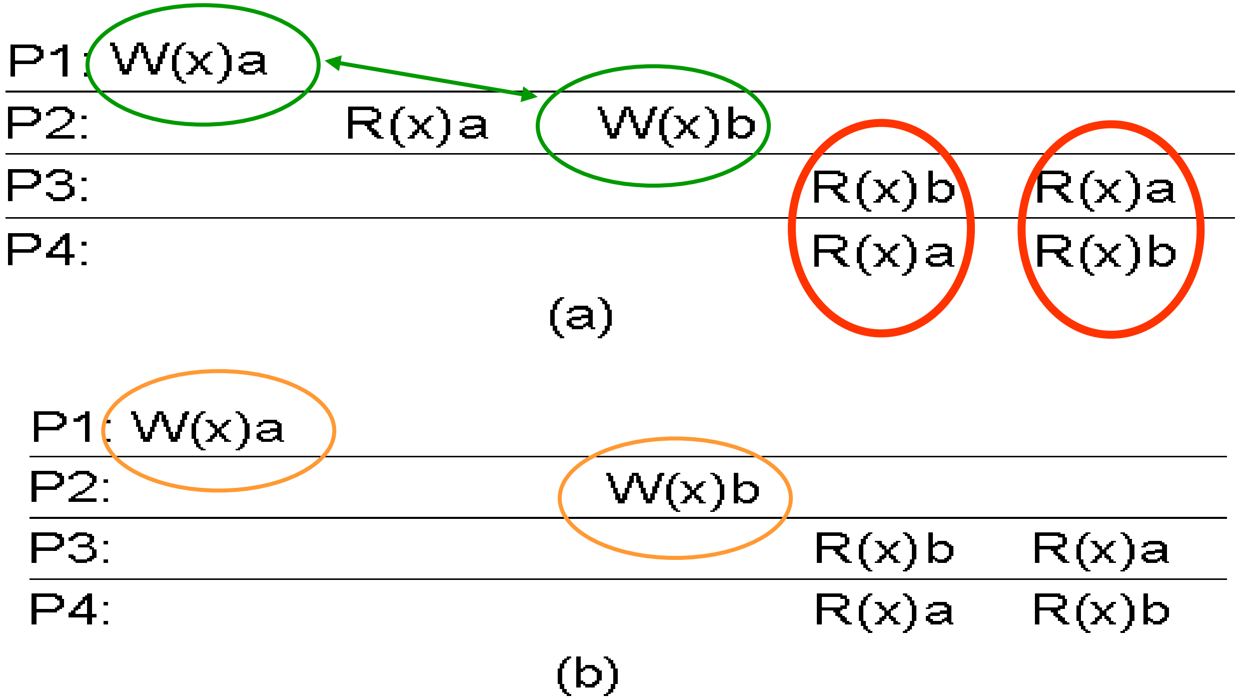
Causal Consistency (2)



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 that the writes $W_2(x)b$ and $W_1(x)c$ are concurrent. Causal consistency requires keeping tracks of which processes have seen which writes.

Causal Consistency (3)



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) A violation of a casually-consistent store. W_2(x)b may be related to W_1(x)a *because the b may be a result of a computation involving* the value read by *R_2 (x)a. The two writes are causally related, so all processes* must see them in the same order.

b) A correct sequence of events in a casually-consistent store. W_1(x)a and W_2(x)b are concurrent and no need to be globally ordered. This situation would not be acceptable by sequentially consistent store.

FIFO Consistency (1)

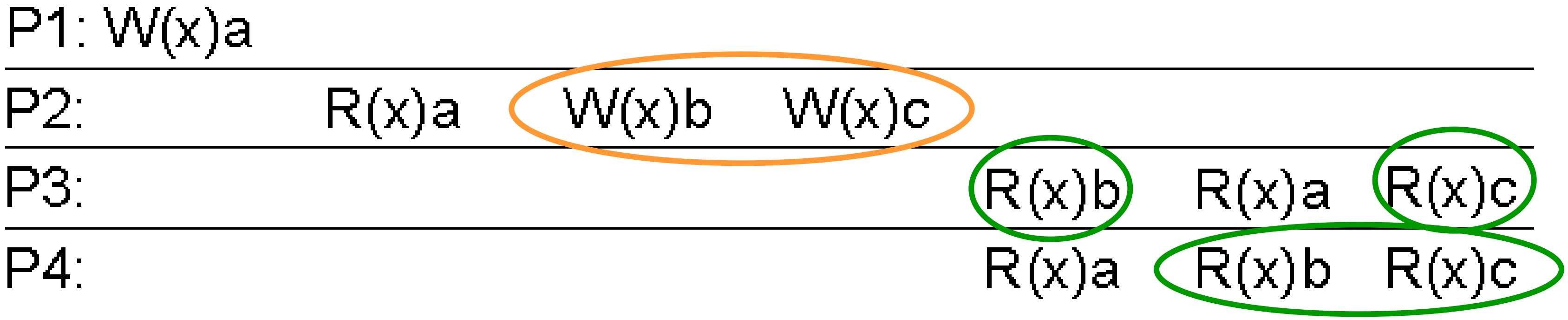
Relaxing consistency requirements we drop causality

Necessary Condition:

> ***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.***

All writes generated by different processes are considered concurrent. It is easy to implement

FIFO Consistency (2)



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 events of FIFO consistency. It is not valid for causal consistency

FIFO Consistency (3)

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

x = 1;

print (y, z);

y = 1;

print(x, z);

z = 1;

print (x, y);

Prints: 00

(a)

x = 1;

y = 1;

print(x, z);

print (y, z);

z = 1;

print (x, y);

Prints: 10

(b)

y = 1;

print (x, z);

z = 1;

print (x, y);

x = 1;

print (y, z);

Prints: 01

(c)

The statements in bold are the ones that generate the output shown. Their concatenated output is 001001, that is incompatible with sequential consistency

FIFO Consistency(4)

The idea being illustrated here is that FIFO consistency would allow the three processes to have different views. Specifically,
(a) above is how P1 *might* see things;
(b) is how P2 may see it, and
(c) is how P3 may see it.
This is, of course, because P1 executes the "**print(y,z);**" statement, P2 executes "**print(x,z);**", and P3 executes "**print(x,y);**".

In the figure, the statements in bold are the ones that generate the output shown. Their concatenated output is 001001, that is incompatible with sequential consistency.
It stands to reason that the actions of local statements will always be observed before statements executed by other processes that may execute on other machines. The point here is that in sequential consistency, all processes must agree on the order of statement execution, while with FIFO consistency, that is not necessary.

FIFO Consistency (5)

Different processes can see the operations in different order

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

The book states that intuitively, "one might naively expect one of three possible outcomes: P1 is killed, P2 is killed, or neither is killed".

Here is something in which I would disagree. My opinion is that the most intuitive possibility is exactly what could happen with FIFO consistency: both processes will be killed.

This would happen if P1 and P2 begin execution simultaneously, because P1 will set x to 1, observe that y is 0, and send a kill signal to P2. Simultaneously, P2 sets y to 1, observes that x is 0, and sends a kill signal to P1.

Regardless of your intuition on this one, the point is that sequential consistency would prevent both processes from being killed.

Grouping Operations
Weak Consistency

We can release the requirements of writes within the same process seen in order everywhere introducing a synchronization variable.

A synchronization operation synchronize all local copies of the data store.

