DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 7
Consistency And Replication

*** Modified by Prof. Rivalino Matias, Jr.**

Outline

- Introduction.
- Data-centric consistency models.
- Client-centric consistency models.
- Replica management.
- Consistency protocols.

Introduction

- Reasons for replication
  - Data are replicated to increase the **availability** of a system.
    - If a filesystem is replicated it may be possible to continue working after one replica crashes.
    - By maintaining multiple copies, it becomes possible to provide better protection against corrupted data. For example, in a three-copy file, in case of one copy is corrupted, the other two are used to decide on the correct value returned.
  - Another reason for replicating data is **performance**.
    - It is important when a DS needs to **scale in terms of size** or the **geographical area** it coves.
    - Scaling with respect to size occurs, e.g., when an increasing nr. of processes need to access data managed by a single server. By replicating the server and dividing the workload among them can improve performance.
    - The basic idea of scaling regarding geographic area is by placing a copy of data near by the processes using them.

Introduction

- Replication: Price to be paid!
  - If replication helps to improve availability and performance, who could be against it?
    - Having multiple copies may **lead to consistency problems**.
    - Whenever a copy is modified, that copy becomes different from the rest, so they need to be applied to all copies to ensure consistency.
    - When and how this **data synchronization** need to be carried out determines the **price of replication**.
  - **Example**: Improving access times to Web pages.
    - To improve performance, Web browsers cache locally previously fetched Web pages. 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modified in the meantime, modifications will not have been propagated immediately to cached copies, making those copies out-of-date.

Introduction

- Replication: Price to be paid!
  - **Example**: Improving access times to Web pages (cont'd).
    - 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.
    - Other 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.
    - Another solution is to mark pages that can be cached. For example, pages with real-time updated content should not be cached. In this case the decision is left to the page (or content) author, which may differ from the user's choice.

Introduction

- Replication as scaling technique
  - Replication and caching for performance are widely applied as scaling techniques.
    - Placing copies of data close to the processes using them can improve performance through reduction of access time.
  - A trade-off is that keeping copies up to date require more network bandwidth.
    - Consider a process *P* accessing a local replica *N* times per second, whereas the replica itself is updated *M* times per second. Assume that an update refreshes the previous version of the local replica.
    - If $N \ll M$, i.e., 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.
    - 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 scaling technique
  - A more serious problem, however, is that keeping multiple copies consistent may itself be subject to serious scalability problems.
    - A collection of copies is consistent when the copies are always the same.
    - When an update operation is performed on one copy, the update should be propagated to all copies before a subsequent operation takes place.
    - This type of consistency is sometimes informally (and imprecisely) referred to as **tight consistency** as provided by what is also called **synchronous replication**.
    - The key idea is that an update is performed at all copies as a single atomic operation, or transaction.
    - Implementing atomicity involving a large number of replicas that may be widely dispersed across a large-scale network is inherently difficult when operations are also required to complete quickly.

Introduction

- Replication as scaling technique
  - Difficulties come from the fact that we need to synchronize all replicas.
    - In essence, 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.
    - Global synchronization simply takes a lot of communication time, especially when replicas are spread across a wide-area network.

Introduction

- Replication as scaling technique
  - We are now faced with a dilemma:
    - On the one hand, scalability problems can be alleviated by applying **replication** and **caching**, leading to improved performance.
    - On the other hand, to keep all copies consistent generally requires
    - global synchronization, which is inherently costly in terms of (network) performance.
    - **"The cure may be worse than the disease!"**
  - In many cases the only real solution is to **relax the consistency constraints**:
    - If we can relax the requirement that updates need to be executed as atomic operations, we may be able to avoid (instantaneous) global synchronizations and may thus gain performance.
    - The price paid is that copies may not always be the same everywhere.
    - To what extent consistency can be relaxed depends on the access and update patterns of the replicated data, as well as on the purpose for which those data are used.

Data-centric 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.
  - 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. A data operation is classified as a **write** operation when it changes the data and is otherwise classified as a **read** operation.

Data-centric consistency models

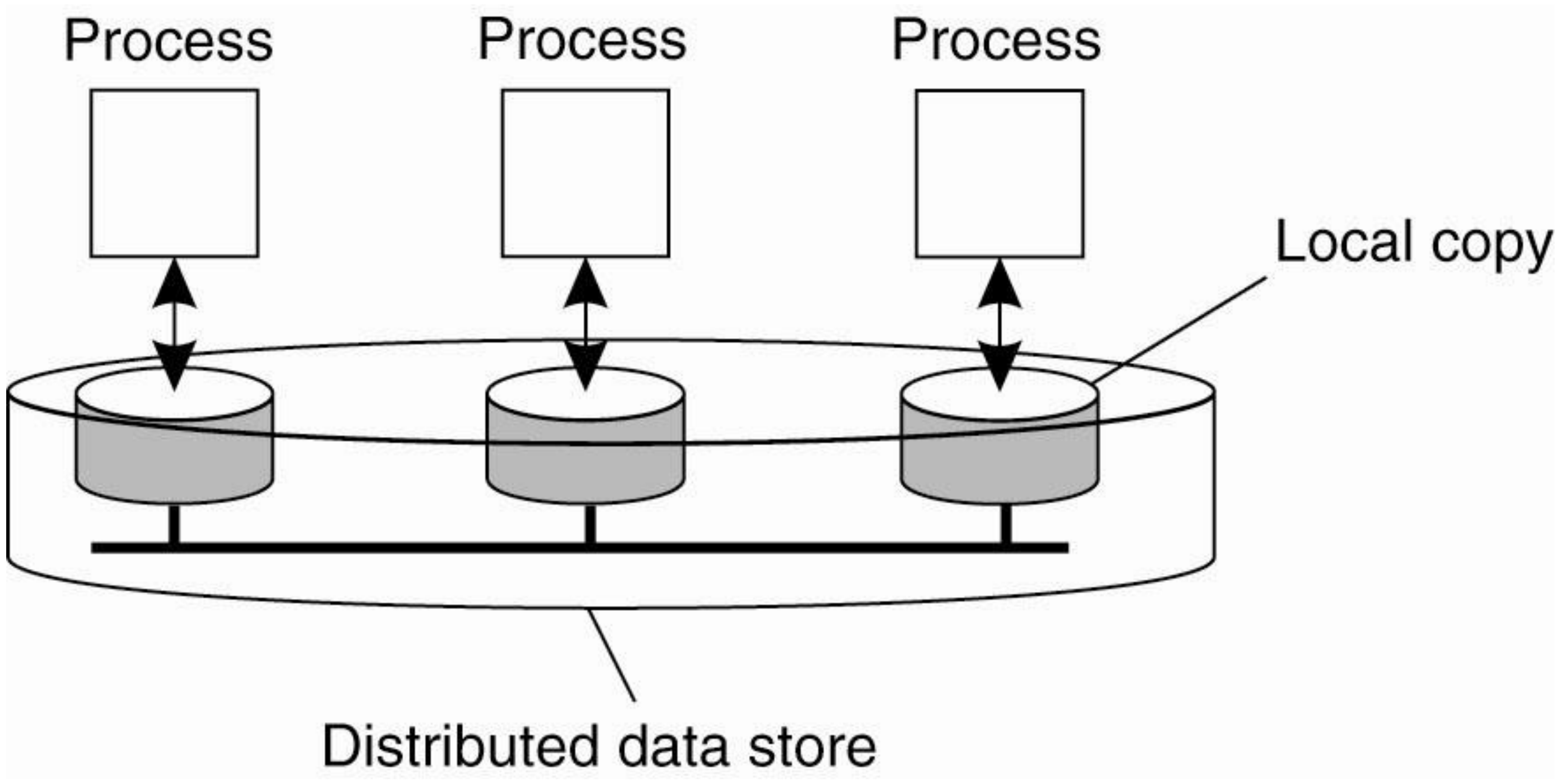


Figure 7-1. The general organization of a logical data store, physically distributed and replicated across multiple processes.

Data-centric consistency models

- A **consistency model** is essentially a contract between processes and the data store.
    - 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.

Data-centric consistency models

- A **consistency model** is essentially a contract between processes and the data store (cont'd).
  - Each model effectively restricts the values that a **read** operation on a data item can return.
  - As expected, the ones with major restrictions are easy to use.
    - e.g., when developing applications, whereas those with minor restrictions are generally considered to be difficult to use in practice.
  - **The trade-off is that the easy-to-use models do not perform nearly as well as the difficult ones.**

Data-centric consistency models

- Continuous consistency:
  - **There is no such thing as a best solution to replicating data.**
  - 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.

Data-centric consistency models

- Continuous consistency (cont'd)
  - 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 (obsolescence) between replicas;
    - Deviation with respect to the ordering of update operations.
  - They refer to these deviations as forming **continuous consistency** ranges.

