JC4001: Distributed Systems

Topic 4: Consistency and Replication

Xiaonan Liu
xiaonan.liu@abdn.ac.uk

Content

- Introduction
  - Reasons for Replication
- Data-centric Consistency Models
  - Consistent Ordering of Operations
  - Eventual Consistency
  - Continuous Consistency
- Client-centric Consistency Models
  - Monotonic Reads
  - Monotonic Writes
  - Read Your Writes
  - Writes Follow Reads
- Replica Management
  - Finding the best Server Location
  - Content Replication and Placement
  - Content Distribution
  - Managing Replicated Objects

Content

- Consistency Protocols
  - Sequential consistency: Primary-based protocols
  - Sequential consistency: Replicated-write protocols
  - Implementing continuous consistency
  - Implementing client-centric consistency
- Example: Caching and replication in the Web
  - Cooperative caching
  - Web-cache consistency
  - CDN
  - Alternatives for caching and replication

Introduction

- Reasons for Replication

Why replicate

Assume a simple model in which we make a copy of a specific part of a system (meaning code and data).

- Increase reliability: If a file system is replicated, you can keep working even if one copy crashes by switching to another. Multiple copies also help protect against data corruption. For example, with three copies of a file, if one write operation fails, we can trust the result from the other two.
- Performance: Replication enhances performance when a system needs to scale, either in size or across a wide area. For size. if more processes need to access data managed by one server, replicating the server and spreading the load can help. For geography, placing copies of data closer to where they are used reduces access time and improves perceived performance. However, this might consume more network bandwidth to keep replicas updated.

Introduction

- Reasons for Replication

The problem

Having multiple copies, means that when any copy changes, that change should be made at all copies: replicas need to be kept the same, that is, be kept consistent.

Take Web pages, for example. To speed up access, browsers often cache pages locally. If you revisit a page. the browser quickly shows the stored version, But if you always want the latest version, you might miss updates. If a page changes, the cached version won't reflect those changes.

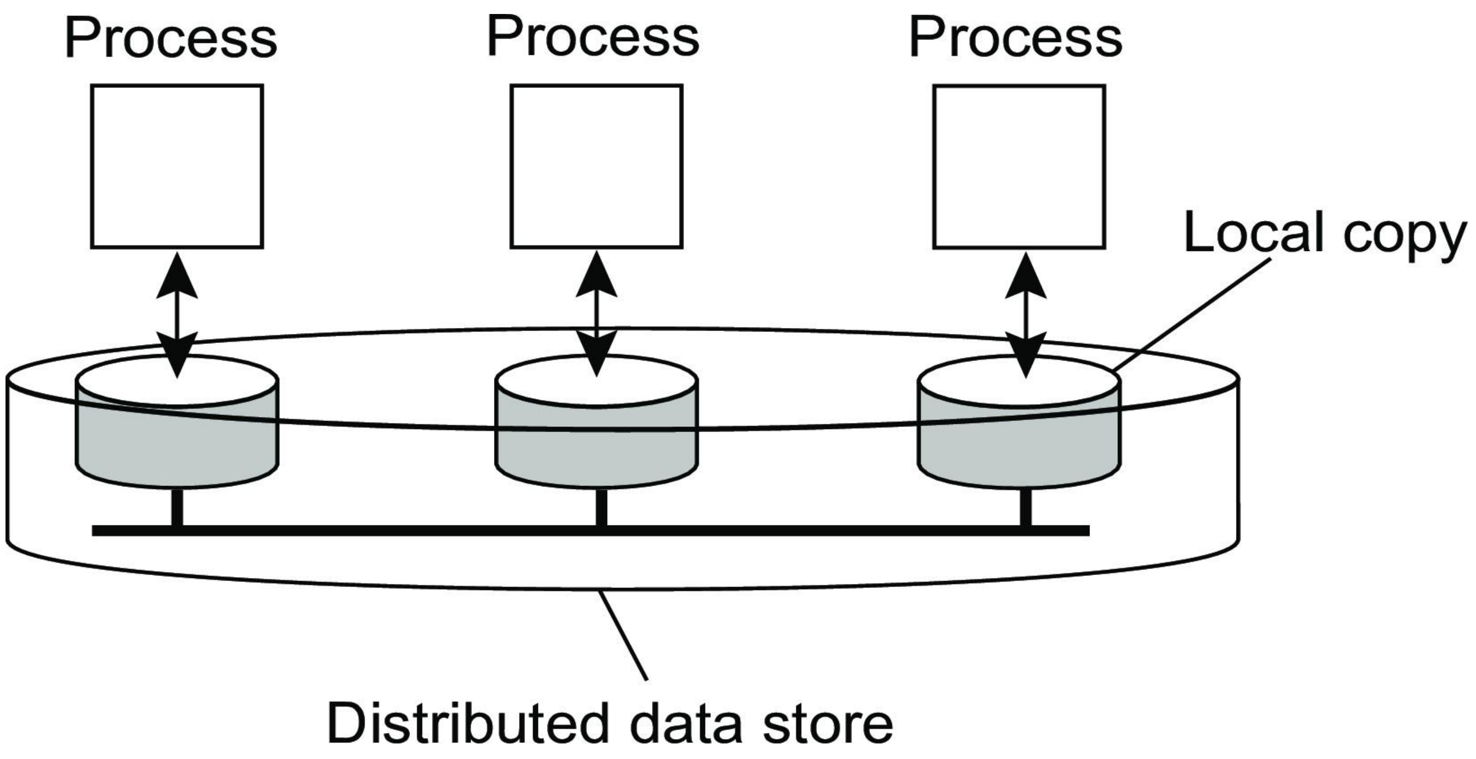
Solutions:

1. Not caching at all, letting the server handle everything. But this can lead to slow access if there’s no nearby replica.
2. Update or invalidate cached copies, but tracking all caches and sending updates can slow down the server.

Data-centric Consistency Models

Data Store

Refer to how data is read and written across systems like shared memory, databases, or file systems. A data store may be physically distributed across multiple machine. In particular, each process that can access data from the store is assumed to have a local (or nearby) copy available of the entire store. When data changes, or "write operations" happen, those changes are sent to all the other copies. If a data operation doesn't change anything, it's just a read.