Weak consistency uses synchronization variables to propagate writes to and from a machine at appropriate points. Properties of weak consistency:

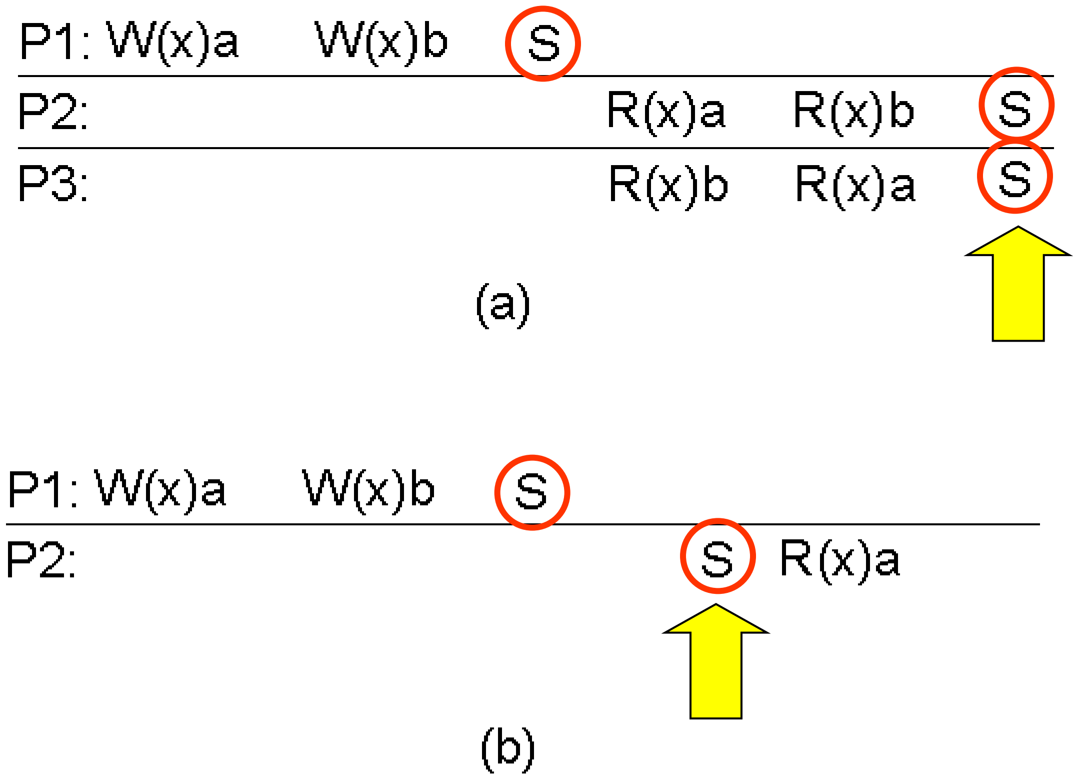
- ***Accesses to synchronization variables associated with a data store are sequentially consistent.***
- ***No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.***
- ***No read or write operation on data items are allowed to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

The same idea is used in multi processors:

- accessing a synchronization variable "flushes the pipeline".
- at a synchronization point, all processors have consistent versions of data.

It forces consistency on a **group** of operations, not on individual write and read. It limits the time when consistency holds, not the form of consistency.

Grouping Operations
Weak Consistency



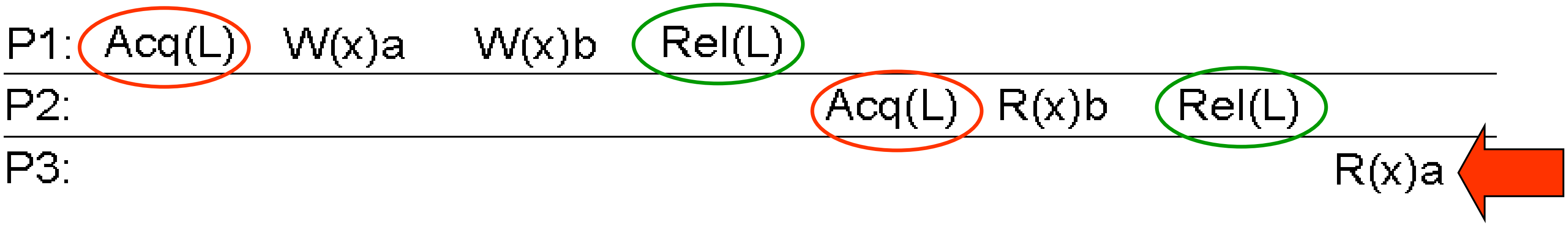
convention: S means access to synchronization variable

a) A valid sequence of events for weak consistency.

b) Tanenbaum says the second is not weakly consistent. Do you agree? (What is the order of synchronizations? How do the rules prevent this execution?).

Grouping Operations
Release Consistency

If it is possible to know the difference between entering a critical region or leaving it, a more efficient implementation might be possible. To do that, two kinds of synchronization variables are needed. Release consistency : acquire operation to tell that a critical region is being entered; release operation when a critical region is to be exited



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.

Shared data kept consistent are called protected

Grouping Operations
Release Consistency

The same idea in multi-processors, Release consistency is like weak consistency, but there are two operations “lock” and “unlock” for synchronization:

- (“acquire/release” are the conventional names)
- doing a “lock” means that writes on other processors to protected variables will be known
- doing an “unlock” means that writes to protected variables are exported
- and will be seen by other machines when they do a “lock” (lazy release consistency) or immediately (eager release consistency)

Grouping Operations
Release Consistency

Rules:

- ***Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.***

- ***Before a release is allowed to be performed, all previous reads and writes by the process must have completed.***

- ***Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).***

Explicit acquire and release calls are required

Grouping Operations
Entry Consistency

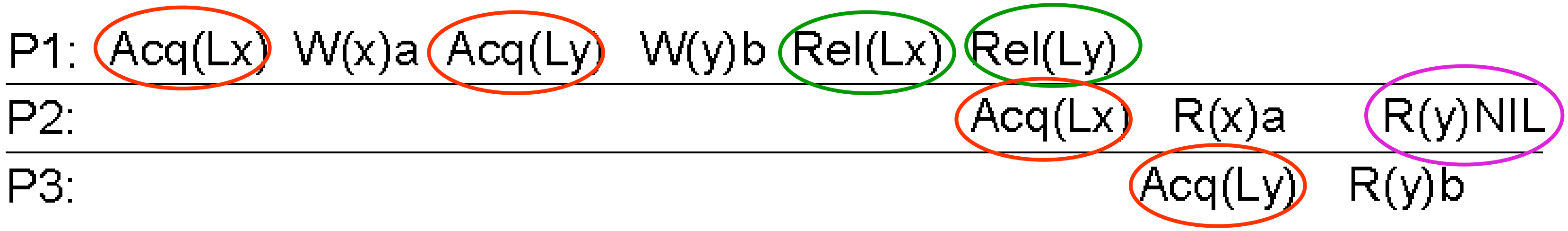
Conditions:

- ***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 non-exclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.***

Grouping Operations
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

A valid event sequence for entry consistency. Lock are associated with each data item

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 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

(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 vs. Coherence

The models we have discussed so far all deal with the fact that 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.

Where data consistency is concerned with a set of data items, coherence models describe what can be expected to only a single data item. In this case, we assume that a data item is replicated at several places; it is said to be coherent when the various copies abide to the rules as defined by its associated coherence model.

A popular model is that of sequential consistency, but now applied to only a single data item. In effect, it means that in the case of concurrent writes, all processes will eventually see the same order of updates taking place.