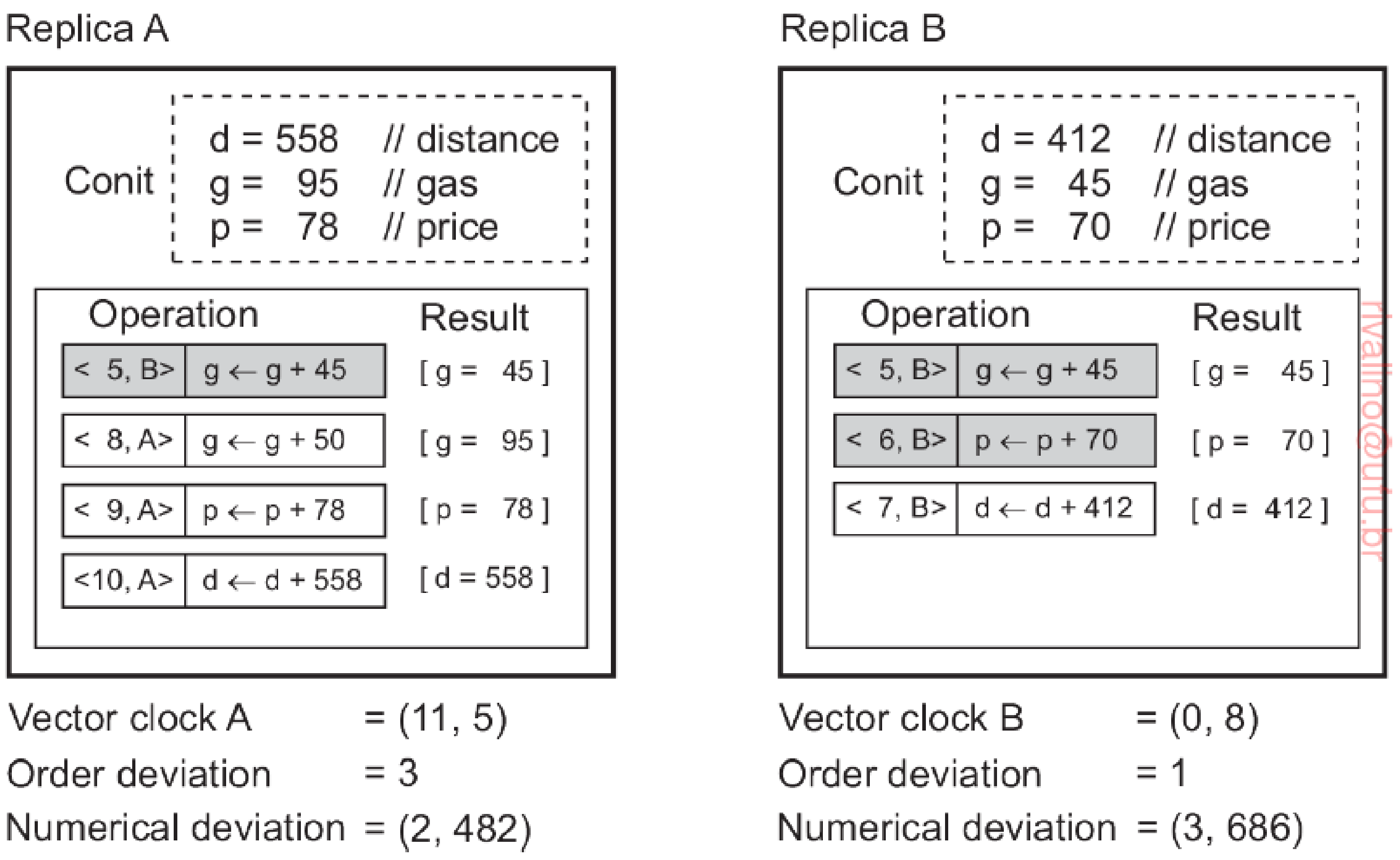
Data-centric consistency models

- The notion of a **conit**
  - To define inconsistencies, Yu and Vahdat [2002] introduce a **consistency unit**, or **conit**:
    - A **conit** specifies the unit over which consistency is to be measured.
  - To give an example of a conit, and at the same time illustrate numerical and ordering deviations, consider the situation of keeping track of a fleet of cars.
    - The fleet owner is interested in knowing how much he pays on average for gas.
    - Whenever a driver tanks gasoline, he reports the amount of gasoline that has been tanked (recorded as **g**), the price paid (recorded as **p**), and the total distance since the last time he tanked (recorded by the variable **d**).
    - Technically, the three variables **g**, **p**, and **d** form a **conit**.
    - This **conit** is replicated across two servers.

Data-centric consistency models

- The notion of a **conit** (cont'd)
  - The task of the servers is to keep the **conit** "consistently" replicated.
  - To this end, each replica server maintains a two-dimensional vector clock.
  - We use the notation $< T, R >$ to express an operation that was carried out by replica R at (its) logical time T.

Continuous consistency (1)



Replica A
Conit
d = 558 // distance
g = 95 // gas
p = 78 // price
Operation
Result
< 5, B> g ← g + 45 [g = 45]
< 8, A> g ← g + 50 [g = 95]
< 9, A> p ← p + 78 [p = 78]
<10, A> d ← d + 558 [d = 558]
Vector clock A = (11, 5)
Order deviation = 3
Numerical deviation = (2, 482)
Replica B
Conit
d = 412 // distance
g = 45 // gas
p = 70 // price
Operation
Result
< 5, B> g ← g + 45 [g = 45]
< 6, B> p ← p + 70 [p = 70]
< 7, B> d ← d + 412 [d = 412]
Vector clock B = (0, 8)
Order deviation = 1
Numerical deviation = (3, 686)

Figure 7.2: An example of keeping track of consistency deviations.

Data-centric consistency models

- The notion of a **conit** (cont'd)
  - In the previous figure, we see two replicas that operate on a **conit** containing the data items: **g**, **p**, and **d**.
  - All variables are assumed to have been initialized to zero.
  - Replica **A** received the operation **$< 5, \ B >$: g $\leftarrow$ g + 45** from replica **B**.
  - The above operation shaded in gray indicate that **A** has *commited* this operation to its local store, which makes it permanent.
  - Replica **A** has three *tentative* update operations listed < 8, A >, < 9, A >, and < 10, A >, respectively.
    - The fact that **A** has tree tentative operations pending is referred to as an **order deviation** of value 3, in this case.
    - Analogously, there is an order deviation of order 1 in **B**.

Data-centric consistency models

- The notion of a **conit** (cont'd)
  - The **numerical deviation** in a replica consists of two components:
    - The **nr. of operations** at all other replicas that have not yet been seen by the local replica, along with the sum of corresponding missed values.
    - In the previous example, **A** has not yet seen operations **< 6, B >** and **< 7, B >** with a total value of **70 + 412** units, leading to a numerical deviation of **(2, 482)**.
    - Likewise, **B** is still missing the three tentative operations at **A**, with a total summed value of **686**, bringing **B**'s numerical deviation to **(3, 686)**.
    - Using these notions, it becomes possible to specify consistency schemes.
    - For example, we may restrict order deviation by specifying an acceptable maximal value. Likewise, we may want two replicas to never numerically deviate by more than 1000 units.

Continuous consistency (2)

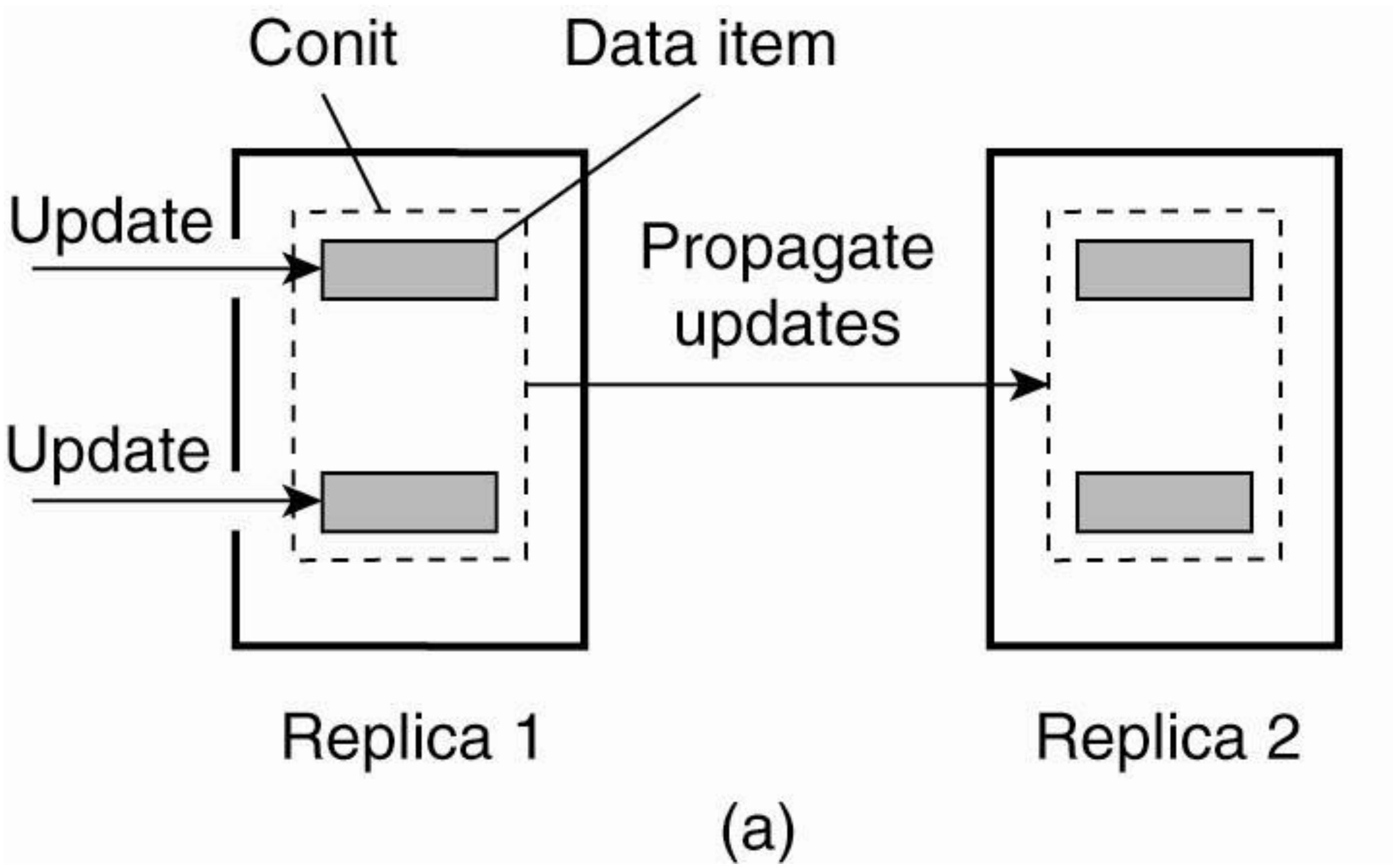


Figure 7-3. Choosing the appropriate granularity for a conit. (a) Two updates lead to update propagation.

Continuous consistency (3)

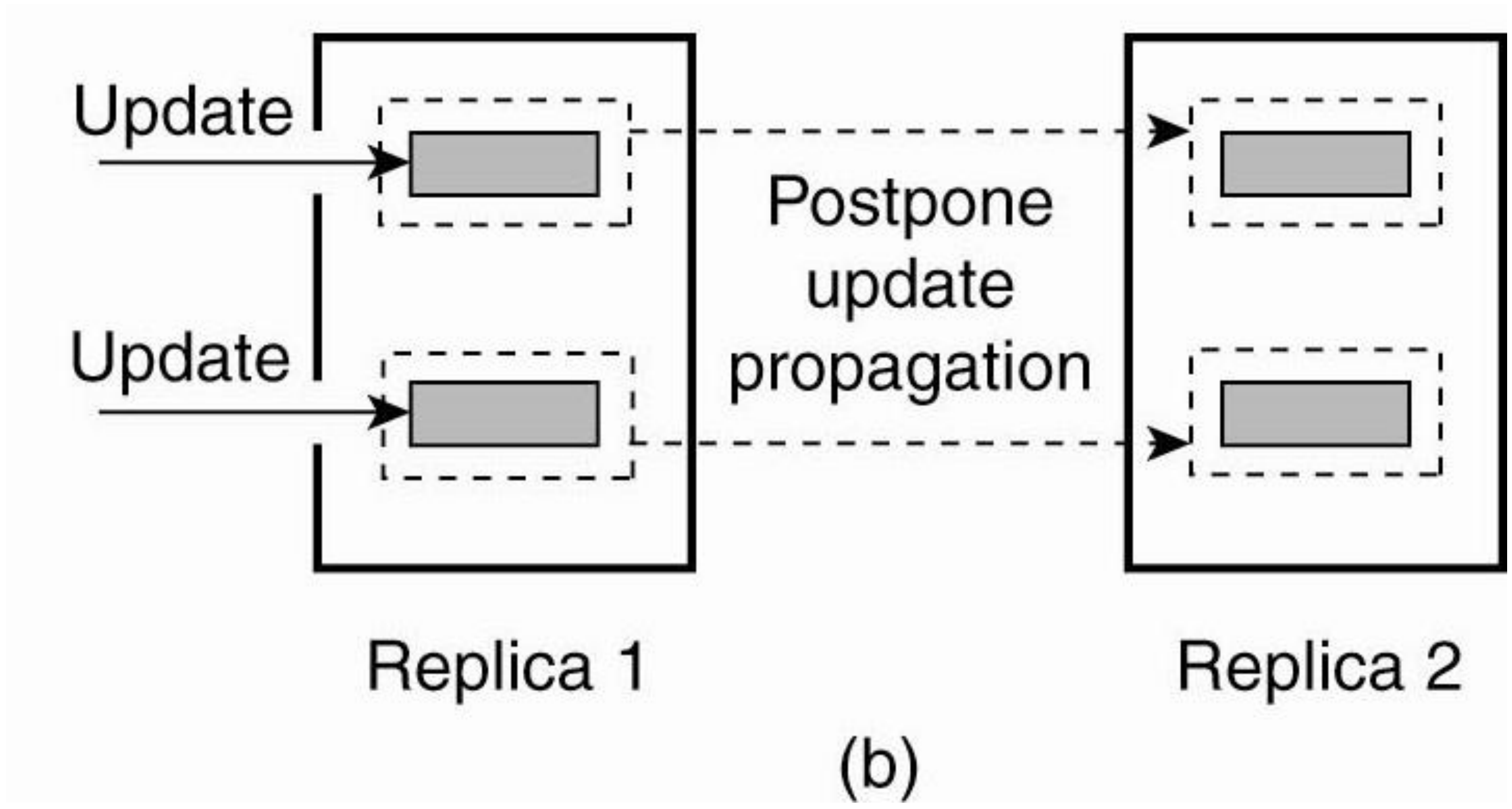


Figure 7-3. Choosing the appropriate granularity for a conit.
(b) No update propagation is needed (yet).