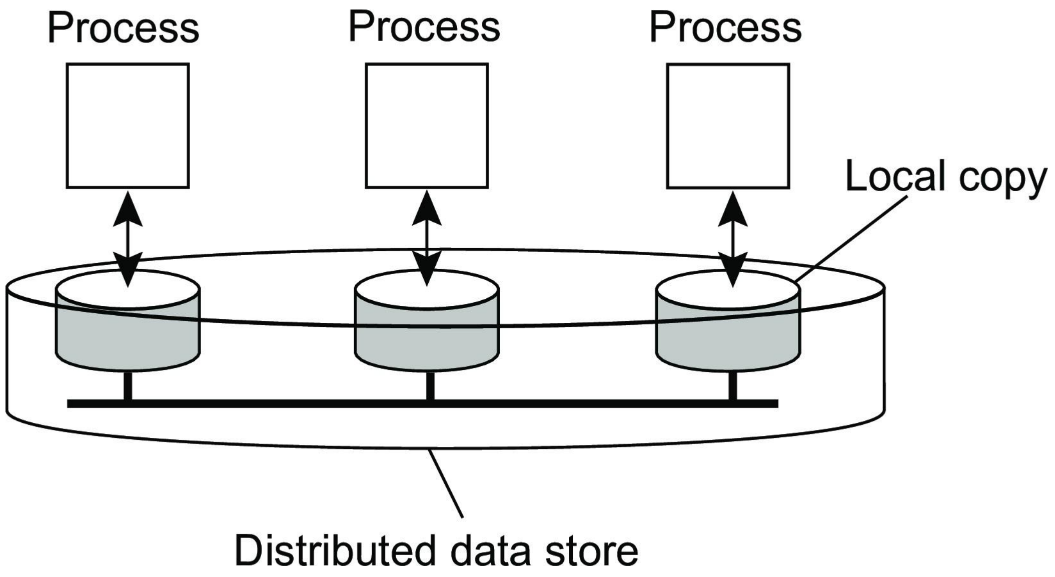
Data-centric Consistency Models

Consistency model

A contract between the processes using the data and the store itself, where the consistency model guarantees if processes follow certain rules, the data store will work correctly. For example, when a process reads data, it expects to see the latest changes. Without a global clock, it is difficult to define precisely which write operation is the latest one. So, we have different consistency models that set rules on what you can expect when reading data. Some models have strict rules, making them easy to work with but potentially slower. Others do not have strict rules, which can improve performance but make things more difficult to handle. Namely, the stricter models are user-friendly but might not perform well, while the more relaxed ones are faster but require more effort to manage. That's the trade-off!

Essential

A data store is a distributed collection of storages:



Data-centric Consistency Models

- Consistent Ordering of Operations

Parallel programming

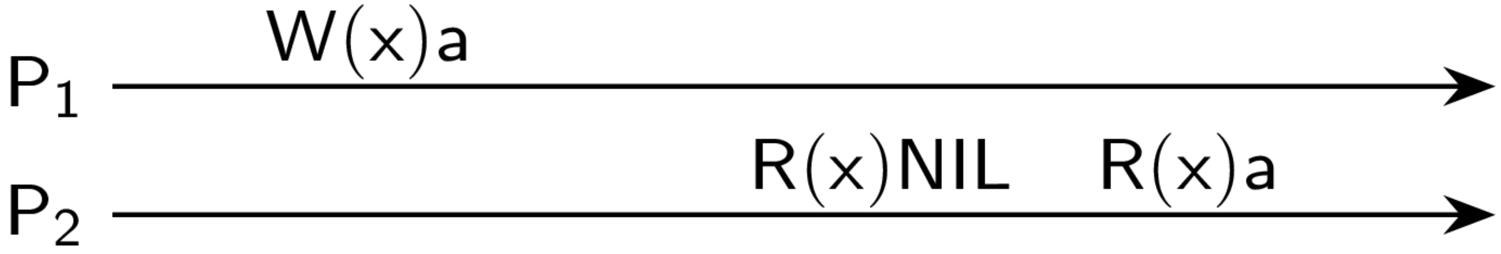
When you have multiple processes sharing and accessing resources at the same time in parallel or distributed computing, you need a way to manage those concurrent accesses, especially when those resources are replicated.

Read and write operations

- $W_i(x)a$: Process P_i writes value a to x
- $R_i(x)b$: Process P_i reads value b from x
- All data items initially have value *NIL*

Possible behavior

We omit the index when possible and draw according to time (x-axis):



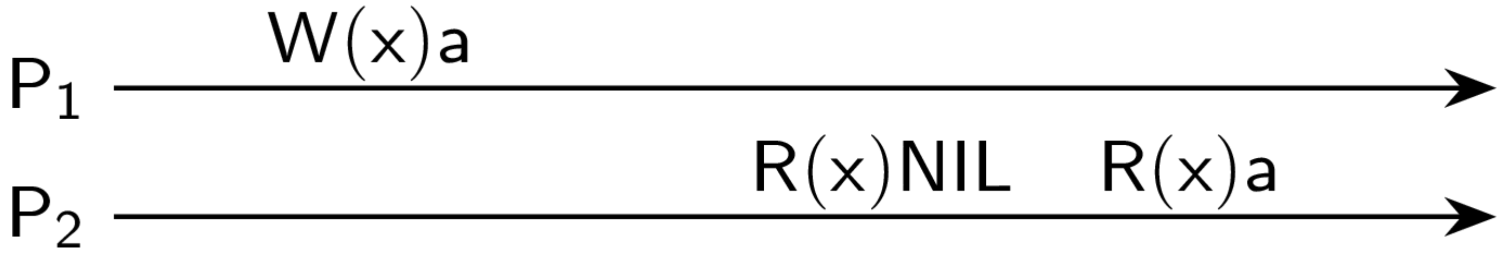
Data-centric Consistency Models

- Consistent Ordering of Operations

Possible behavior

We omit the index when possible and draw according to time (x-axis):

Process P1 writes the value a to data item x. This write operation happens first on P1's local copy of the data store and then gets propagated to other copies. Later, process P2 reads x and first gets NIL, and then, after some time, it reads a from its local copy. This delay in updating P2's copy is totally normal and expected.



Data-centric Consistency Models

- Consistent Ordering of Operations: Sequential Consistency

Definition

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

It is when a data store behaves as if all read and write operations happened in a specific order. Each process sees these operations in the sequence set by its own program, even if they're on different machines. What's important here is that all processes view the operations in the same order, but it doesn’t matter when they actually happened in real-time.