Client Centric Consistency Models

The consistency models described in the previous section aim at providing a system wide consistent view on a data store. An important assumption is that concurrent processes may be simultaneously updating the data store, and that it is necessary to provide consistency in the face of such concurrency.

Being able to handle-concurrent operations on shared data while maintaining sequential consistency is fundamental to distributed systems. For performance reasons, sequential consistency may possibly be guaranteed only when processes use synchronization mechanisms such as transactions or locks.

we take a look at a special class of distributed data stores. The data stores we consider are characterized by **the lack of simultaneous updates, or when such updates happen, they can easily be resolved**.

Most operations involve reading data. By introducing special client-centric consistency models, it turns out that many inconsistencies can be hidden in a relatively cheap way.

Client Centric Consistency Modes
Examples

Example-1: in many database systems, most processes hardly ever perform update operations; they mostly read data from the database. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only reading processes.

Example-2: consider a worldwide naming system such as DNS. The DNS name space is partitioned into domains, where each domain is assigned to a naming authority, which acts as owner of that domain. Only that authority is allowed to update its part of the name space.

Consequently, conflicts resulting from two operations that both want to perform an update on the same data (i.e., **write-write conflicts)**, never occur. The only situation that needs to be handled are **read-write** conflicts. As it turns out, it is often acceptable to propagate an update in a **lazy fashion**, meaning that a reading process will see an update only after some time has passed since the update took place.

Client Centric Consistency Modes Examples

Example-3: is the World Wide Web. In virtually all cases, Web pages are updated by a single authority, such as a webmaster or the actual owner of the page. There are normally **no write-write conflicts** to resolve. On the other hand, to improve efficiency, browsers and Web proxies are often configured to keep a fetched page in a local cache and to return that page upon the next request.

These examples can be viewed as cases of (large-scale) distributed and replicated databases that tolerate a relatively high degree of inconsistency. They have in common that if no updates take place for a long time, all replicas will gradually become consistent. This form of consistency is called **eventual consistency.**

Client Centric Consistency
Eventual Consistency

In many cases concurrency appears only in restricted form.

In many applications most processes only read data

Some degrees of inconsistency can be tolerate

In some cases if for a long time no update takes place all replicas gradually become consistent

⇓

Eventual consistency

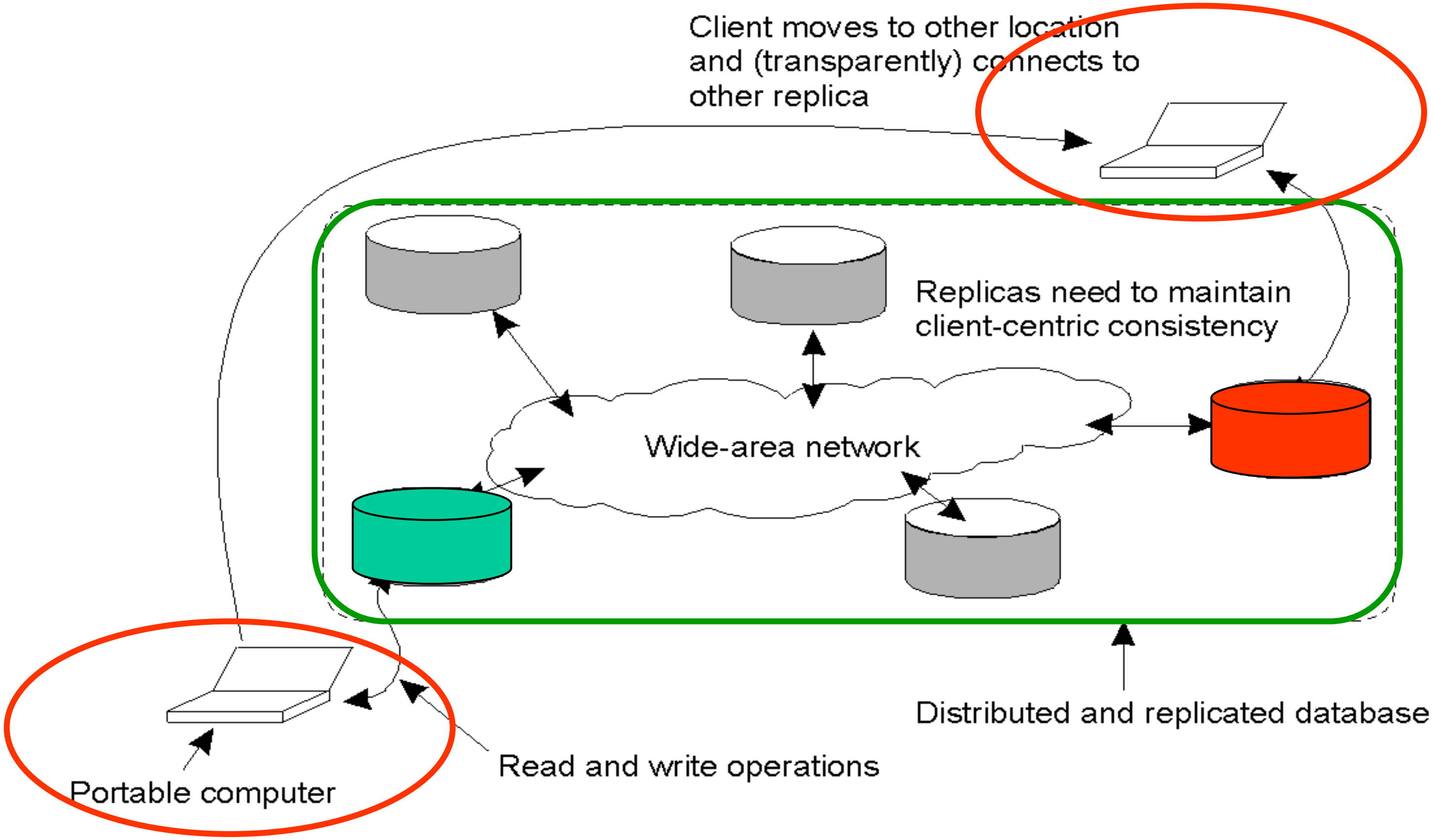
Client Centric Consistency
Eventual Consistency

Eventual consistency essentially requires only that updates are guaranteed to propagate to all replicas. Write-write conflicts are often relatively easy to solve when assuming that only a small group of processes can perform updates. Eventual consistency is therefore often cheap to implement.

However, problems arise when different replicas are accessed over a short period of time. This is best illustrated by considering a mobile user accessing a distributed database, as shown in Fig. 7-11

Client Centric Consistency
Eventual Consistency

Eventual consistency for replicated data is fine if clients always access the same replica. **Client centric** consistency provides consistency guarantees for a single client with respect to the data stored by that client


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

A mobile user accessing different replicas of a distributed database has problems with eventual consistency.

Client centric models

Clients access distributed data store using, generally, the local copy. Updates are **eventually** propagated to other copies.

- **Monotonic read**

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.