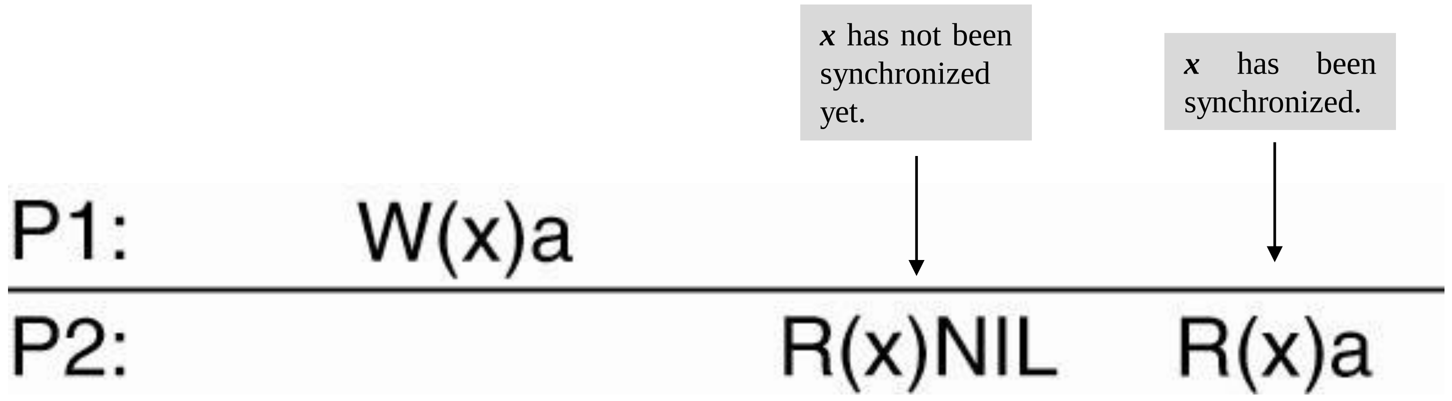
Data-centric consistency models

- Consistent ordering of operations
  - In parallel and distributed computing, multiple processes will need to share resources and access these resources simultaneously.
  - Researchers have sought to express the semantics of concurrent accesses when shared resources are replicated.
  - The way of express that is based on the **continuous consistency model**.
    - When tentative updates at replicas need to be committed, replicas will need to reach agreement on a global, i.e., consistent ordering of those updates.

Data-centric consistency models

- **Sequential consistency**
  - Let's express this approach as follows:
    - The time axis is always drawn horizontally, with time increasing from left to right.
    - $W_i(x)$**a** denotes that process **P**$_i$ writes value **a** to data item *x*.
    - $R_i(x)$**b** denotes that process **P**$_i$ reads *x* and is returned the value **b**.
    - Assume that each data item has initial value NIL.

Sequential Consistency (1)



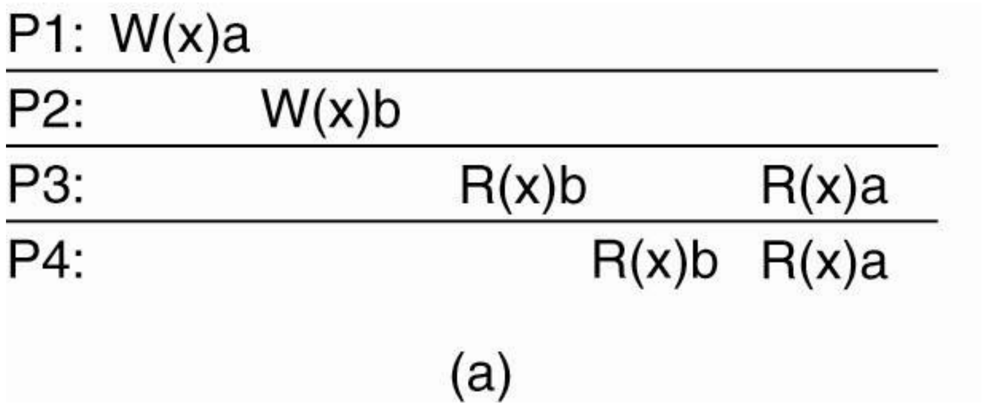
x has not been synchronized yet.
x has been synchronized.
P1:
W(x)a
P2:
R(x)NIL
R(x)a

Figure 7-4. Behavior of two processes operating on the same data item. The horizontal axis is time.

Sequential Consistency (2)

- A data store is said to be sequentially consistent when it satisfies the following condition:
  - The result of any execution is the same as if the (**read** and **write**) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.
- This means that when processes run concurrently on different hosts, any valid interleaving of **read** and write operations is acceptable behavior, but all processes see the same interleaving of operations.
  - Nothing is said about time; that is, there is no reference to the "most recent" write operation on a data item.
  - Time does not play a role in this approach (see next slide).

Sequential Consistency (3)



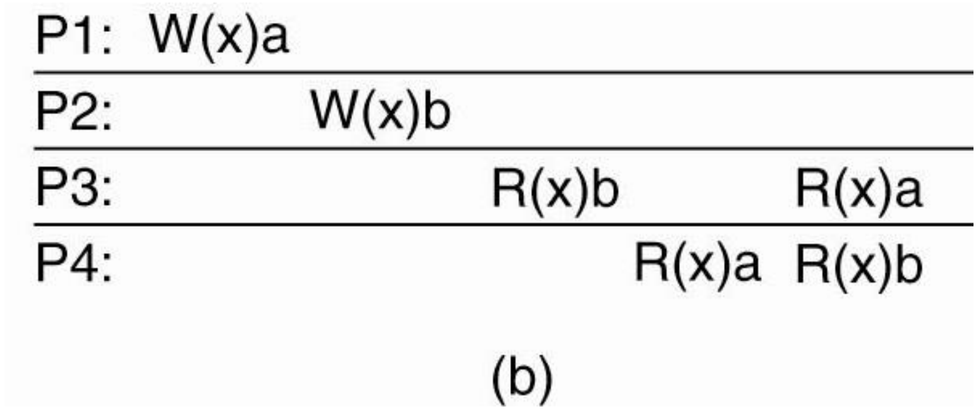


Figure 7-5. (a) A sequentially consistent data store.
(b) A data store that is not sequentially consistent.

Sequential Consistency (4)

- In Figure 7.5(a) (previous slide), processes P_3 and P_4 first read value b, and later value a.
  - The write operations appear consistent to all processes.
- In contrast, Figure 7.5(b) shows an example that violates sequential consistency
  - Because not all processes see the same interleaving of write operations.

Causal Consistency (1)

- The causal consistency model
  - Proposed by [Hutto and Ahamad, 1990], it represents a weakening of sequential consistency, since it makes a distinction between events that are potentially causally related and those that are not.
  - **Causality**: If event **b** is caused or influenced by an earlier event **a**, causality requires that everyone else first see **a**, then see **b**.
- E.g., in a distributed shared data base
  - Supposes that P1 writes a data item **x**.
  - Then P2 reads **x** and writes **y**.
  - The reading of **x** and writing of **y** are **potentially causally** related.
  - On the other hand, if two processes concurrently write two different data items, these are not causally related.

Causal Consistency (1)

- The causal consistency model (cont'd)
  - 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 hosts.
- Example
  - In the next slide, we have an event sequence that is allowed with a casually consistent store but forbidden with a sequentially consistent store (or a strictly consistent store).
  - **The thing to note is that** the writes $W_2(x)$**b** and $W_1(x)$**c** are concurrent, so it is not required that all processes see them in the same order.

Causal Consistency (2)

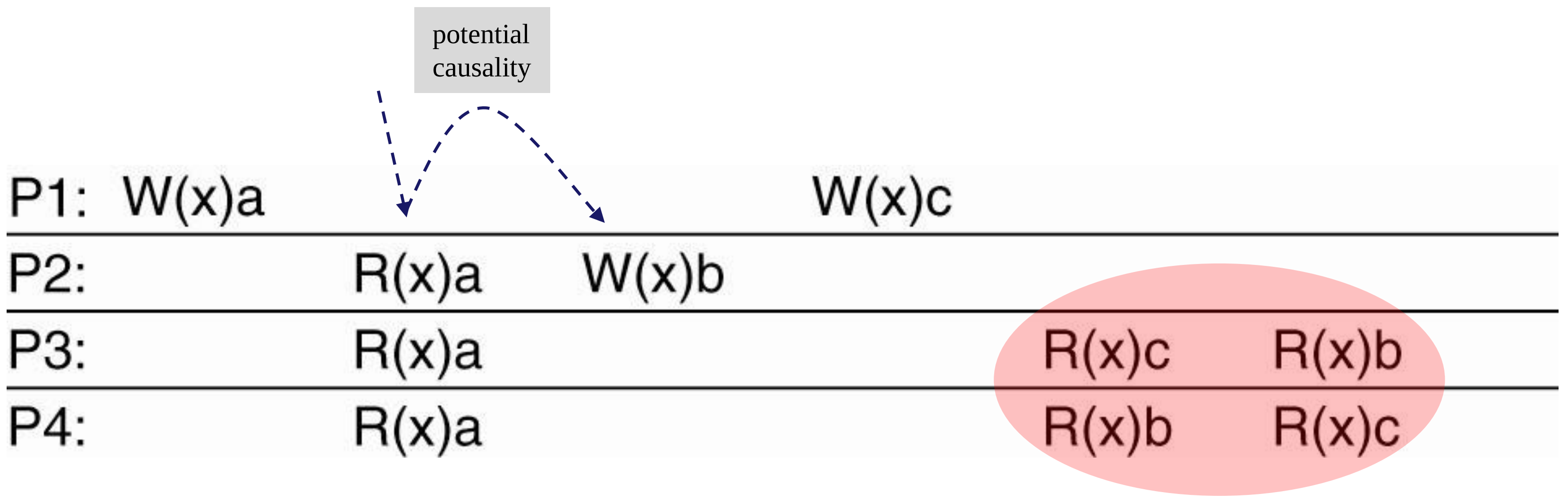


Figure 7-8. This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

Causal Consistency (3)