Data-centric Consistency Models

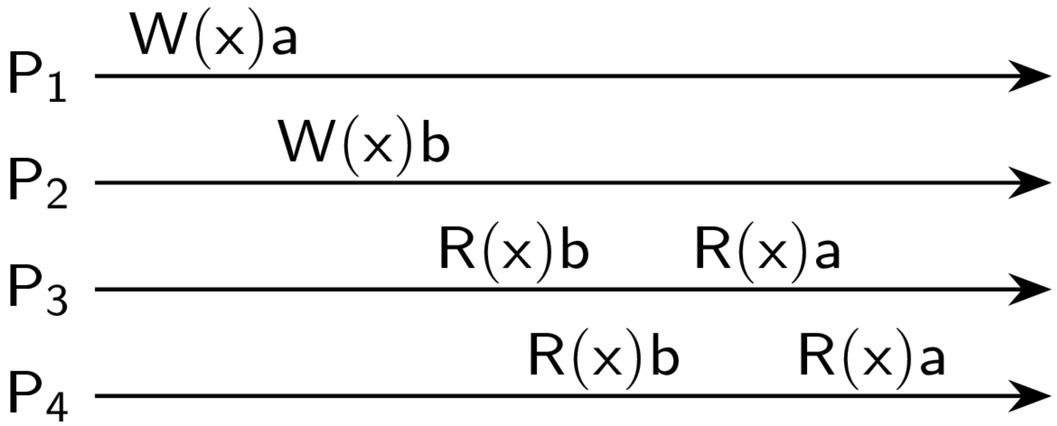
- Consistent Ordering of Operations: Sequential Consistency

A sequentially consistent data store

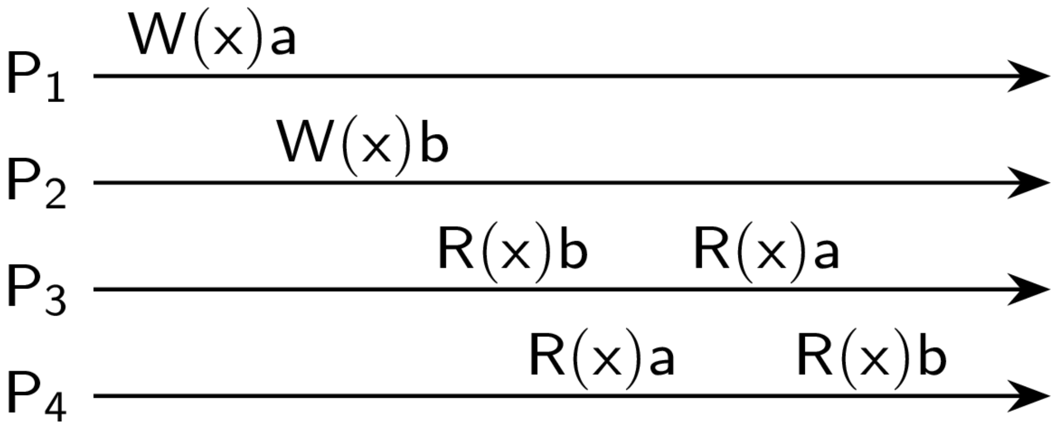
Process P1 writes value 'a' to x first. Later, process P2 writes 'b' to x. However, processes P3 and P4 read 'b' first and then 'a'. So, it seems like P2's write happened before P1's write.

A data store that is not sequentially consistent

It breaks the rule of sequential consistency because not all processes see the same order of writes. For P3, it looks like x was changed to 'b' first and then to 'a'. But for P4, the final value of x is 'b’. It is because P4 first reads a and then reads b, so it shows that P1’s write happened before P2’s write.



A sequentially consistent data store



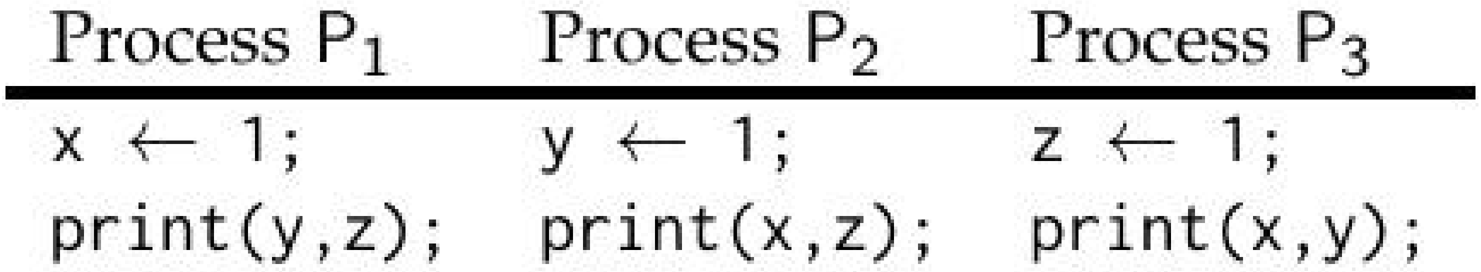
A data store that is not sequentially consistent

Data-centric Consistency Models

- Consistent Ordering of Operations: Sequential Consistency

Example

Three concurrent processes (initial values: 0)



Process P_1	Process P_2	Process P_3
x ← 1;	y ← 1;	z ← 1;
print(y,z);	print(x,z);	print(x,y);

Example execution sequences

Execution 1	Execution 2	Execution 3	Execution 4
P_1: x ← 1;	P_1: x ← 1;	P_2: y ← 1;	P_2: y ← 1;
P_1: print(y,z);	P_2: y ← 1;	P_3: z ← 1;	P_1: x ← 1;
P_2: y ← 1;	P_2: print(x,z);	P_3: print(x,y);	P_3: z ← 1;
P_2: print(x,z);	P_1: print(y,z);	P_2: print(x,z);	P_2: print(x,z);
P_3: z ← 1;	P_3: z ← 1;	P_1: x ← 1;	P_1: print(y,z);
P_3: print(x,y);	P_3: print(x,y);	P_1: print(y,z);	P_3: print(x,y);
Prints: 001011 *Signature:* 00 10 11	*Prints:* 101011 *Signature:* 10 10 11	*Prints:* 010111 *Signature:* 11 01 01	*Prints:* 111111 *Signature:* 11 11 11
(a)	(b)	(c)	(d)