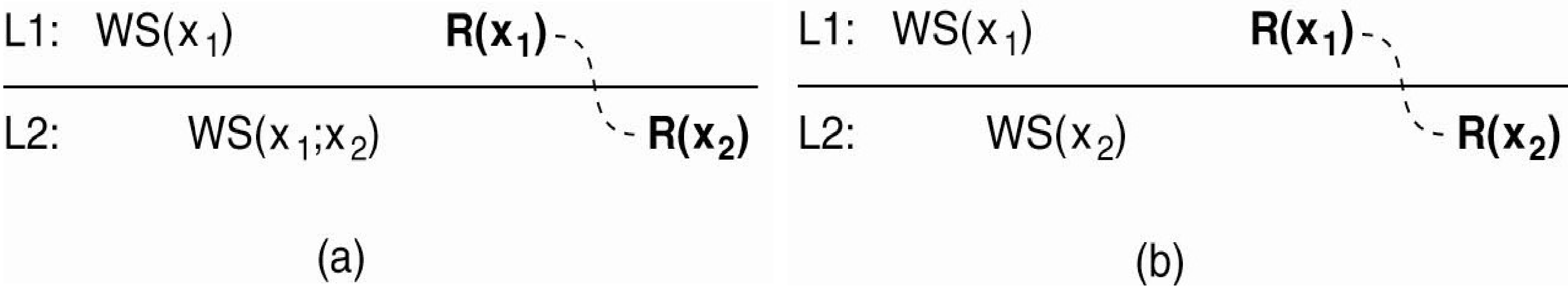
In other words, monotonic-read consistency guarantees that if a process has seen a value of *x at time t, it will never see an older version of x at a later time.*

As an example where monotonic reads are useful, consider a distributed email database. In such a database, each user's mailbox may be distributed and replicated across multiple machines. Mail can be inserted in a mailbox at any location. However, updates are propagated in a lazy (i.e., on demand) fashion. Only when a copy needs certain data for consistency are those data propagated to that copy.

*Suppose a user reads his mail in San Francisco. Assume that only reading mail does not affect the mailbox, that is, messages are not removed, stored in subdirectories, or even tagged as having already been read, and so on. When the user later flies to New York and opens his mailbox again, **monotonic-read consistency** guarantees that the messages that were in the mailbox in San Francisco will also be in the mailbox when it is opened in New York.*

Monotonic Reads

two different local copies of the data store are shown, *L1,* and *L2.* we are interested in the operations carried out by a single process P. These specific operations are shown in boldface are connected by a dashed line representing the order in which they are carried out by P.


L1: $WS(x_1)$ $\mathbf{R(x_1)}$
L2: $WS(x_1;x_2)$ $\mathbf{R(x_2)}$
(a)
L1: $WS(x_1)$ $\mathbf{R(x_1)}$
L2: $WS(x_2)$ $\mathbf{R(x_2)}$
(b)

To guarantee monotonic-read consistency, all operations in WS(x1) should have been propagated to L2 before the second read operation takes place. In other words, we need to know for sure that WS(x1) is part of WS(x2) which is expressed as WS(x1;X2).

(a) A monotonic-read consistent data store.

(b) A data store that does not provide monotonic reads.

Monotonic Writes

- **Monotonic write**

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

Thus 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. In other words, 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.

Note that monotonic-write consistency resembles data-centric FIFO consistency. The essence of FIFO consistency is that write operations by the same process are performed in the correct order everywhere. This ordering constraint also applies to monotonic writes, except that we are now considering consistency only for **a single process** instead of for a collection of concurrent processes.

Monotonic Writes

To ensure monotonic-write consistency, it is necessary that the previous write operation at L1 has already been propagated to L2. This explains operation W(Xl) at L2, and why it takes place before W(X2)·

In other words, 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.

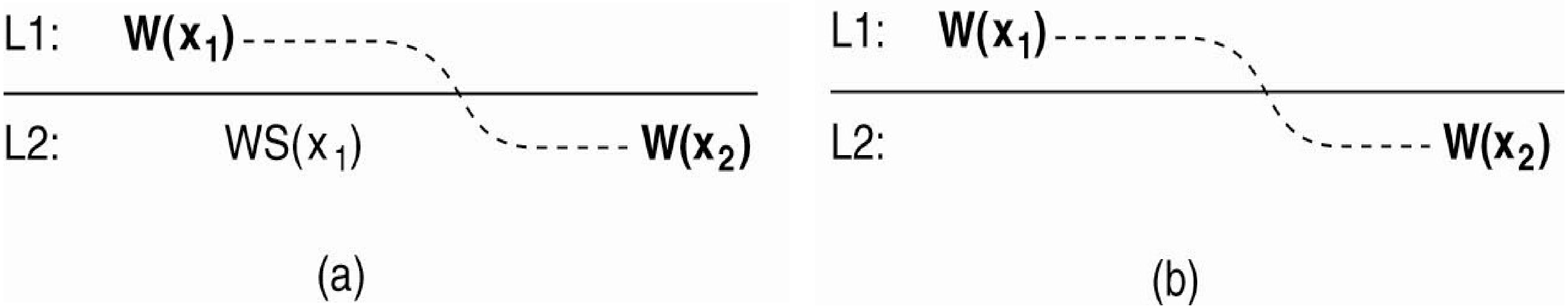


Figure 7-13. The write operations performed by a single process P at two different local copies of the same data store.

(a) A monotonic-write consistent data store.
(b) A data store that does not provide monotonic-write consistency.

Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall, Inc. All rights reserved. 0-13-239227-5

Read your Writes

- **Read your writes**
  The effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process

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

Example-1: The **absence** of read-your-writes consistency is sometimes experienced when updating Web documents and subsequently viewing the effects. Update operations frequently take place by means of an editor or word processor, which saves the new version on a file system that is shared by the Web server.

The user's Web browser accesses the same file, possibly after requesting it from the local Web server. Once the file has been fetched, either the server or the browser often caches a local copy for subsequent accesses. Consequently, when the Web page is updated, the user will not see the effects if the browser or the server returns the cached copy instead of the original file.

Read your Writes

Read-your-writes consistency can guarantee that if the editor and browser are integrated into a single program, the cache is invalidated when the page is updated, so that the updated file is fetched and displayed.

Example-2: Similar effects occur when updating passwords. For example, to enter a digital library on the Web, it is often necessary to have an account with an accompanying password. However, changing a password make take some time to come into effect, with the result that the library may be inaccessible to the user for a few minutes. The delay can be caused because a separate server is used to manage passwords and it may take some time to subsequently propagate (encrypted) passwords

to the various servers that constitute the library.

Read Your Writes

In Fig. 7-14(a), process P performed a write operation W(XI) 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(XI ;X2), which states that W(Xl) is part of WS(X2). In contrast, in Fig. 7-14(b), W(Xl) 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·

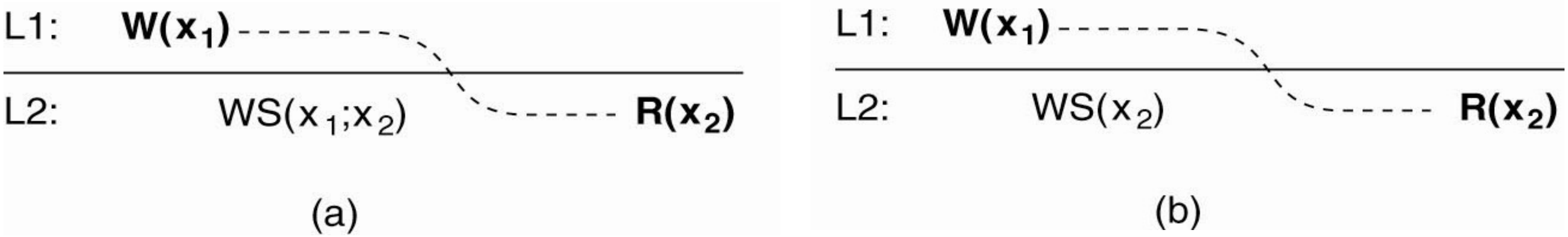


Figure 7-14. (a) A data store that provides read-your-writes consistency. (b) A data store that does not.