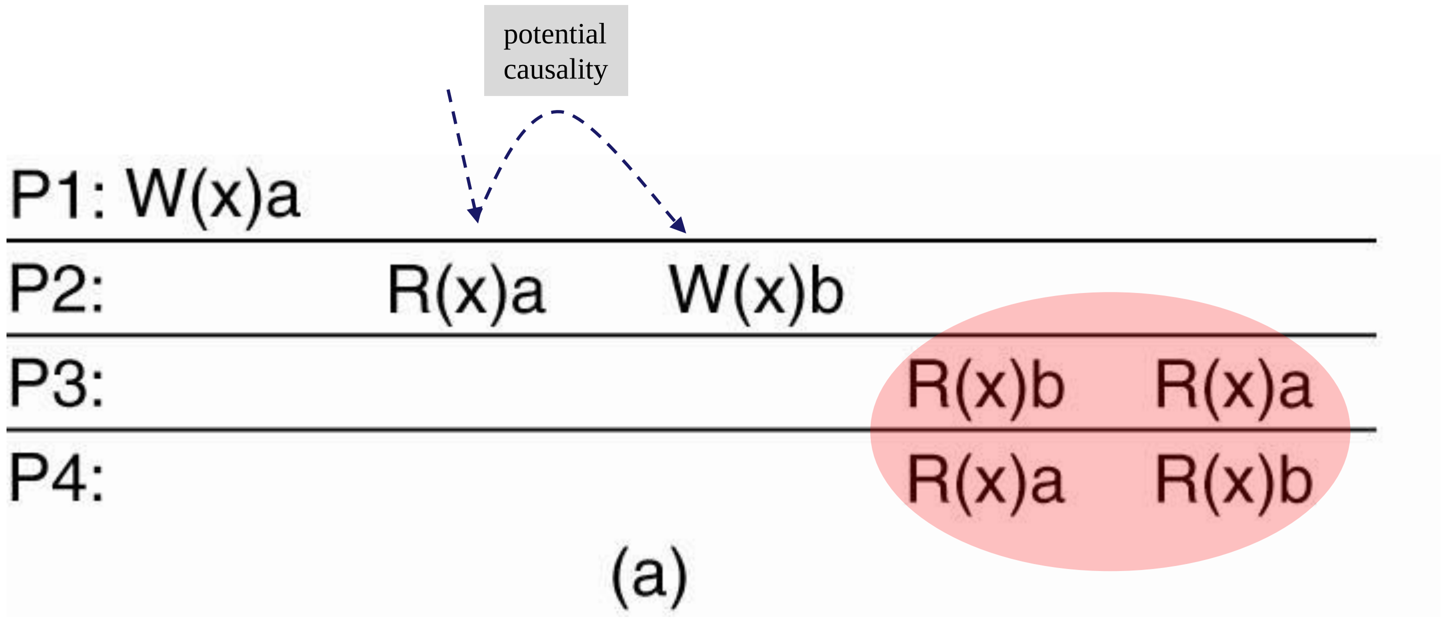


Figure 7-9. (a) A violation of a causally-consistent store.

Causal Consistency (4)

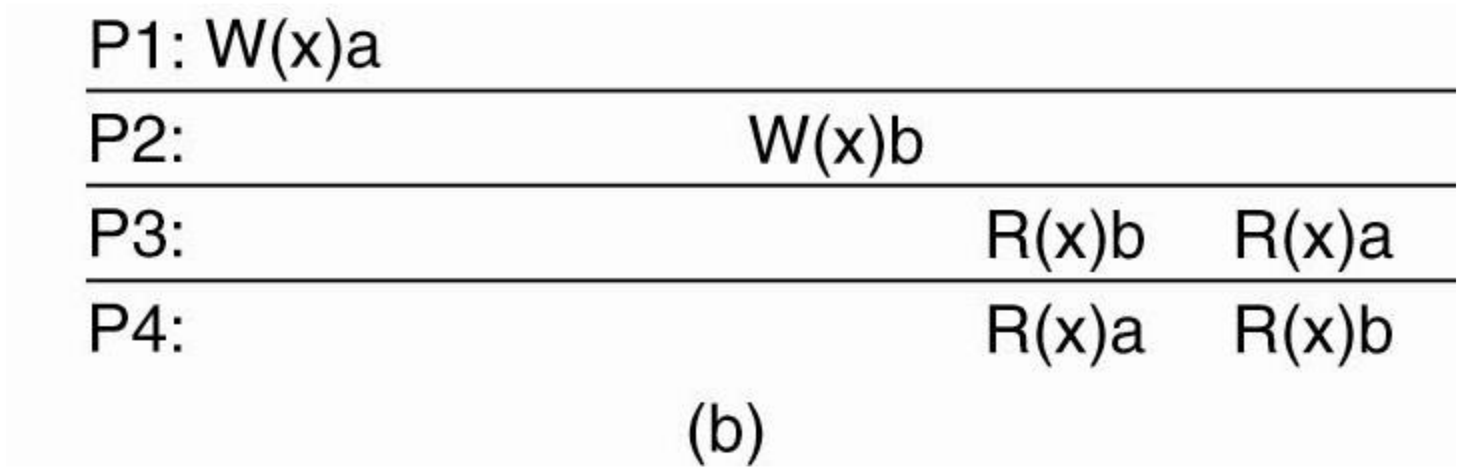


Figure 7-9. (b) A correct sequence of events in a causally-consistent store.

Data-centric consistency models

- Consistency vs. Coherence
  - A **consistency model** describes what can be expected with respect to that set of data items when multiple processes concurrently operate on them. The set is 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 hold for only a single data item [Cantin et al., 2005].
  - We assume that a replicated **data item** is said to be coherent when the various copies abide to the rules as defined by its associated consistency 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

- Data-centric consistency models aim at providing a systemwide 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 strong consistency is fundamental to distributed systems.
- Client-centric consistency
  - Provides guarantees for a *single client* concerning the consistency of accesses to a data store by that client.
  - No guarantees are given concerning concurrent accesses by different clients.
  - It tolerates a relatively high degree of inconsistency.

Client-centric consistency models

- Client-centric consistency
  - To what extent processes operate in a concurrent fashion, and to what extent consistency needs to be guaranteed, may vary.
  - There are many situations in which concurrency appears only in a restricted form. For example:
    - In many database systems, processes hardly ever perform update operations; they mostly read data from the database.
    - The question is how fast updates should be made available to only-reading processes.
    - In the globally operating CDNs, developers often choose to propagate updates slowly, assuming that most clients are redirected to the same replica and thus will never experience inconsistencies.

Client-centric consistency models

- Client-centric consistency
  - In many cases, distributed and replicated database applications can tolerate a relatively high degree of inconsistency.
    - If no updates take place for a long time, all replicas will gradually become consistent, i.e., have the same data stored.
    - This form of consistency is called **eventual consistency**.
  - Data stores that are **eventually consistent** thus have the property that in the absence of **write-write conflicts**, all replicas will converge toward identical copies of each other.
  - 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.

Client-centric consistency models

- Client-centric consistency
  - **Eventually consistent data stores** generally work fine as long as clients always access the same replica.
  - However, problems arise when different replicas are accessed over a short period of time (see example on next slide).

Client-centric consistency models

- Example:
  - The mobile user, Alice, accesses the database by connecting to one of the replicas in a transparent way.
    - the application running on Alice’s mobile device is unaware on which replica it is actually operating.
  - Alice performs several update operations and then disconnects.
  - Later, she accesses the database again, after moving to a different location or by using a different access device.
  - At that point, she may be connected to a different replica than before.
    - If the updates performed previously have not yet been propagated, Alice will notice inconsistent behavior.
    - In particular, she would expect to see all previously made changes, but instead, it appears as if nothing at all has happened.

Eventual Consistency

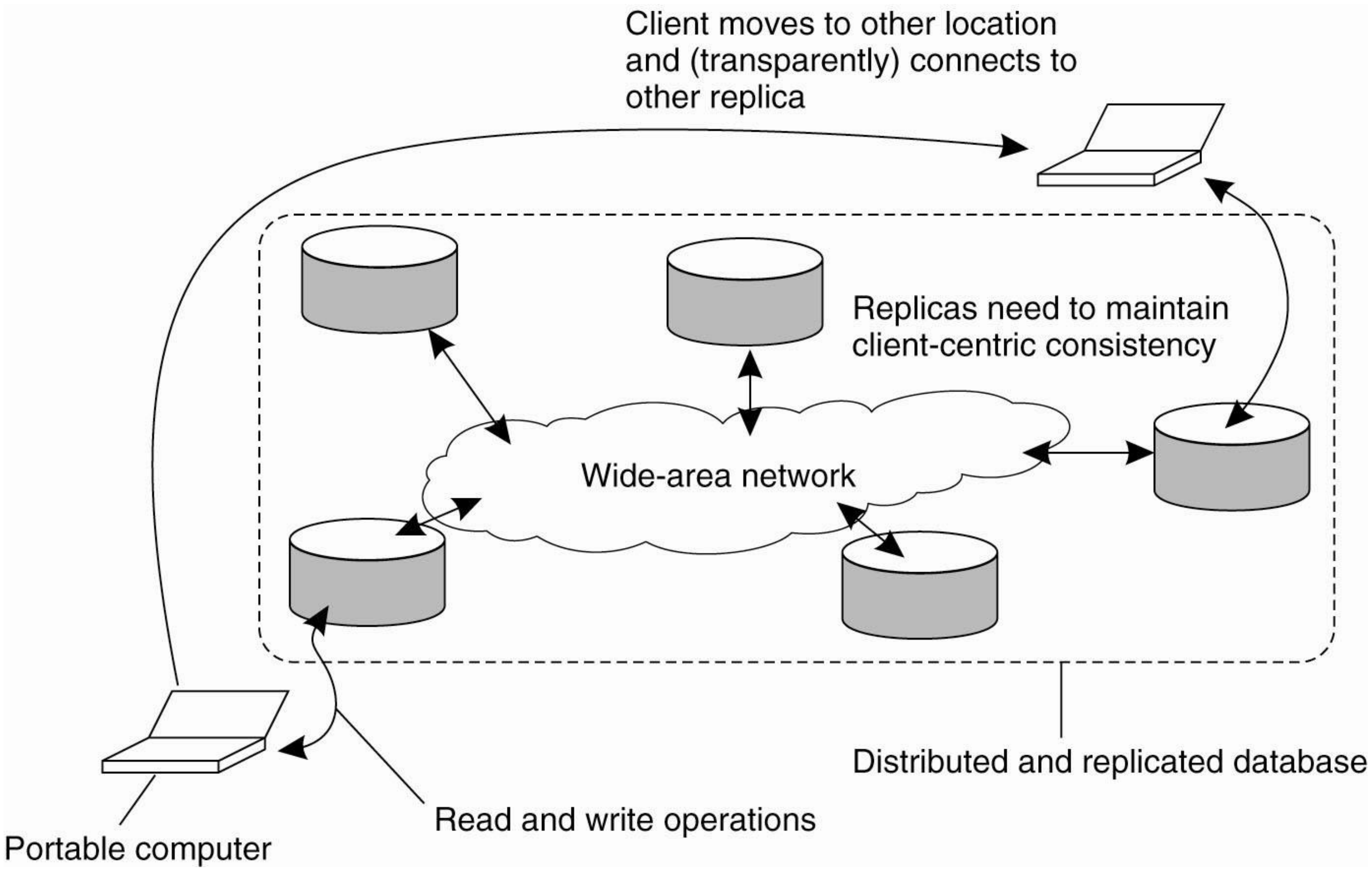

Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

Figure 7-11. The principle of a mobile user accessing different replicas of a distributed database.