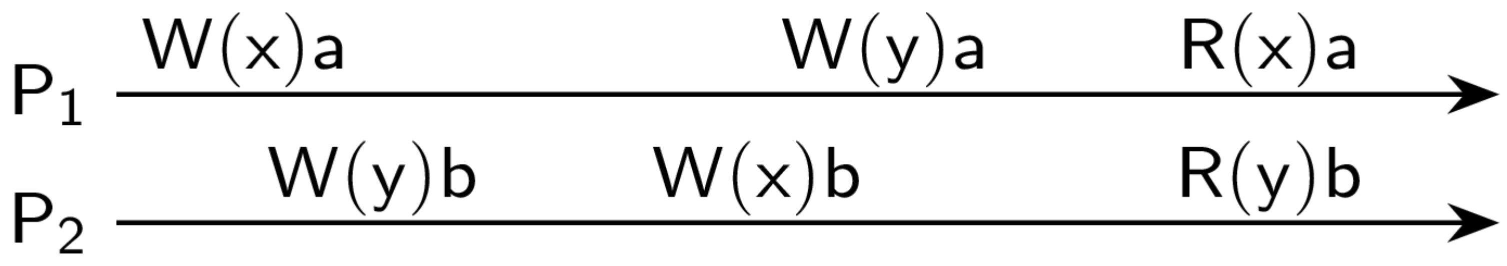
Data-centric Consistency Models

- Consistent Ordering of Operations: Sequential Consistency

How tricky can it get?

Sequential consistency is a crucial model because it's the easiest to understand when developing concurrent applications. It matches our expectations when multiple programs work on shared data simultaneously. However, implementing it isn't easy.

Seemingly okay



P_1: W(x)a W(y)a R(x)a

P_2: W(y)b W(x)b R(y)b

But not really (don't forget that P_1 and P_2 act concurrently)

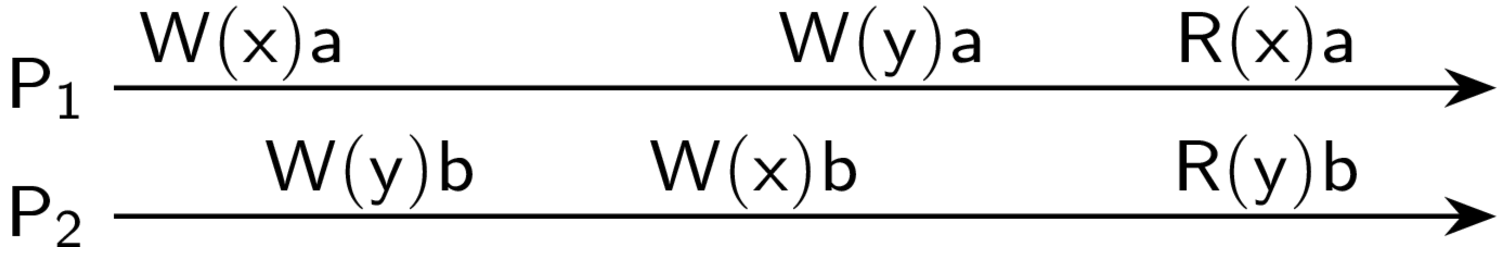
Possible ordering of operations	Result	
$W_1(x)a$; $W_1(y)a$; $W_2(y)b$; $W_2(x)b$	$R_1(x)b$	$R_2(y)b$
$W_1(x)a$; $W_2(y)b$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a$; $W_2(y)b$; $W_2(x)b$; $W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_1(x)a$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_1(x)a$; $W_2(x)b$; $W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_2(x)b$; $W_1(x)a$; $W_1(y)a$	$R_1(x)a$	$R_2(y)a$

Data-centric Consistency Models

- Consistent Ordering of Operations: Sequential Consistency

How tricky can it get?

Consider two variables, x and y. If P1 reads 'a' from x and P2 reads 'b' from y, it seems OK at first. But there’s no way to order these operations on x and y to make both reads consistent when considering each process's order. This means the operations by P1 and P2 aren’t easily matched, showing that sequential consistency doesn't always combine well across multiple data items. As shown in the table, there are many possible ordering of operations, and lead to many results.



But not really (don’t forget that P_1 and P_2 act concurrently)

Possible ordering of operations	Result	
$W_1(x)a; W_1(y)a; W_2(y)b; W_2(x)b$	$R_1(x)b$	$R_2(y)b$
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_2(x)b; W_1(x)a; W_1(y)a$	$R_1(x)a$	$R_2(y)a$

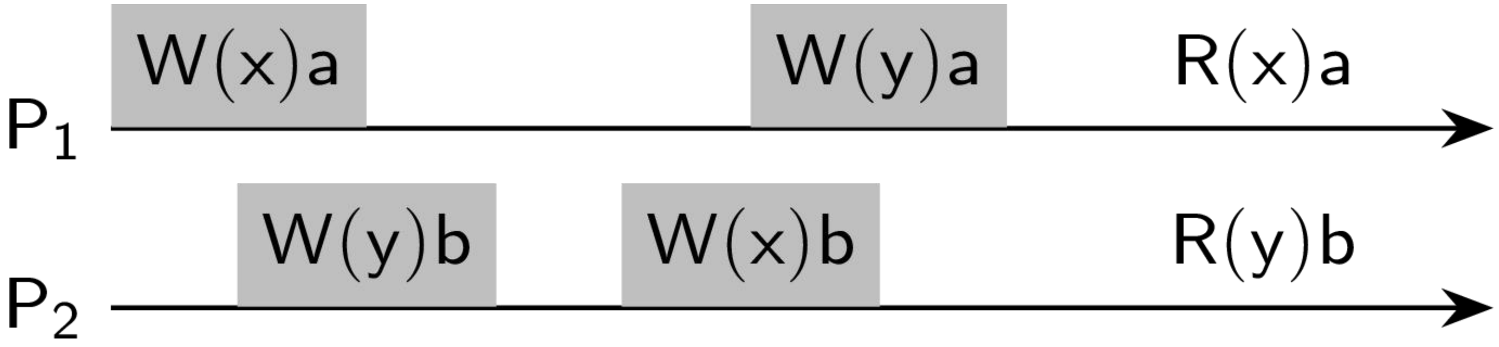
Data-centric Consistency Models

- Consistent Ordering of Operations: Sequential Consistency

Linearizability

It assumes operations happen instantly at some point between their start and end. In our example, linearizability means each write operation's effects should show up within the time it's executed. For instance, if W2(y)b finishes before W1(y)a starts, y will end up as 'a'. Similarly, x will be 'b' if W1(x)a finishes before W2(x)b starts.