Writes follow Reads

- **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 more recent values of x that was read

In other words, 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.

Writes follow Reads

Example: Writes-follow-reads consistency can be used to guarantee that users of a network newsgroup see a posting of a reaction to an article only after they have seen the original article.

To understand the problem, assume that a user first reads an article A. Then, he reacts by posting a response B. By requiring writes-follow-reads consistency, B will be written to any copy of the newsgroup only after A has been written as well.

Note that users who only read articles need not require any specific client-centric consistency model. The writes-follows reads consistency assures that reactions to articles are stored at a local copy only if the original is stored there as well.

Writes Follow Reads

In Fig. 7-15(a), 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.)*
In contrast, no guarantees are given that the operation performed at *L2, as shown in Fig. 7-15(b),* are performed on a copy that is consistent with the one just read at *L1.*

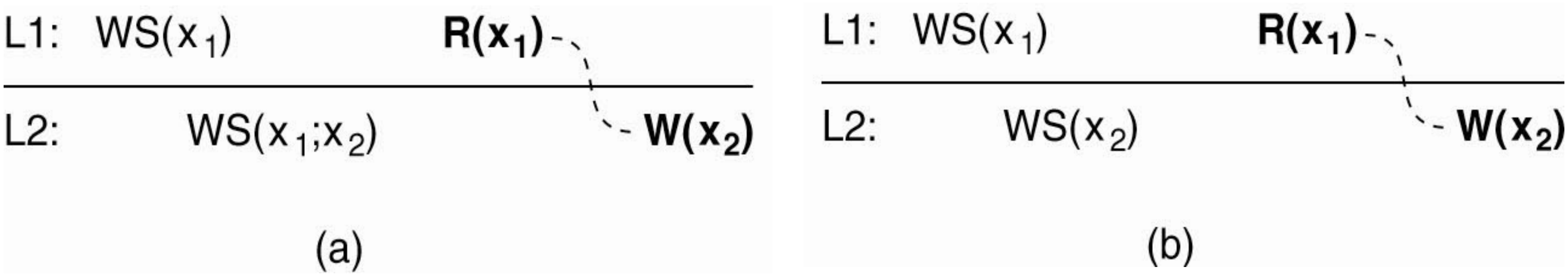


Figure 7-15. (a) A writes-follow-reads consistent data store.
(b) A data store that does not provide writes-follow-reads consistency.

Replica Management
Replica Placement

A key issue for any distributed system that supports replication is to decide where, when, and by whom replicas should be placed, and subsequently which mechanisms to use for keeping the replicas consistent.

The placement problem itself should be split into two sub-problems: that of **placing** ***replica servers***, *and that* of **placing** ***content***.

Replica-server placement is concerned with finding the best locations to place a server that can host (part of) a data store.

Content placement deals with finding the best servers for placing content. Note that this often means that we are looking for the optimal placement of only a single data item. Obviously, before content placement can take place, replica servers will have to be placed first. In the following, take a look at these two different placement problems, followed by a discussion on the basic mechanisms for managing the replicated content.

Replica-Server Placement

Replica-server placement is often more of a management and commercial issue than an optimization problem.

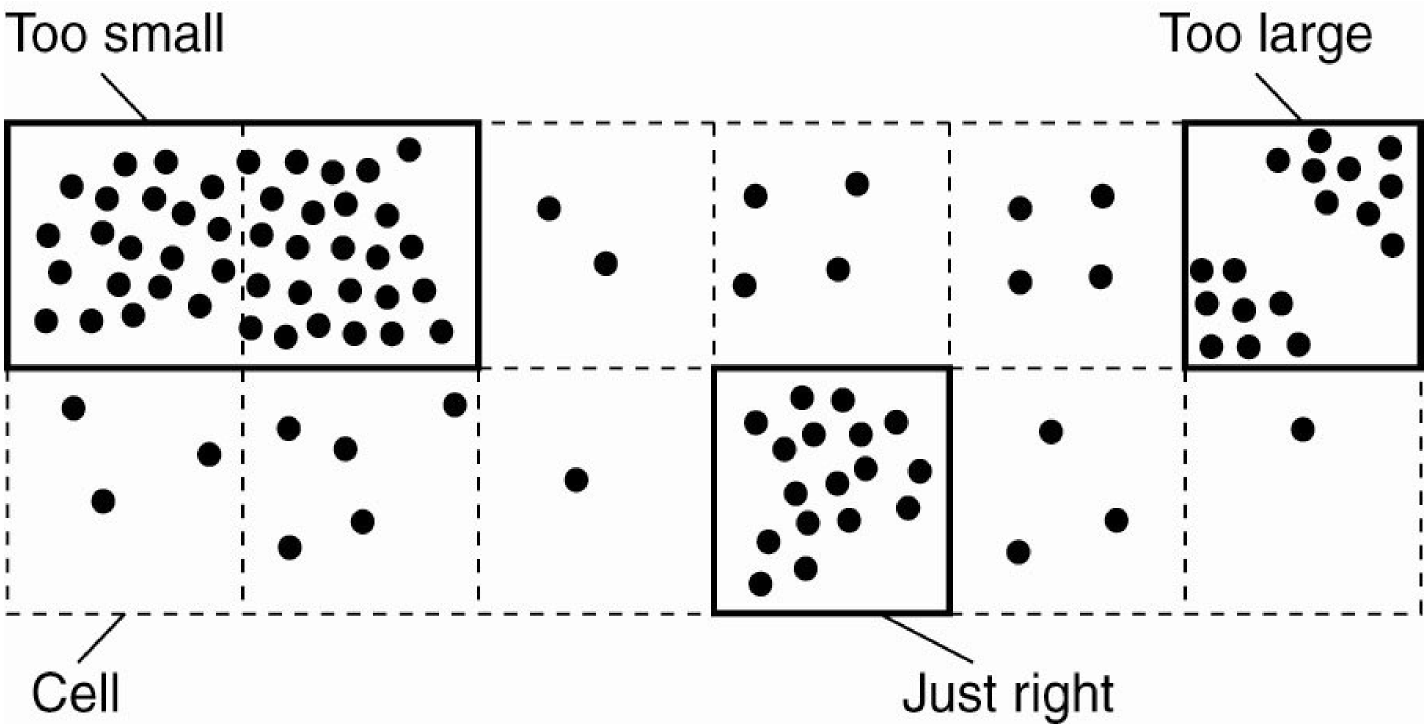


Figure 7-16. Choosing a proper cell size for server placement.

Distribution Protocols

Replica Placement

Where, when, by whom copies of data are to be placed?