Client-centric consistency models

- The problem mentioned before can be alleviated by introducing client-centric consistency.
  - In essence, it provides guarantees for a single client concerning the consistency of accesses to a data store by that client.
  - When a process accesses the data store, it generally connects to the locally (or nearest) available copy, although, in principle, any copy will do just fine.
  - All read and write operations are performed on that local copy. Updates are eventually propagated to the other copies.
  - No guarantees are given concerning concurrent accesses by different clients.

Client-centric consistency models

- Client-centric consistency models are described using the following notations.
  - Let $\mathbf{x}_i$ denote the version of data item $\mathbf{x}$.
  - Version $\mathbf{x}_i$ is the result of a series of **write** operations that took place since initialization, its **write set** $WS(\mathbf{x}_i)$.
  - By appending write operations to that series we obtain another version $\mathbf{x}_j$ and say that $\mathbf{x}_j$ follows from $\mathbf{x}_i$.
  - We use the notation $WS(\mathbf{x}_i; \mathbf{x}_j)$ to indicate that $\mathbf{x}_j$ follows from $\mathbf{x}_i$.
  - If we do not know if $\mathbf{x}_j$ follows from $\mathbf{x}_i$, we use the notation $WS(\mathbf{x}_i | \mathbf{x}_j)$.
- There are essentially four client-centric consistency models.
  - Monotonic reads, Monotonic writes,
  - Read-your-writes, Writes-follow-reads.

Monotonic Reads (1)

- A data store is said to provide **monotonic-read consistency** if the following condition holds:
  - If a process reads the value of a data item **x**, any successive read operation on **x** by that process will always return that same value or a more recent value.
- In other words:
  - **Monotonic-read consistency** guarantees that once a process (*the client*) has seen a value of **x**, it will never see an older version of **x**.

Monotonic Reads (2)

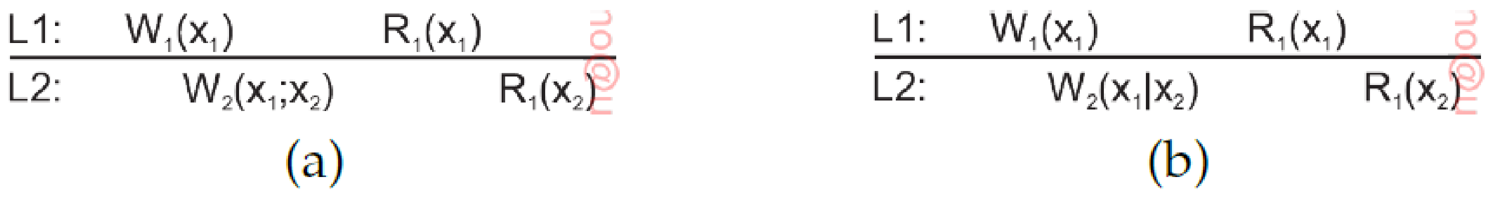

L1: $W_1(x_1)$ $R_1(x_1)$
L2: $W_2(x_1;x_2)$ $R_1(x_2)$
(a)
L1: $W_1(x_1)$ $R_1(x_1)$
L2: $W_2(x_1|x_2)$ $R_1(x_2)$
(b)

Figure 7.16: The read operations performed by a single process *P* at two different local copies of the same data store. (a) A monotonic-read consistent data store. (b) A data store that does not provide monotonic reads.

Monotonic Writes (1)

- A data store is said to provide **monotonic-write consistency** if 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.
- More formally:
  - If we have two successive operations, $W_k(x_i)$ and $W_k(x_j)$ by process P_k, then, regardless where $W_k(x_j)$ takes place, we also have $WS(x_i; x_j)$.

Monotonic Writes (2)

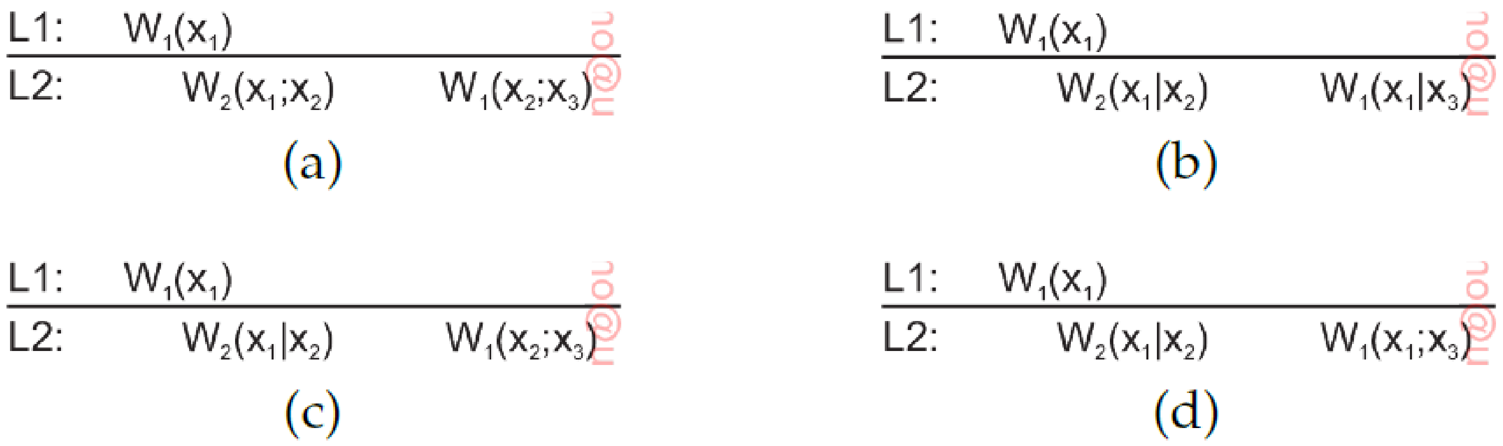
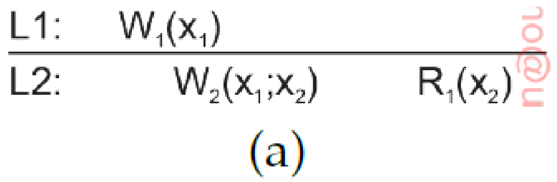

L1: $W_1(x_1)$
L2: $W_2(x_1;x_2)$ $W_1(x_2;x_3)$
(a)
L1: $W_1(x_1)$
L2: $W_2(x_1|x_2)$ $W_1(x_1|x_3)$
(b)
L1: $W_1(x_1)$
L2: $W_2(x_1|x_2)$ $W_1(x_2;x_3)$
(c)
L1: $W_1(x_1)$
L2: $W_2(x_1|x_2)$ $W_1(x_1;x_3)$
(d)

Figure 7.17: The write operations performed 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. (c) Again, no consistency as $WS(x_1|x2)$ and thus also $WS(x_1|x_3)$. (d) Consistent as $WS(x_1;x_3)$ although x_1 has apparently overwritten x_2.

Read your Writes (1)

- 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.
- 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.

Read Your Writes (2)



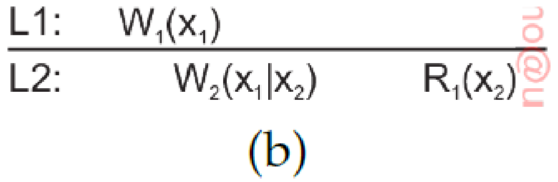


Figure 7.18: (a) A data store that provides read-your-writes consistency. (b) A data store that does not.

Writes Follow Reads (1)

- A data store is said to provide **writes-follow-reads** consistency if the following condition holds:
  - A **write** operation by a process on a data item **x** following a previous **read** operation on **x** by the same process is guaranteed to take place on the same or a more recent value of **x** that was **read**.
- 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 (2)

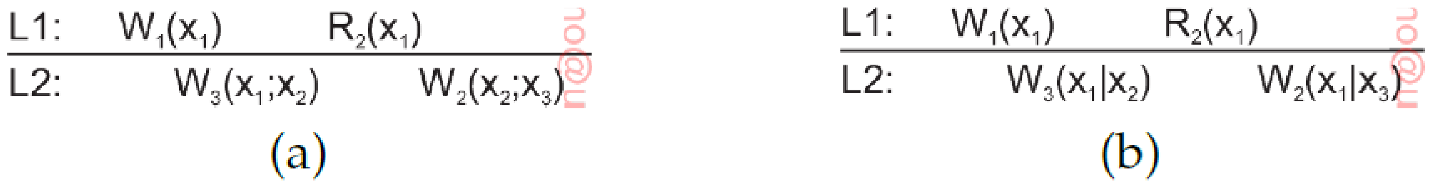

L1: $W_1(x_1)$ $R_2(x_1)$
L2: $W_3(x_1;x_2)$ $W_2(x_2;x_3)$
(a)
L1: $W_1(x_1)$ $R_2(x_1)$
L2: $W_3(x_1|x_2)$ $W_2(x_1|x_3)$
(b)

Figure 7.19: (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.
  - Subsequently, it is required to decide which mechanisms to use for keeping the replicas consistent.
- The **placement problem** itself should be split into two subproblems:
  - **Placing replica servers and 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.

Replica Management

- **Finding the best server location**:
  - Over a decade ago, one could be concerned about where to place an individual server.
  - Nowadays it has changed considerably with the advent of the many large-scale data centers located across the Internet.
  - Likewise, connectivity continues to improve, making precisely locating servers less critical.
  - The placement of replica servers is not an intensively studied problem for the simple reason that it is often more of a management and commercial issue than an optimization problem.
  - Nonetheless, analysis of client and network properties are useful to come to informed decisions.

Replica Management

- **Finding the best server location**:
  - There are various ways to compute the best placement of replica servers, but all boil down to an optimization problem in which the best K out of N locations need to be selected ($K < N$).
    - These problems are known to be computationally complex and can be solved only through heuristics.
  - Qiu et al. [2001] take the distance between clients and locations as their starting point. Distance can be measured in terms of latency or bandwidth.
  - Radoslavov et al. [2001] propose to ignore the position of clients and only take the topology of the Internet as formed by the autonomous systems (AS).
    - They first consider the largest AS and place a server on the router with the largest number of network interfaces (i.e., links). This algorithm is then repeated with the second largest AS, and so on.

Replica Management

- **Finding the best server location**:
  - One problem with the previously cited algorithms is that they are computationally expensive.
    - For example, both the previous algorithms have a complexity that is higher than $\mathbf{O}(N^2)$, where N is the number of locations to inspect.
    - In practice, this means that for even a few thousand locations, a computation may need to run for tens of minutes. This may be unacceptable.
  - Szymaniak et al. [2006] have developed a method by which a region for placing replicas can be quickly identified.
    - A region is identified to be a collection of nodes accessing the same content, but for which the internode latency is low.
    - The goal of the algorithm is first to select the most demanding regions–that is, the one with the most nodes–and then to let one of the nodes in such a region act as replica server.