Operations complete within a given time (shaded area)



With better results

Possible ordering of operations	Result	
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$

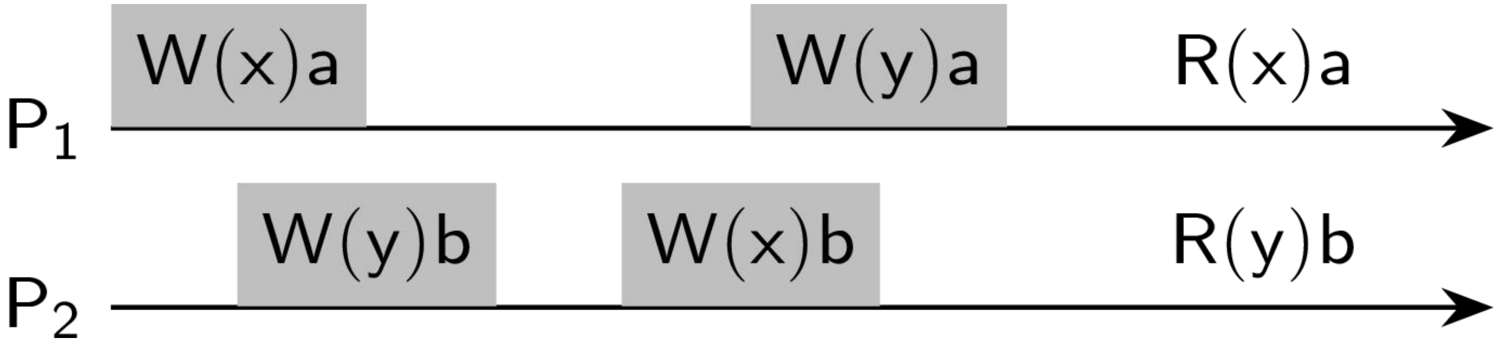
Data-centric Consistency Models

- Consistent Ordering of Operations: Sequential Consistency

Linearizability

The shaded area represents the time during which each operation is being executed. Linearizability says that the effect of an operation should happen somewhere within that shaded time frame. Essentially, this means that once a write operation finishes, its results should be spread to the other data stores.

Operations complete within a given time (shaded area)



With better results

Possible ordering of operations				Result	
$W_1(x)a;$	$W_2(y)b;$	$W_1(y)a;$	$W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a;$	$W_2(y)b;$	$W_2(x)b;$	$W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b;$	$W_1(x)a;$	$W_1(y)a;$	$W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b;$	$W_1(x)a;$	$W_2(x)b;$	$W_1(y)a$	$R_1(x)b$	$R_2(y)a$

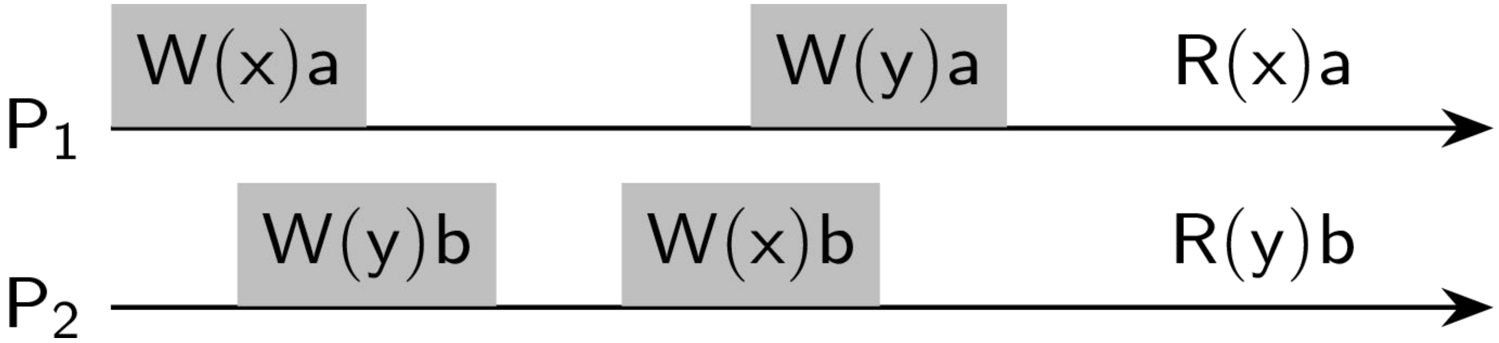
Data-centric Consistency Models

- Consistent Ordering of Operations: Sequential Consistency

Linearizability

W2(y)b finishes before W1(y)a starts, so y ends up with the value a. Similarly, W1(x)a finishes before W2(x)b starts, so x ends up with the value b. Implementing linearizability, especially on many-core systems, can slow things down significantly, but it makes programming much easier. So, there’s a trade-off between performance and simplicity.

Operations complete within a given time (shaded area)



With better results

Possible ordering of operations	Result	
$W_1(x)a; W_2(y)b;\ W_1(y)a;\ W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b;\ W_2(x)b;\ W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a;\ W_1(y)a;\ W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a;\ W_2(x)b;\ W_1(y)a$	$R_1(x)b$	$R_2(y)a$