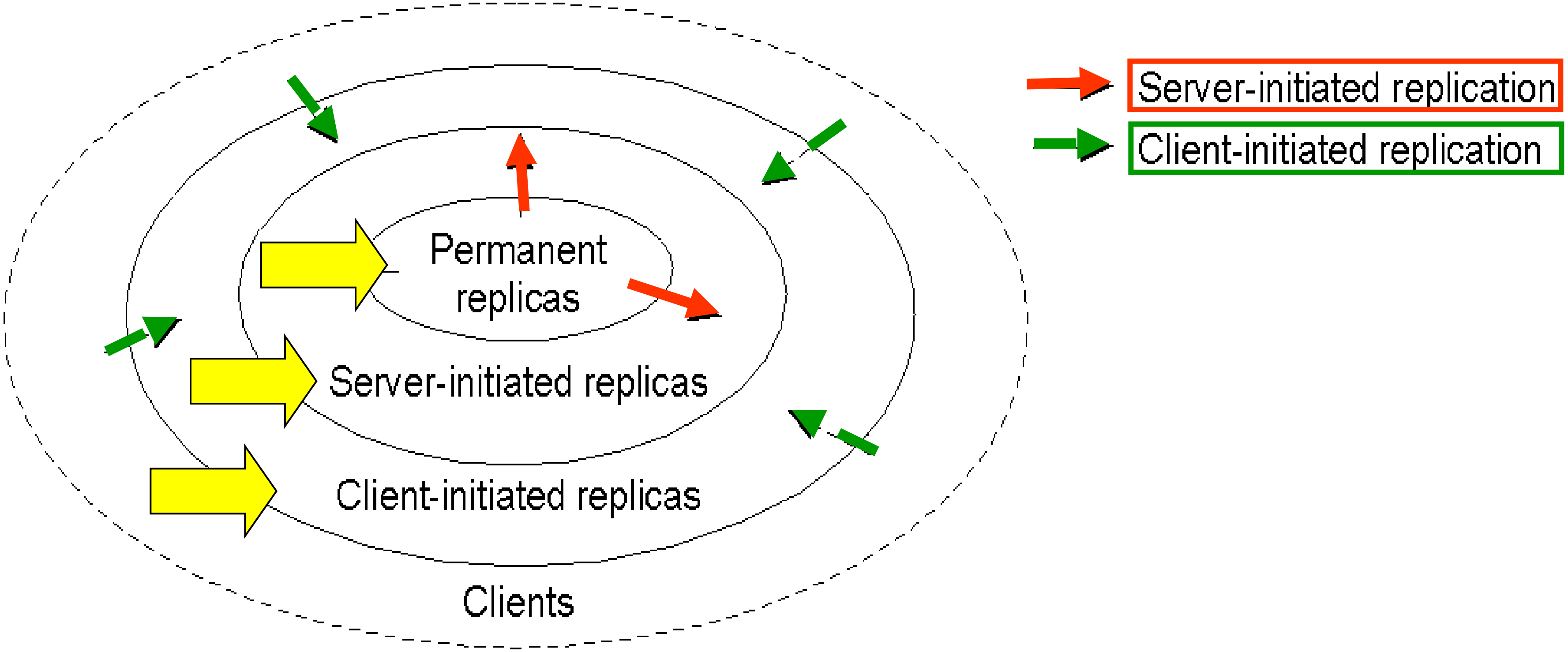

Server-initiated replication
Client-initiated replication
Permanent replicas
Server-initiated replicas
Client-initiated replicas
Clients

Figure 7-17: The logical organization of different kinds of copies of a data store into three concentric rings.

Content Replication and Placement

Permanent Replica: Permanent replicas can be considered as the initial set of replicas that constitute a distributed data store. In many cases, the number of permanent replicas is small. Consider, for example, a Web site. Distribution of a Web site generally comes in one of **two forms**.

The **first** kind of distribution is one in which 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.

The **second** form of distributed Web sites is what is called **mirroring**. In this case, a Web site is copied to a limited number of servers, called mirror sites which are **geographically spread across the Internet**. In most cases, clients simply choose one of the various mirror sites from a list offered to them. Mirrored Web sites have in common with cluster-based Web sites that there are only a few number of replicas, which are more or less **statically** configured.

Server-Initiated Replicas

In contrast to permanent replicas, server-initiated replicas are copies of a data store that exist to enhance performance and which are created at the initiative of the (owner of the) data store.

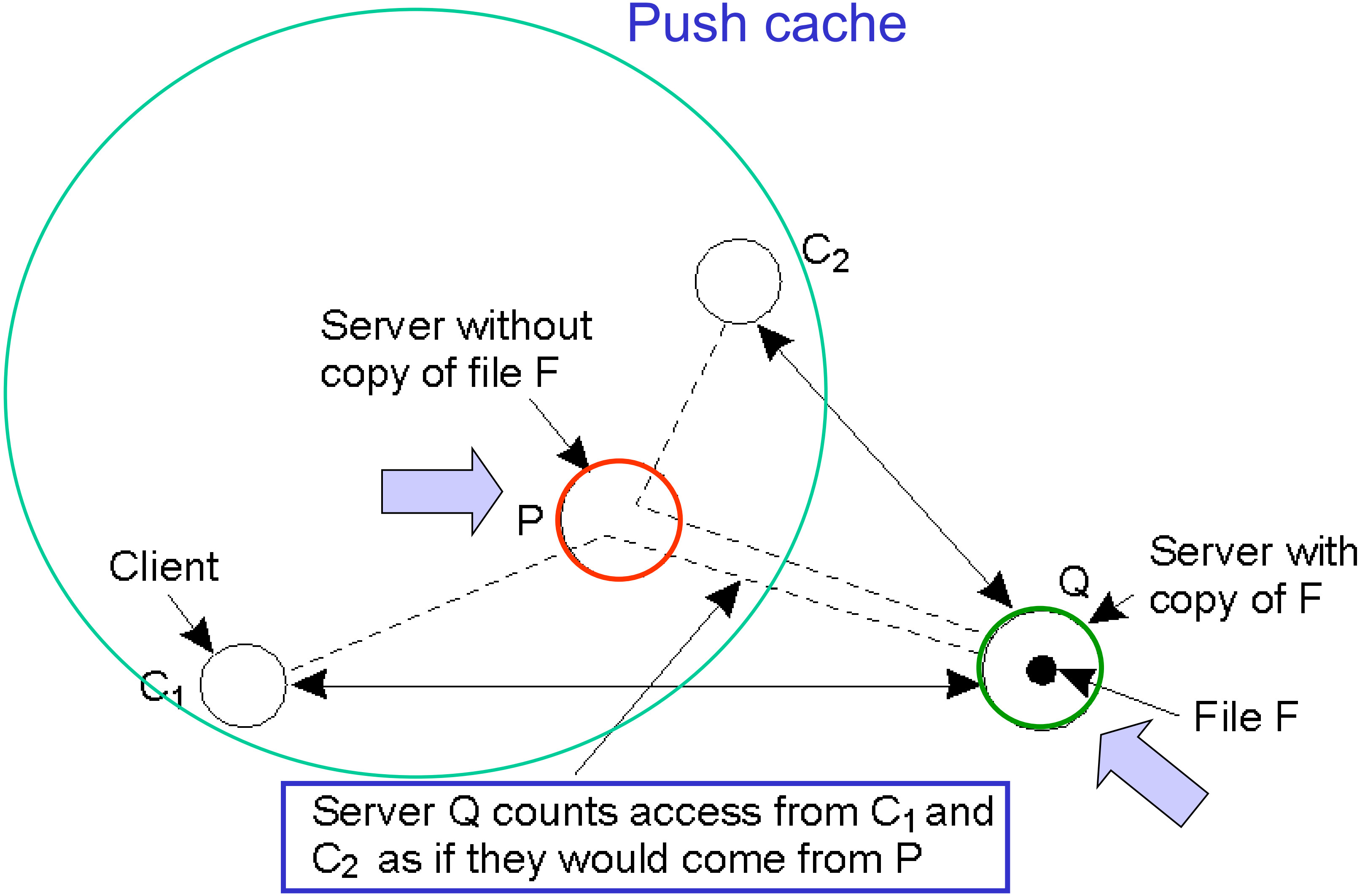
Consider, for example, a Web server placed in New York. Normally, this server can handle incoming requests quite easily, but it may happen that over a couple of days a sudden burst of requests come in from an unexpected location far from the server. In that case, it may be worthwhile to install a number of temporary replicas in regions where requests are coming from.