Replica-Server Placement

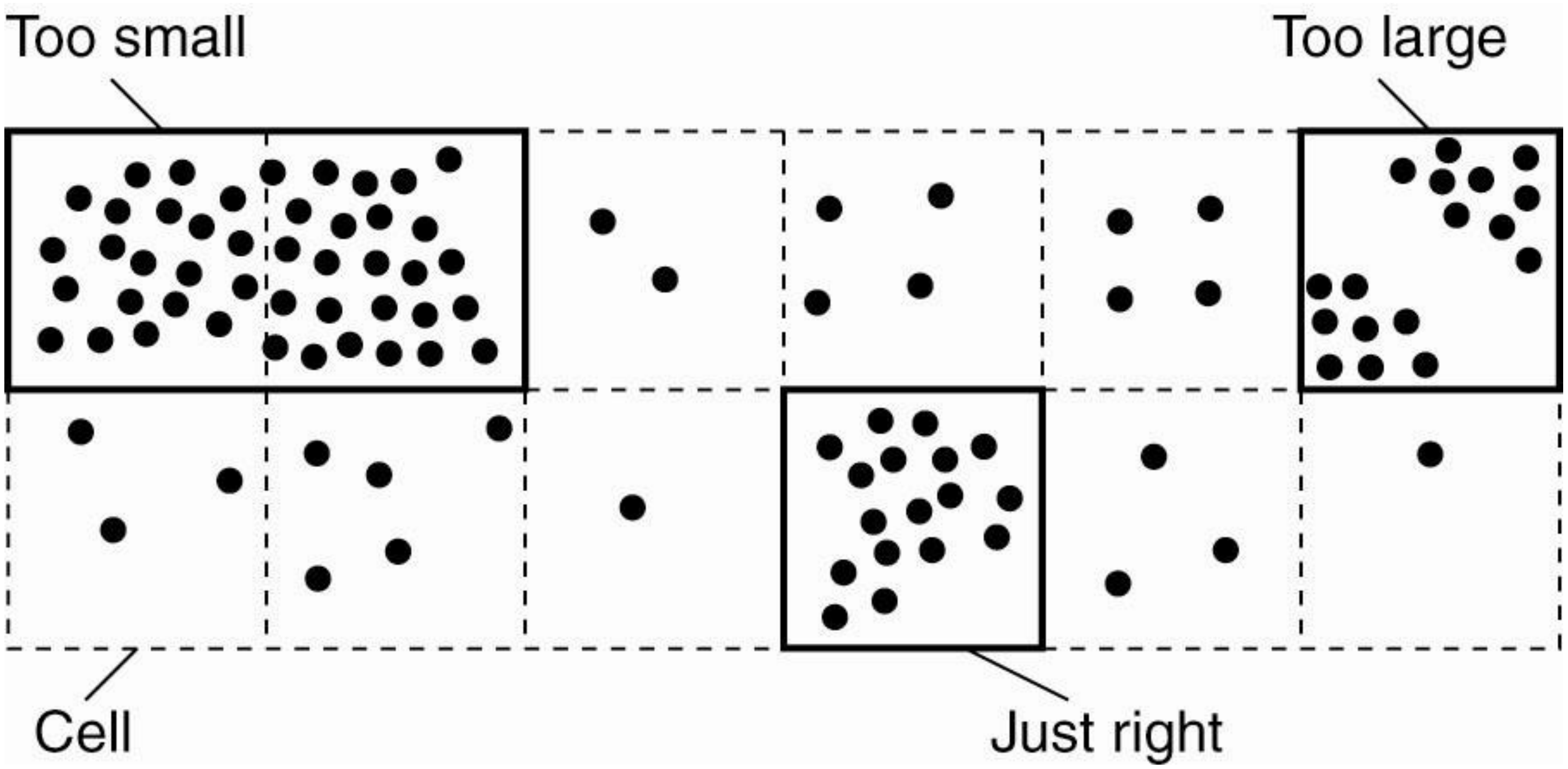


Figure 7-16. Choosing a proper cell size for server placement.

Replica Management

- Content replication & placement:
  - When it comes to content replication and placement, three different types of replicas can be distinguished logically organized:
    - **Permanent replicas**
    - **Client-initiated replicas**
    - **Server-initiated replicas**

Content Replication and Placement

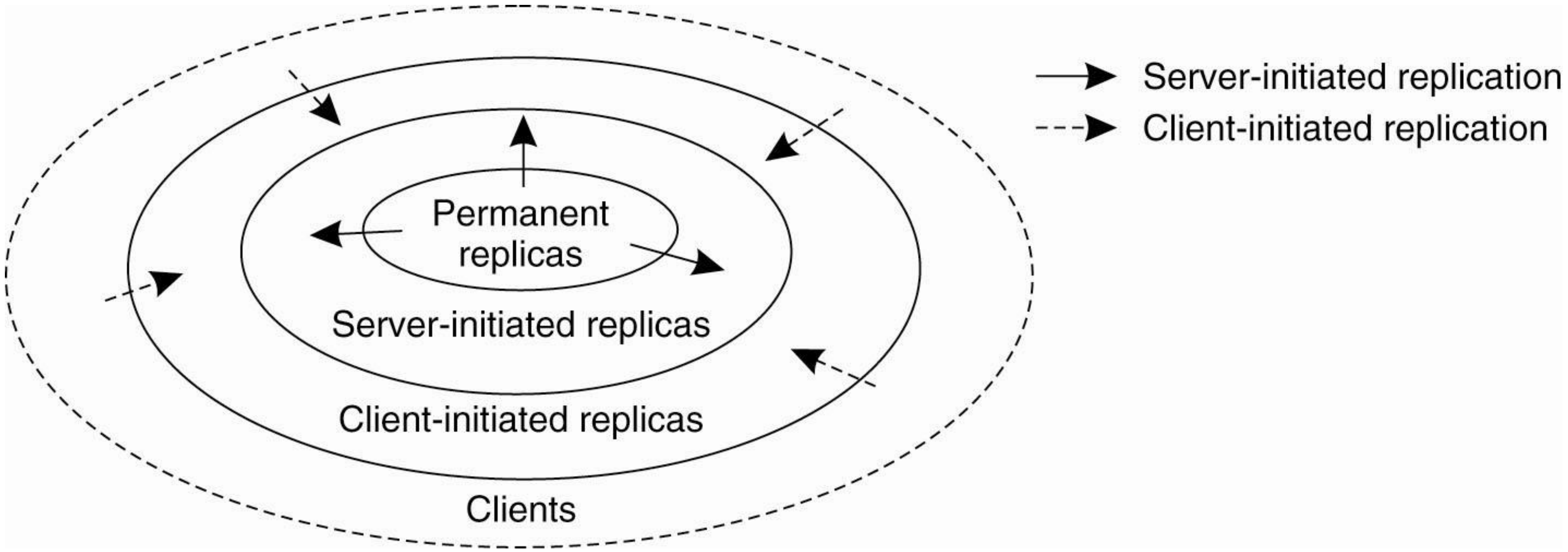

Server-initiated replication
Client-initiated replication
Permanent replicas
Server-initiated replicas
Client-initiated replicas
Clients

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

Replica Management

- Content replication & placement:
  - **Permanent replicas**:
    - They are the initial set of replicas that constitute a distributed data store.
    - In many cases, the number of permanent replicas is small.
    - For example: a Web site.
      - ✓ Distribution of a Web site generally comes in one of **two forms**:
      - ✓ **First:** the files that constitute a site are replicated across a limited number of servers at a single location.
      - ✓ **Second:** it 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 mirror sites from a list. Mirrored Web sites have only a few replicas, which are more or less statically configured.

Replica Management

- Content replication & placement:
  - **Server-initiated replicas**:
    - In contrast to permanent replicas, server-initiated replicas are copies of a data store that exist to enhance performance, and 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.

Server-Initiated Replicas

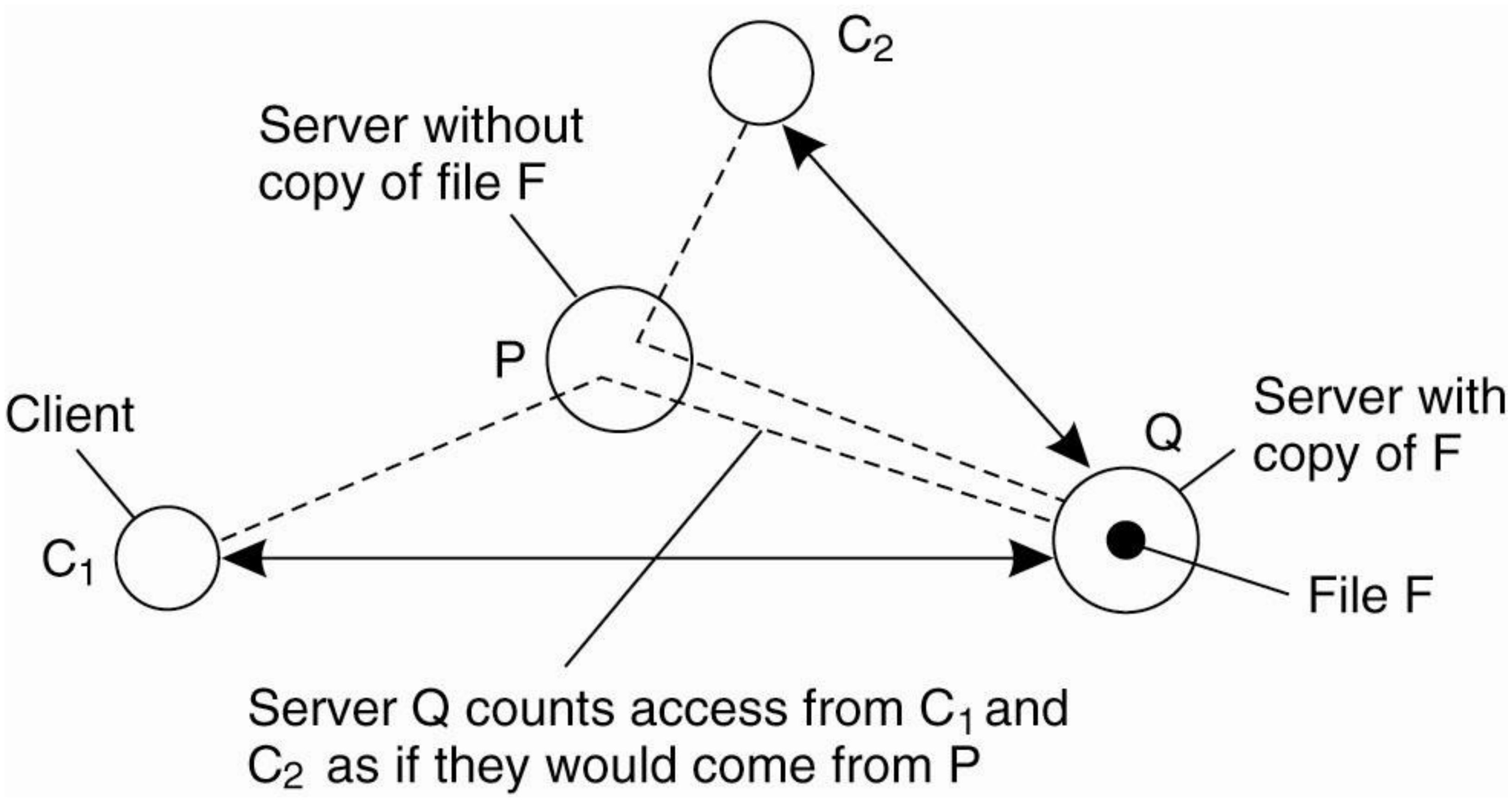

C_2
Server without copy of file F
P
Client
C_1
Q
Server with copy of F
File F
Server Q counts access from C_1 and C_2 as if they would come from P

Figure 7-18. Counting access requests from different clients.