Data-centric Consistency Models

- Consistent Ordering of Operations: Causal Consistency

Definition

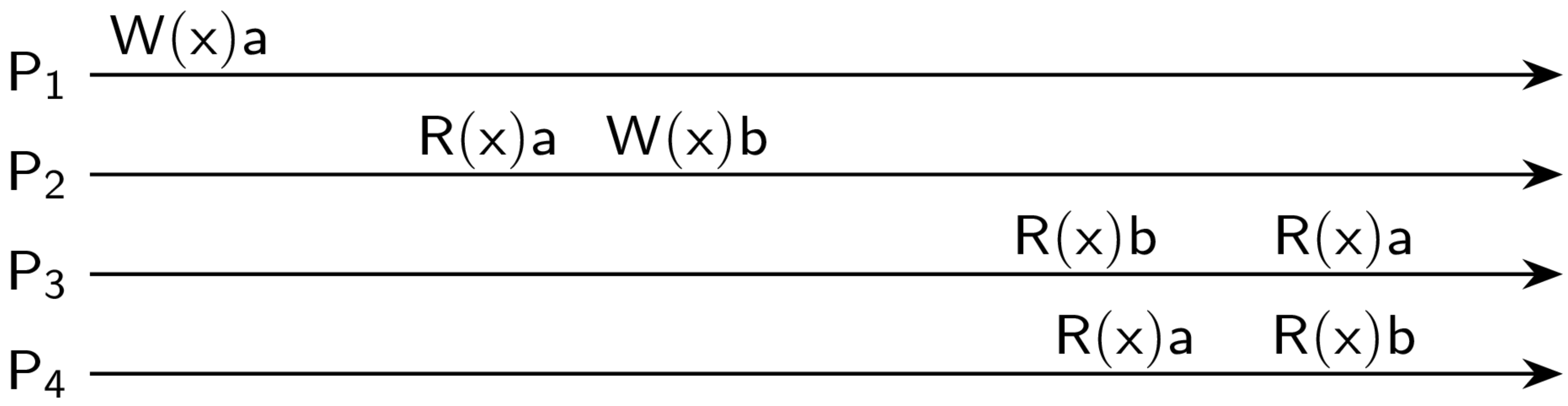
It is a bit weaker than sequential consistency. It makes a distinction between events that might be causally related and those that are not related. Basically, if event b is caused or influenced by an earlier event a, causality means everyone should first see a, then see b. Imagine a simple interaction using a distributed shared database. Let's say process P1 writes a data item x, and then P2 reads x and writes y. The reading of x and the writing of y could be causally related because y might depend on the value of x that P2 read (which was written by P1). However, if two processes independently write two different data items at the same time, those events aren't causally related. We call these operations concurrent.

Data-centric Consistency Models

- Consistent Ordering of Operations: Causal Consistency

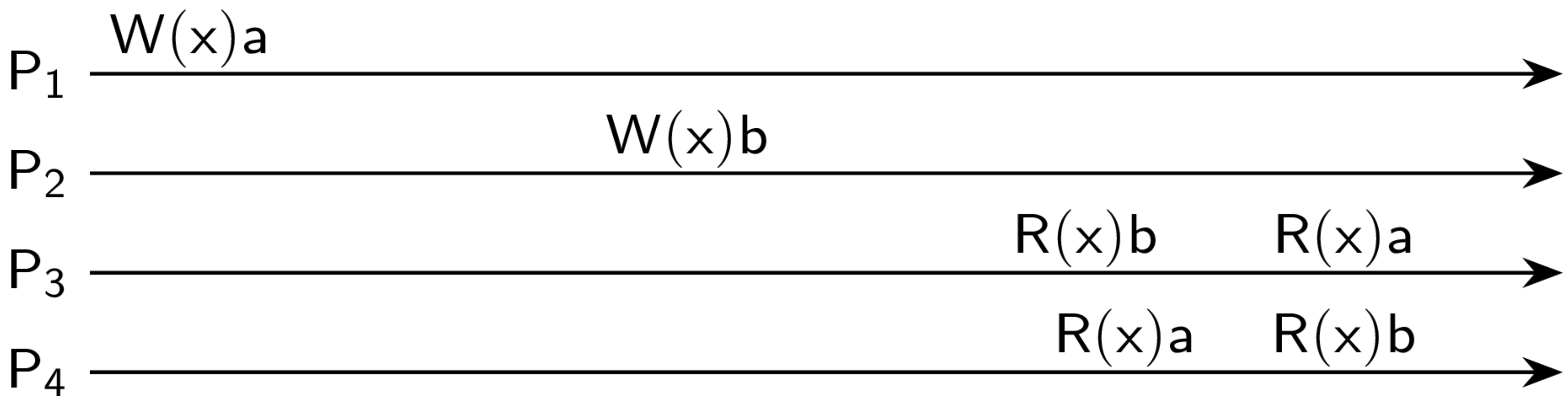
Rule of causal consistency

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 by different processes.



A violation of a causally-consistent store

W2(x)b potentially depending on W1(x)a because writing b to x might result from a computation involving the value read by R2(x)a. These two writes are causally related, so all processes need to see them in the same order.



The read has been removed, so W1(x)a and W2(x)b are now concurrent writes.

A correct sequence of events in a causally-consistent store

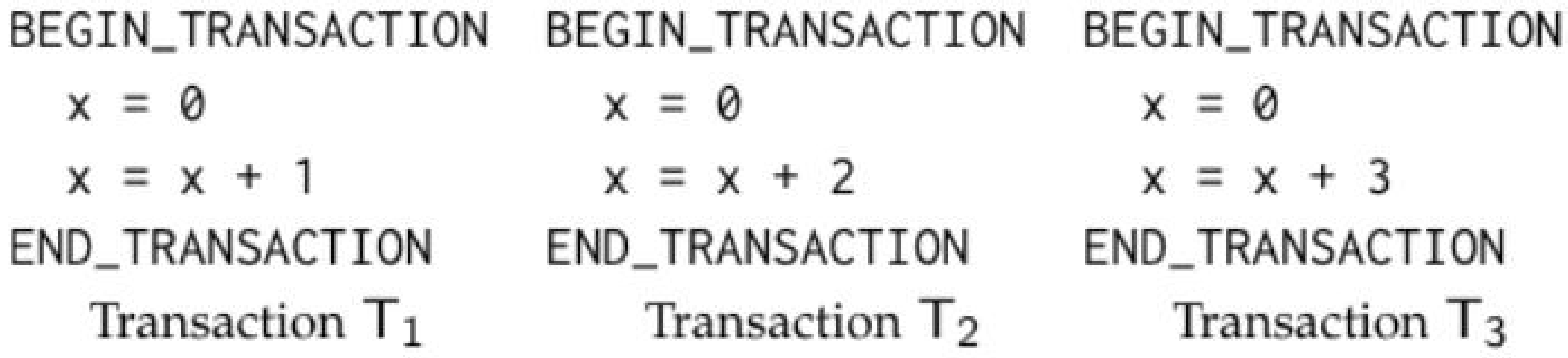
Data-centric Consistency Models

- Consistent Ordering of Operations: Consistency models, serializability, transactions

Overwhelming, but often already known

Transactions group: a series of read and write operations, and guarantees they are executed in the order they appear. When multiple transactions operate on the same dataset, the final values can be explained by a specific ordering of those transactions.