To provide optimal facilities such hosting services can dynamically replicate files to servers where those files are needed to enhance performance, that is, close to demanding (groups of) clients.

The algorithm for dynamic replication takes **two issues** into account. **First**, replication can take place to reduce the load on a server. **Second**, 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.

Server-Initiated Replicas

Push cache



Server Q counts access from C_1 and C_2 as if they would come from P

Web case. Counting access requests from different clients.

Client-initiated Replicas

An important kind of replica is the one initiated by a client. Client-initiated replicas are more commonly known as **(client) caches**. In essence, a cache is a local storage facility that is used by a client to temporarily store a copy of the data it has just requested. In principle, managing the cache is left entirely to the client.

The data store from where the data had been fetched has nothing to do with keeping cached data consistent. However, as we shall see, there are many occasions in which the client can rely on participation from the data store to inform it when cached data has become stale. Client caches are used only to improve access times to data.

Content Distribution
State versus Operations

Replica management also deals with propagation of (updated) content to the relevant replica servers. There are various trade-offs to make, which we discuss next. Possibilities for what is to be propagated:

1. Propagate only a notification of an update.
2. Transfer data from one copy to another.
3. Propagate the update operation to other copies.

Pull versus Push Protocols

Another design issue is whether updates are pulled or pushed. In a push-based approach, also referred to as server-based protocols, updates are propagated to other replicas without those replicas even asking for the updates.

Push-based approaches are often used between permanent and server-initiated replicas, but can also be used to push updates to client caches.

Server-based protocols are applied when replicas generally need to maintain a relatively high degree of consistency. In other words, replicas need to be kept identical.

Consequently, the read-to-update ratio at each replica is relatively high.

Pull versus Push Protocols

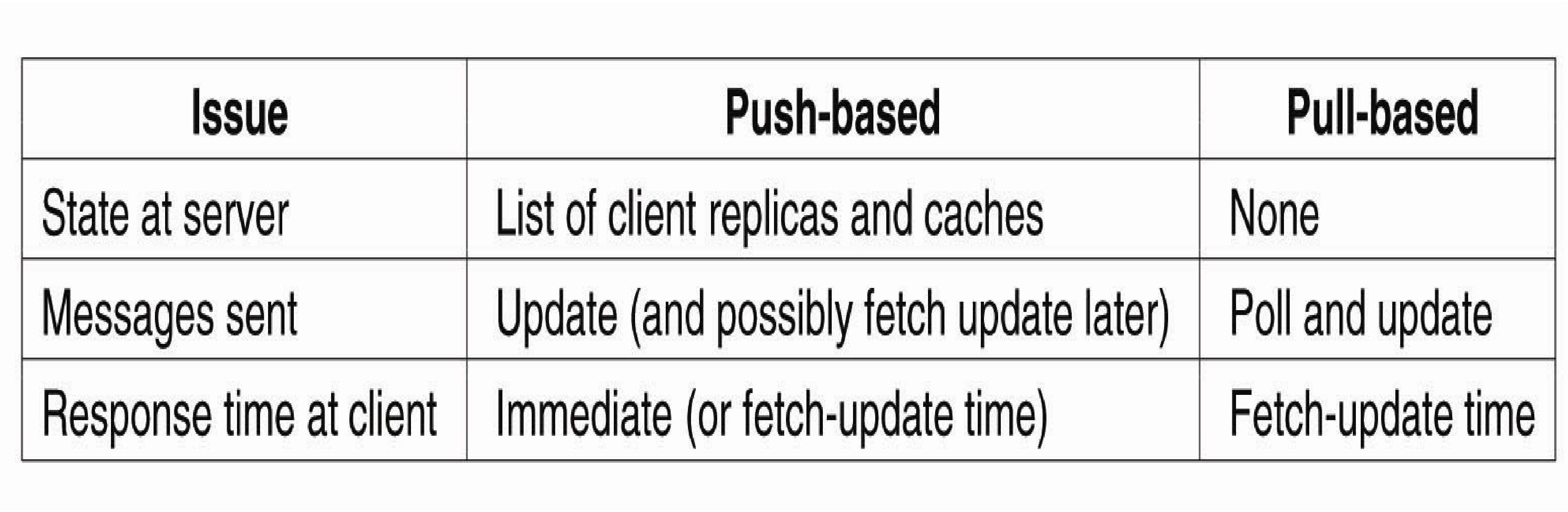
In contrast, in a pull-based approach, a server or client requests another server to send it any updates it has at that moment.

Pull-based protocols, also called client-based protocols, are often used by client caches. For example, a common strategy applied to Web caches is first to check whether cached data items are still up to date. When a cache receives a request for items that are still locally available, the cache checks with the original Web server whether those data items have been modified since they were cached.

In the case of a modification, the modified data are first transferred to the cache, and then returned to the requesting client. If no modifications took place, the cached data are returned. In other words, the client polls the server to see whether an update is needed.

A pull-based approach is efficient when the read-to-update ratio is relatively low.

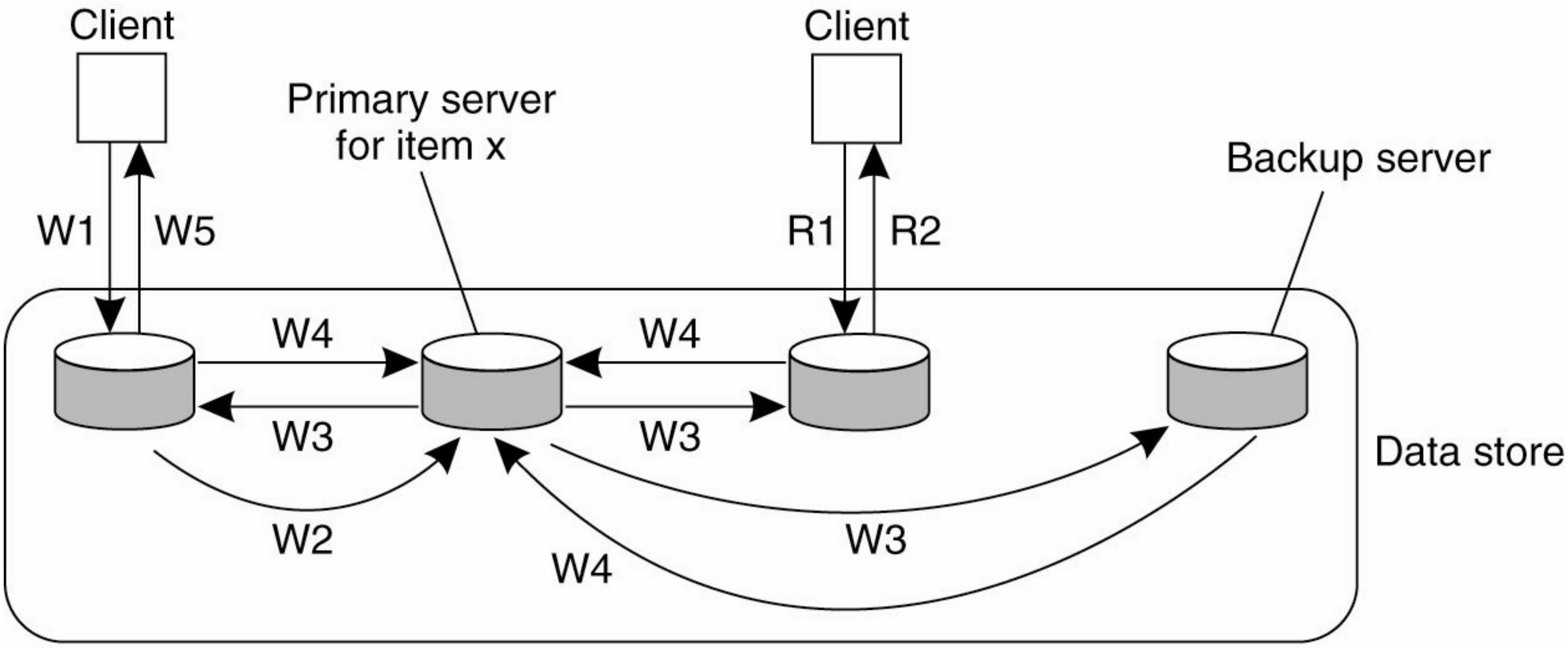
Pull versus Push Protocols



Issue	Push-based	Pull-based
State at 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

Figure 7-19. A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems.