Replica Management

- Content replication & placement:
  - **Client-initiated replicas**:
    - These 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.
    - Client *caches* are used only to improve access times to data.
    - When most operations involve only reading data, performance can be improved by letting the client store requested data in a nearby *cache*.
    - Such a cache could be located on the client's machine, or on a separate machine in the same local-area network as the client.
    - The next time that same data needs to be read, the client can simply fetch it from this local cache.
    - This scheme works fine as long as the fetched data have not been modified in the meantime.

Replica Management

- Content replication & placement:
  - **Client-initiated replicas**:
    - Data are generally kept in a *cache* for a limited amount of time, for example, to prevent extremely stale data from being used, or simply to make room for other data.
    - Whenever requested data can be fetched from the local *cache*, a **cache hit** is said to have occurred.
    - To improve the number of **cache hits**, *caches* can be shared between clients.
      - ✓ The underlying assumption is that a data request from client C_1 may also be useful for a request from another nearby client C_2.

Replica Management

- Content replication & placement:
  - **Client-initiated replicas**:
    - Placement of client *caches* is relatively simple:
      - ✓ A *cache* is normally placed on the same machine as its client, or otherwise on a machine shared by clients on the same local-area network (e.g., using Squid for web caching).
      - ✓ However, in some cases, extra levels of *caching* are introduced by system administrators by placing a shared *cache* between a number of departments or organizations, or even placing a *shared cache* for an entire region such as a province or country.
      - ✓ Yet another approach is to 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 [Noble et al., 1999].

Replica Management

- Content distribution:
  - Replica management also deals with propagation of (updated) content to the relevant replica servers.
  - **State vs. Operations**
    - Important design issues on what is actually to be propagated.
      - ✓ **Propagate only a notification of an update.**
      - ✓ **Transfer data from one copy to another.**
      - ✓ **Propagate the update operation to other copies.**

Replica Management

- Content distribution:
  - **Propagate only a notification:**
    - Propagating a notification is what *invalidation protocols* do.
    - In an *invalidation protocol*, other copies are informed that an update has taken place and that the data they contain are no longer valid.
    - The invalidation may specify which part of the data store has been updated, so that only part of a copy is actually invalidated.
    - The important issue is that no more than a notification is propagated.
    - Whenever an operation on an invalidated copy is requested, that copy generally needs to be updated first, depending on the specific consistency model that is to be supported.
    - The main advantage is that it uses little network bandwidth.
      - ✓ The only information that needs to be transferred is a specification of which data are no longer valid.
      - ✓ Such protocols generally work best when there are many update operations compared to read operations, that is, the **read-to-write ratio** is relatively small.

Replica Management

- Content distribution:
  - **Transfer data from a copy to another:**
    - It is useful when the **read-to-write ratio** is relatively high.
      - In that case, the probability that an update will be effective in the sense that the modified data will be read before the next update takes place is high.
    - Instead of propagating modified data, it is also possible to log the changes and transfer only those logs to save bandwidth.
    - In addition, transfers are often aggregated in the sense that multiple modifications are packed into a single message, thus saving communication overhead.

Replica Management

- Content distribution:
  - **Propagate the update operation to other copies.**
    - This approach is not to transfer any data modifications at all, but to tell each replica which update operation it should perform (and sending only the parameter values that those operations need).
    - This is also referred to as **active replication**, assumes that each replica is represented by a process capable of “actively” keeping its associated data up to date by performing operations [Schneider, 1990].
    - The main benefit of active replication is that updates can often be propagated at minimal bandwidth costs, provided the size of the parameters associated with an operation are relatively small.
    - Moreover, the operations can be of arbitrary complexity, which may allow further improvements in keeping replicas consistent.
    - On the other hand, more processing power may be required by each replica, especially in those cases when operations are relatively complex.
      - ✓ ....

Replica Management

- Content distribution:
  - **Pull vs. 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 generally applied when strong consistency is required.
      - ✓ This need for strong consistency is related to the fact that permanent and
      - ✓ server-initiated replicas, and large shared caches, are often shared by many clients, which mainly perform read operations.
      - ✓ Thus, the read-to-update ratio at each replica is relatively high.
      - ✓ Push-based protocols are efficient in the sense that every pushed update can be expected to be of use for at least one, but perhaps more readers. In addition, push-based protocols make consistent data immediately available when asked for.

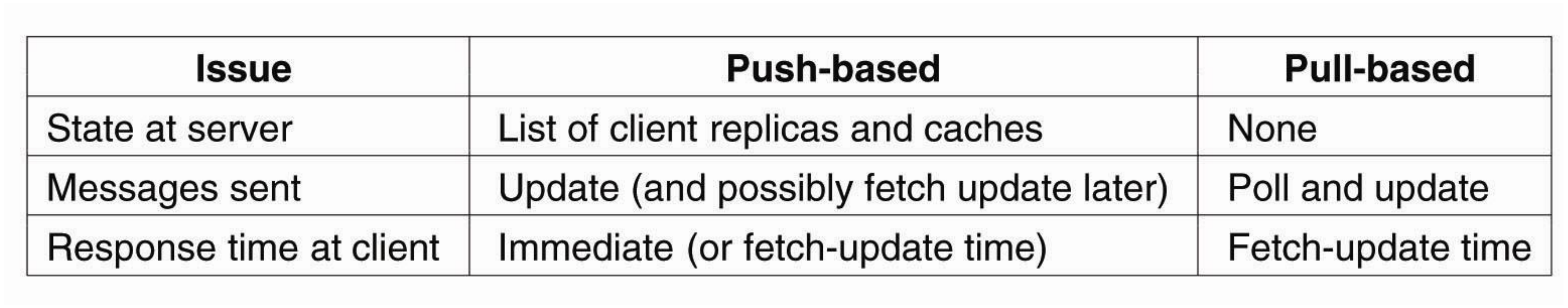
Replica Management

- Content distribution:
  - **Pull vs. 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** (aka **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.

Replica Management

- Content distribution:
  - **Pull vs. push protocols**
    - A pull-based approach is efficient when the read-to-update ratio is relatively low.
    - This is often the case with (nonshared) client caches, which have
    - only one client.
    - However, even when a cache is shared by many clients, a pull-based approach may also prove to be efficient when the cached data items are rarely shared.
    - The **main drawback** of a **pull-based strategy** in comparison to a **push-based approach** is that the response time increases in the case of a cache miss.

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.

Replica Management

- Content distribution:
  - **Pull vs. push protocols**
    - The trade-off of these approaches have led to a hybrid form of update propagation based on leases.
      - In the case of replica management, 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.
      - An alternative is that a client requests a new lease for pushing updates when the previous lease expires.
    - Leases, originally introduced by Gray and Cheriton [1989], provide a convenient mechanism for dynamically switching between a push-based and pull-based strategy.
    - There are three types of leases:
      - Age-based leases
      - Renewal-frequency-based leases
      - State-based lease

Replica Management

- Content distribution:
  - **Pull vs. push protocols**
    - **Age-based leases**
      - The underlying assumption is that data that have not been modified for a long time can be expected to remain unmodified for some time yet to come.
      - This assumption has shown to be reasonable in the case of Web-based data and regular files.
      - By granting long-lasting leases to data items that are expected to remain unmodified, the number of update messages can be strongly reduced compared to the case where all leases have the same expiration time.

Replica Management

- Content distribution:
  - **Pull vs. push protocols**
    - **Renewal-frequency-based leases**
      - With this approach, a server will hand out a long-lasting lease to a client whose cache often needs to be refreshed.
      - On the other hand, a client that asks only occasionally for a specific data item will be handed a short-term lease for that item.
      - The effect of this strategy is that the server essentially keeps track only of those clients where its data are popular; moreover, those clients are offered a high degree of consistency.

Replica Management

- Content distribution:
  - **Pull vs. push protocols**
    - **State-based leases**
      - The last criterion is that of state-space overhead at the server.
      - When the server realizes that it is gradually becoming overloaded, it lowers the expiration time of new leases it hands out to clients.
      - The effect of this **state-based lease strategy** is that server needs to keep track of fewer clients as leases expire more quickly.
      - In other words, the server dynamically switches to **a more stateless mode of operation**, thereby expecting to offload itself so that it can handle requests more efficiently.
      - The obvious drawback is that it may need to do more work when the read-to-update ratio is high.

Consistency Protocols

- A **consistency protocol** describes an implementation of a specific consistency model.
  - **Sequential consistency**
    - Primary-based protocols
    - Replicated-write protocols

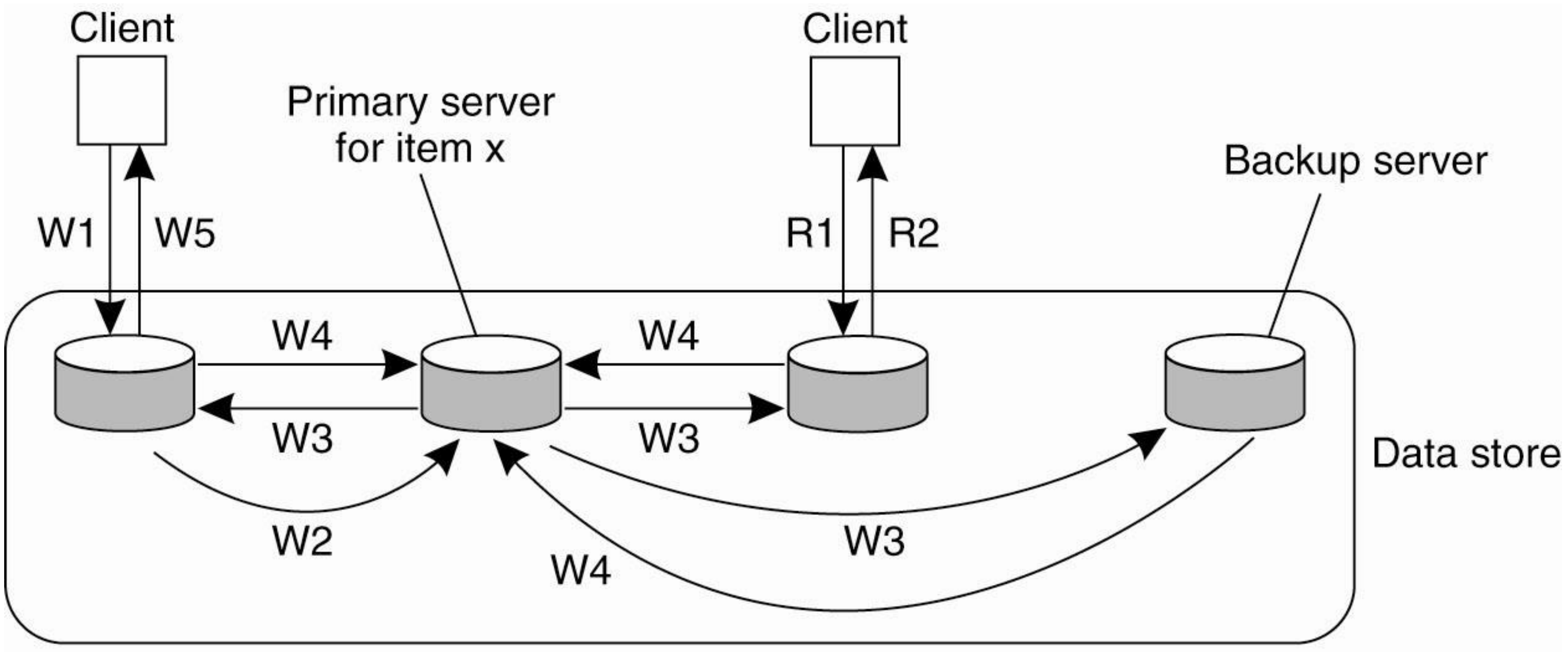
Consistency Protocols

- **Sequential consistency**: Primary-based protocols
  - In practice, we see that distributed applications generally follow consistency models that are relatively easy to understand.
    - **As consistency models become difficult to understand for application developers, they use to be ignored.**
    - Simplicity is appreciated!
  - **Primary-based protocols** have prevailed due to their simplicity.
    - In these protocols, each data item **x** in the data store has an associated **primary**, which is a *process* responsible for coordinating write operations on **x**.
    - A distinction can be made as to whether the **primary** is fixed at a remote server or if write operations can be carried out locally after moving the **primary** to the process where the write operation is initiated.