The keyword is serializability. Again, from the world of transactions: can we order the execution of all operations in a set of transactions in such a way that the final result matches a serial execution of those transactions?



```
BEGIN_TRANSACTION    BEGIN_TRANSACTION    BEGIN_TRANSACTION
  x = 0                x = 0                x = 0
  x = x + 1            x = x + 2            x = x + 3
END_TRANSACTION      END_TRANSACTION      END_TRANSACTION
```

Transaction T_1 Transaction T_2 Transaction T_3

A number of schedules

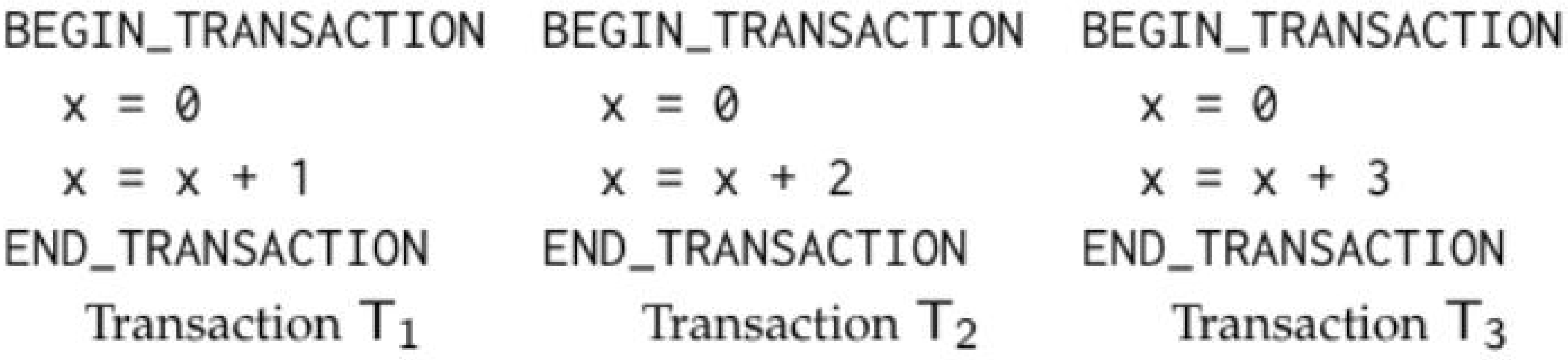
	Time --→						
S1	x = 0	x = x + 1	x = 0	x = x + 2	x = 0	x = x + 3	Legal
S2	x = 0	x = 0	x = x + 1	x = x + 2	x = 0	x = x + 3	Legal
S3	x = 0	x = 0	x = x + 1	x = 0	x = x + 2	x = x + 3	Illegal
S4	x = 0	x = 0	x = x + 3	x = 0	x = x + 1	x = x + 2	Illegal

Data-centric Consistency Models

- Consistent Ordering of Operations: Consistency models, serializability, transactions

Overwhelming, but often already known

In T1 T2 and T3, you don't need to know exactly what's being computed, just that we're dealing with read and write operations. Each transaction Ti can be represented as a series like ⟨ Wi(x); Ri(x); Wi(x)⟩ . To ensure consistency, the transaction system has to create legal schedules, which are orderings of the read and write operations from the three transactions that result in a final outcome equivalent to a serial execution.



```
BEGIN_TRANSACTION   BEGIN_TRANSACTION   BEGIN_TRANSACTION
  x = 0               x = 0               x = 0
  x = x + 1           x = x + 2           x = x + 3
END_TRANSACTION     END_TRANSACTION     END_TRANSACTION
```

Transaction T_1 Transaction T_2 Transaction T_3

A number of schedules

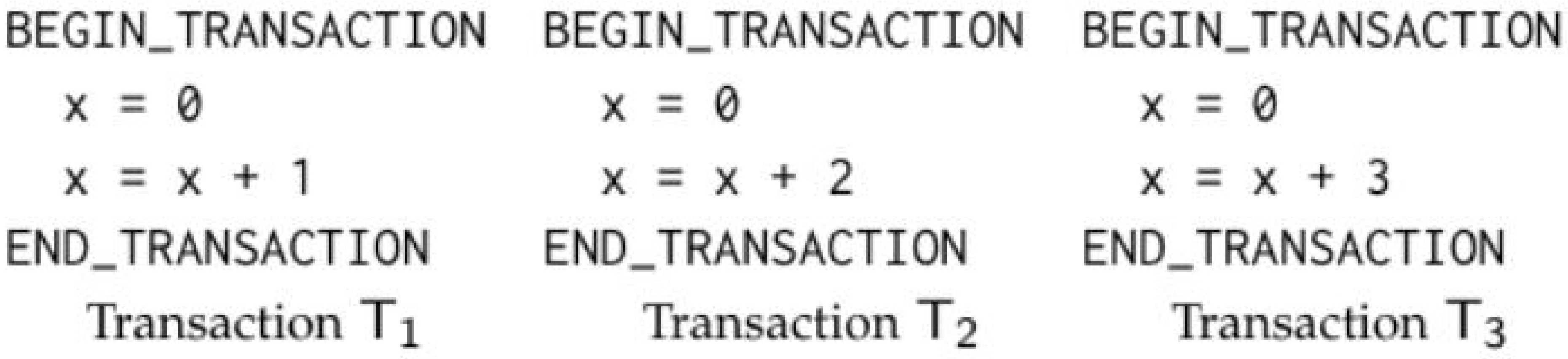
	Time --→						
S1	x = 0	x = x + 1	x = 0	x = x + 2	x = 0	x = x + 3	Legal
S2	x = 0	x = 0	x = x + 1	x = x + 2	x = 0	x = x + 3	Legal
S3	x = 0	x = 0	x = x + 1	x = 0	x = x + 2	x = x + 3	Illegal
S4	x = 0	x = 0	x = x + 3	x = 0	x = x + 1	x = x + 2	Illegal

Data-centric Consistency Models

- Consistent Ordering of Operations: Consistency models, serializability, transactions

Overwhelming, but often already known

Schedules S1 and S2 both produce the result x == 3. This result can be explained by assuming the scheduler executed the transactions in the order T1→T2→T3. This ordering matches S1 but not S2, which doesn't matter. On the other hand, S3 is an illegal schedule with a final result of x == 5. Schedule S4 is interesting: the final result is x == 3, but only by coincidence. Since the scheduler only sees read and write operations and not their effects, scheduling the operations x = x + 1 and then x = x + 2 is illegal.



```
BEGIN_TRANSACTION    BEGIN_TRANSACTION    BEGIN_TRANSACTION
  x = 0                x = 0                x = 0
  x = x + 1            x = x + 2            x = x + 3
END_TRANSACTION      END_TRANSACTION      END_TRANSACTION
```

Transaction T_1 Transaction T_2 Transaction T_3

A number of schedules

	Time --→						
S1	x = 0	x = x + 1	x = 0	x = x + 2	x = 0	x = x + 3	Legal
S2	x = 0	x = 0	x = x + 1	x = x + 2	x = 0	x = x + 3	Legal
S3	x = 0	x = 0	x = x + 1	x = 0	x = x + 2	x = x + 3	Illegal
S4	x = 0	x = 0	x = x + 3	x = 0	x = x + 1	x = x + 2	Illegal

Data-centric Consistency Models

- Consistent Ordering of Operations: Group Operations

Entry consistency: Basic Idea

- Many consistency models work at the level of basic read and write operations. These models were originally created for shared-memory multiprocessor systems and were implemented at the hardware level. In applications, concurrency between programs sharing data is usually managed with synchronization mechanisms like mutual exclusion and transactions. Within a program, the data handled by a series of read and write operations are protected from concurrent access.