Remote-Write Protocols

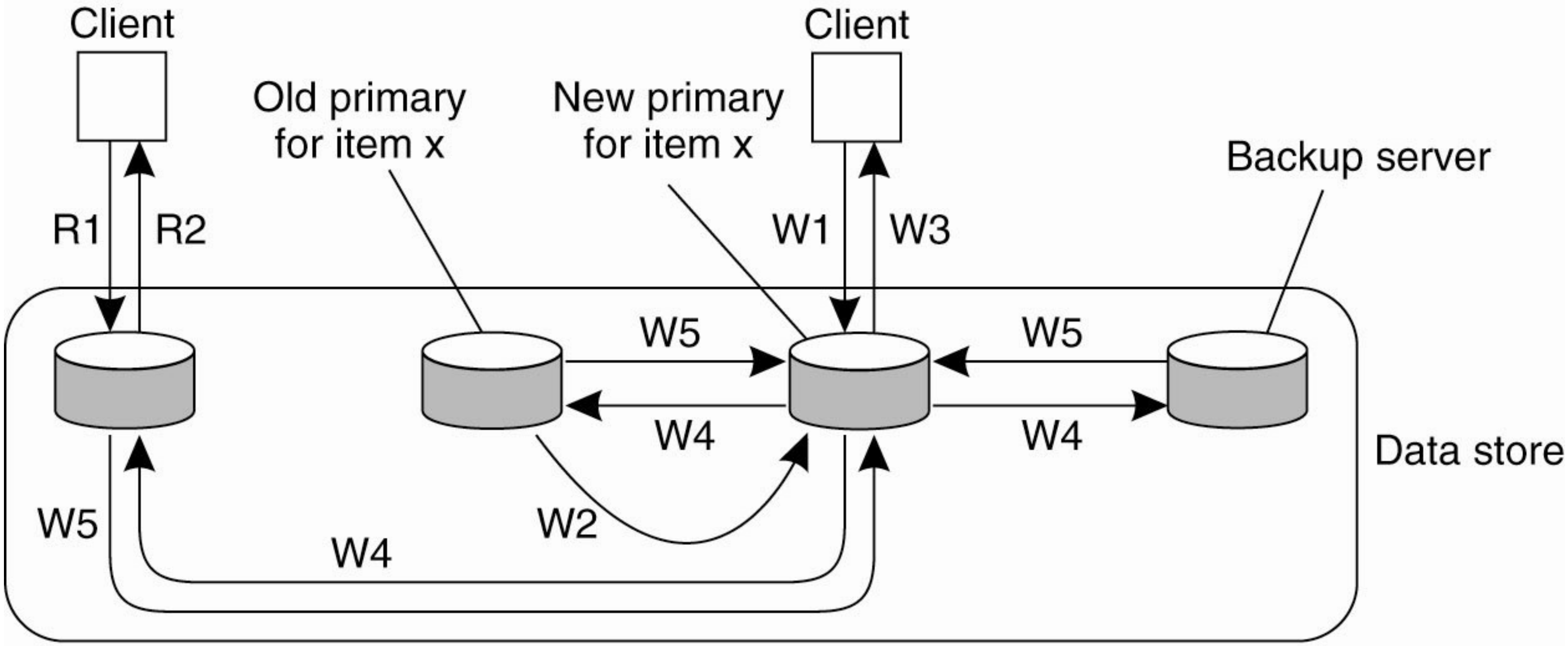


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Figure 7-20. The principle of a primary-backup protocol.

Local-Write Protocols



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

Figure 7-21. Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

Quorum-Based Protocols

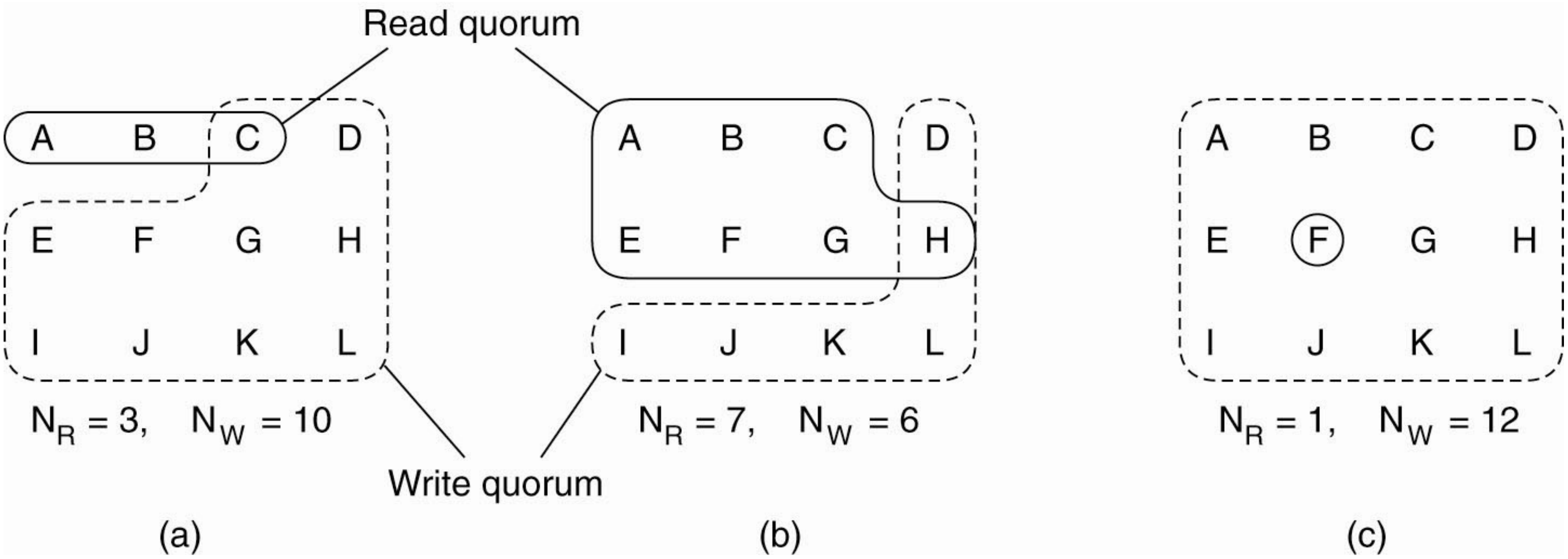


Figure 7-22. Three examples of the voting algorithm. (a) A correct choice of read and write set. (b) A choice that may lead to write-write conflicts. (c) A correct choice, known as ROWA (read one, write all).