Consistency Protocols

- **Sequential consistency**: Primary-based protocols
  - **Remote-write protocols**
    - The simplest **primary-based protocol** that supports replication is the one in which all write operations need to be forwarded to a fixed single server. Read operations can be carried out locally.
    - Such schemes are also known as **primary-backup protocols** [Budhijara et al., 1993].
      - In a **primary-backup protocol**, a process wanting to perform a write operation on data item **x**, forwards that operation to the primary server for **x**.
      - The primary performs the update on its local copy of **x**, and subsequently forwards the update to the backup servers.
      - Each backup server performs the update as well, and sends an acknowledgment to the primary.
      - When all backups have updated their local copy, the primary sends an acknowledgment to the initial process, which, in turn, informs the client.

Remote-Write Protocols



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

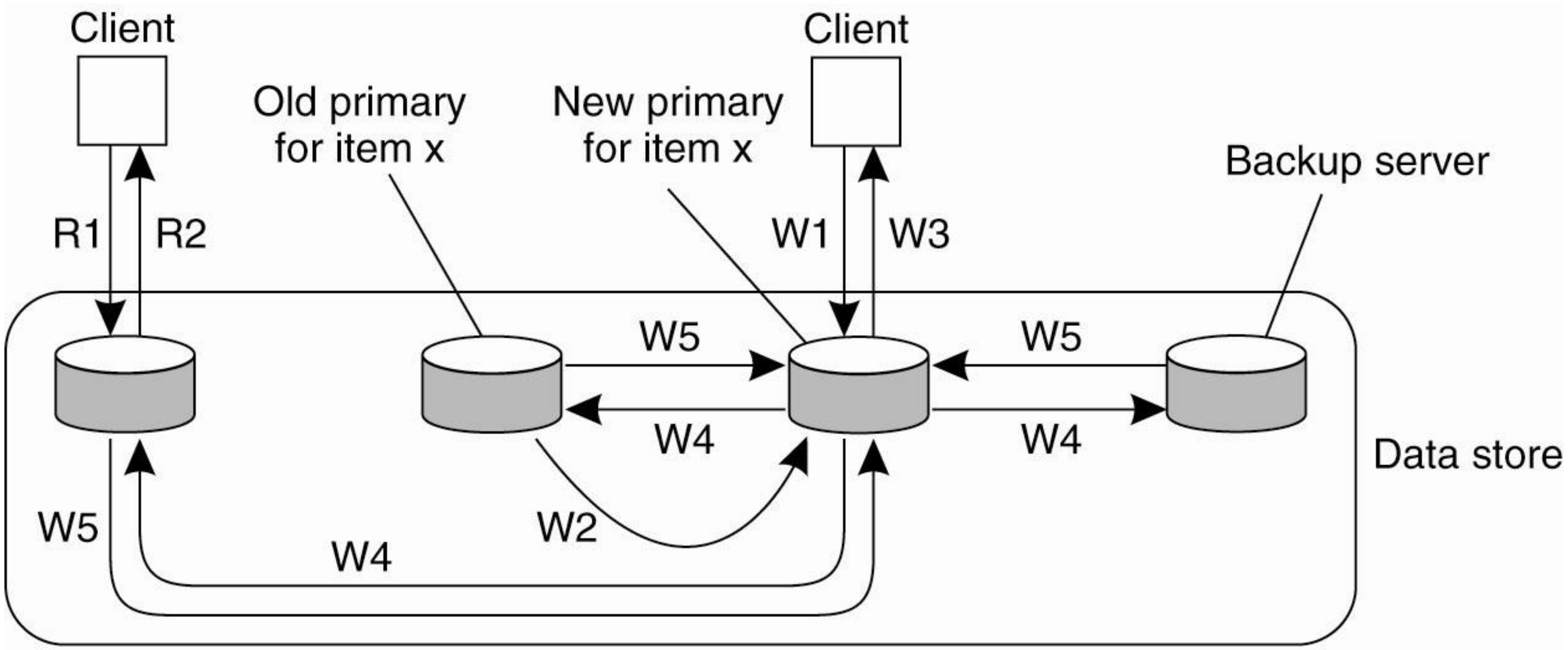
R1. Read request
R2. Response to read

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

Consistency Protocols

- **Sequential consistency**: Primary-based protocols
  - **Local-write protocols**
    - A variant of primary-backup protocols is one in which the primary copy migrates between processes that wish to perform a write operation.
    - Whenever a process wants to update data item **x**, it locates the primary copy of **x**, and subsequently moves it to its own location.
    - The main advantage of this approach is that multiple, successive write operations can be carried out locally, while reading processes can still access their local copy.

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.

Consistency Protocols

- **Sequential consistency**: Replicated-based protocols
  - In this approach, write operations can be carried out at multiple replicas instead of only one.
  - A distinction can be made between:
    - **Active replication:** an operation is forwarded to all replicas.
    - **Quorum protocols**: based on majority voting.

Consistency Protocols

- **Sequential consistency**: Replicated-based protocols
  - **Active replication**
    - Each replica has an associated process that performs update operations.
    - Updates are generally propagated by means of the write operation that causes the update.
      - The operation is sent to each replica; however, it is also possible to send the update.
    - One problem is that operations need to be performed in the same order everywhere.
      - Consequently, what is needed is a total-ordered multicast mechanism.
      - A practical approach to accomplish total ordering is by means of a central coordinator, also called a **sequencer**.
      - One approach is to first forward each operation to the sequencer, which assigns it a unique sequence number and subsequently forwards the operation to all replicas.
      - Operations are carried out in the order of their sequence number.

Consistency Protocols

- **Sequential consistency**: Replicated-based protocols
  - **Quorum-based protocols**
    - A different approach to supporting replicated writes is to use **voting**.
    - The basic idea is to require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item.
    - For example, consider a distributed file system and suppose that a file is replicated on *N* servers.
      - We could make a rule stating that to update a file, a client must first contact at least half the servers plus one (a majority) and get them to agree to do the update.
      - Once they have agreed, the file is changed and a new version number is associated with the new file.
      - The version number is used to identify the version of the file and is the same for all the newly updated files.

Consistency Protocols

- **Sequential consistency**: Replicated-based protocols
  - **Quorum-based protocols**
    - A different approach to supporting replicated writes is to use **voting**.
    - The basic idea is to require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item.
    - For example, consider a distributed file system and suppose that a file is replicated on *N* servers.
      - To read a replicated file, a client must also contact at least half the servers plus one and ask them to send the version numbers associated with the file.
      - If all the version numbers are the same, this must be the most recent version because an attempt to update only the remaining servers would fail because there are not enough of them.
        - For example, if there are five servers and a client determines that three of them have version 8, it is impossible that the other two have version 9.
        - After all, any successful update from version 8 to version 9 requires getting three servers to agree to it, not just two.

Quorum-Based Protocols

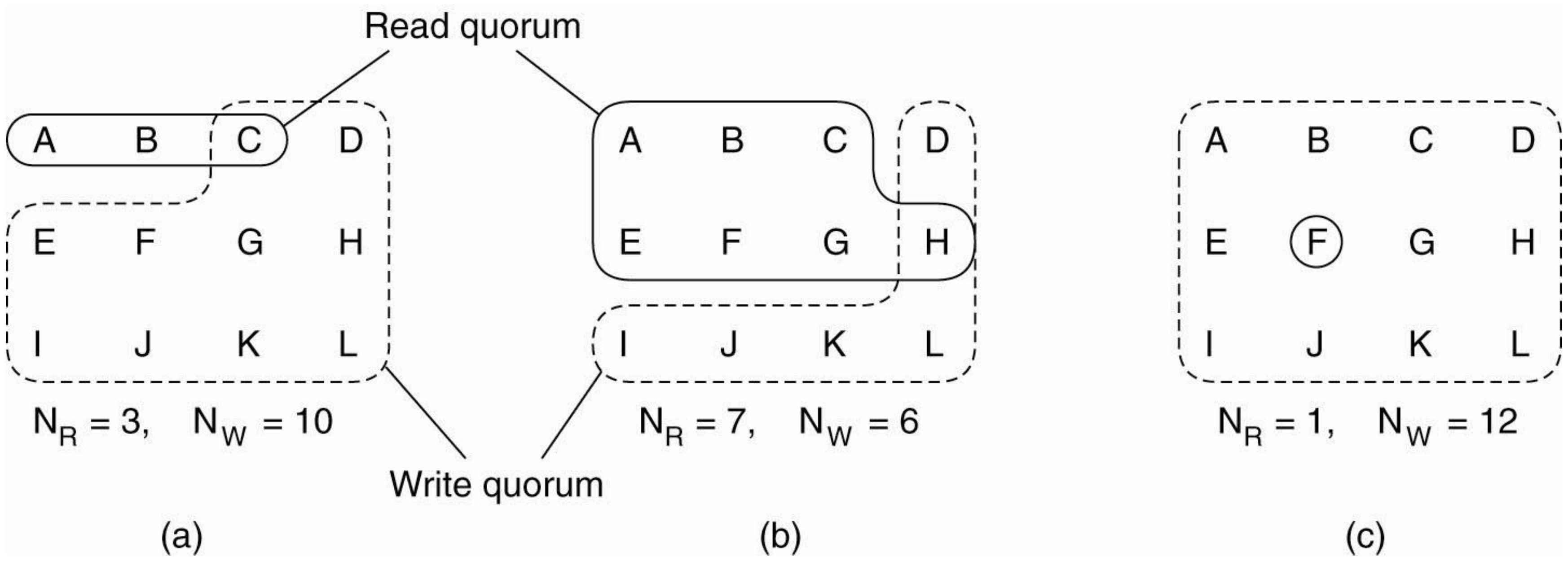


Figure 7-22. Three examples of the voting algorithm. (a) A correct choice of read and write set. (b) A choice that may lead to write-write conflicts. (c) A correct choice, known as ROWA (read one, write all).