- To achieve this, we consider shared synchronization variables, or simply, locks. A lock is linked to shared data items, with each shared item linked to one lock. When a process enters a critical region, it must acquire locks. When it leaves, it releases those locks. Each lock has a current owner - the process that last acquired it. If a process doesn't own a lock but wants to acquire it, it sends a message to the current owner asking for ownership and the current values of the data linked to that lock. While a process has exclusive access to a lock, it can read and write data. Several processes can have nonexclusive access to a lock simultaneously, allowing them to read but not write the data. Nonexclusive access is only granted if no other process has exclusive access.

Data-centric Consistency Models

- Consistent Ordering of Operations: Group Operations

Entry consistency: Criteria to be satisfied

- Acquiring a lock can only succeed if all updates to its associated shared data are complete.
- Exclusive access to a lock can only succeed if no other process has exclusive or nonexclusive access.
- Nonexclusive access to a lock is allowed only if any previous exclusive access is complete, including updates to the data associated with the lock.

Entry consistency: Definition

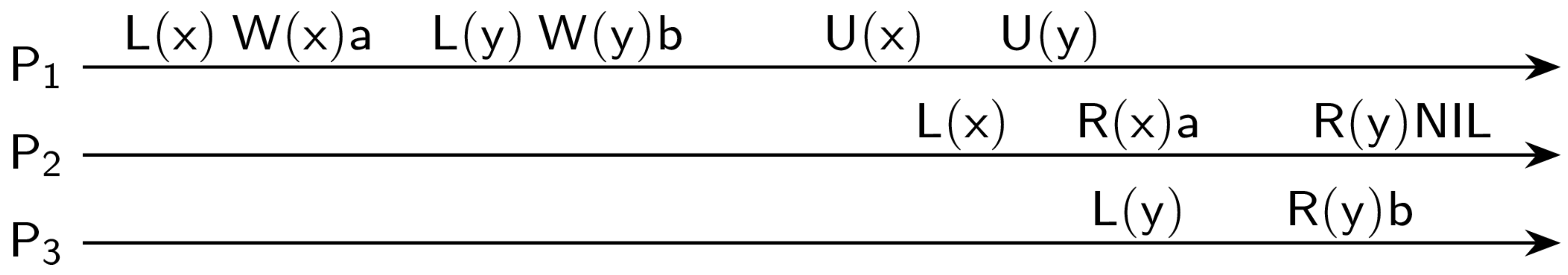
Criteria ensure that lock usage is linearized, maintaining sequential consistency, also, they are the definition of **entry consistency**.

Data-centric Consistency Models

- Consistent Ordering of Operations: Group Operations

A valid event sequence for entry consistency

Each data item has its own lock. We use L(x) to mean locking x and U(x) to mean unlocking it.



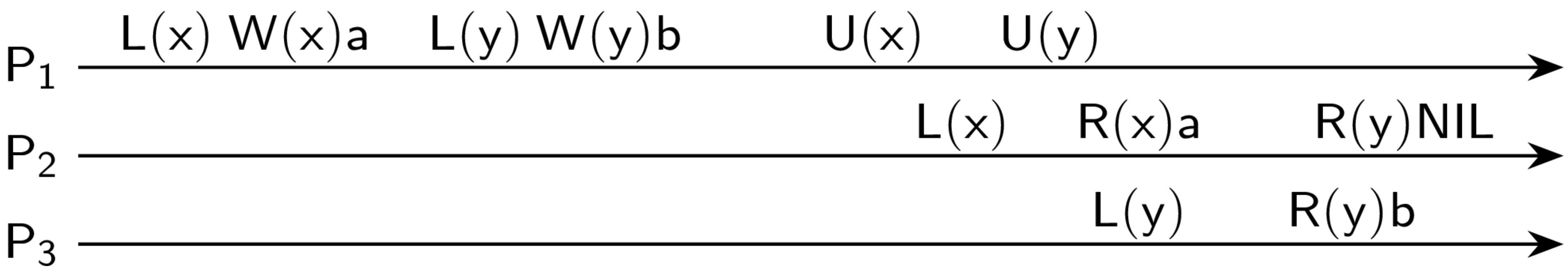
In this example, P1 locks x, changes x, then locks y. P2 locks x too, so it reads the updated value of x, but since it didn't lock y, it might read the initial value of y (NIL). On the other hand, P3 locks y first, so it reads the updated value of y after P1 unlocks it. Each process has its own copy of a variable, but these copies aren't updated instantly or automatically. When a process locks or unlocks a variable, it's telling the system that those copies need to be synchronized. So, a read operation without locking might result in reading outdated local values.

Data-centric Consistency Models

- Consistent Ordering of Operations: Group Operations

A valid event sequence for entry consistency

Each data item has its own lock. We use L(x) to mean locking x and U(x) to mean unlocking it.



Observation

Entry consistency implies that we need to lock and unlock data (implicitly or not).

Challenge

correctly associating data with locks. A straightforward way is to explicitly tell the middleware which data will be accessed, similar to declaring which database tables a transaction will affect. In an object-based approach, we could associate a unique lock with each declared object, effectively making all operations on that object sequential.

Data-centric Consistency Models

- Eventual consistency

  Main issue

  1. To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the the same order everywhere.

  2. In many database systems, most processes only read data, while only one or a few processes actually update it. The key question here is how quickly these updates should be made available to the processes that are only reading. In content delivery networks (CDNs), developers often choose to propagate updates slowly because they assume most clients are directed to the same replica and won't notice any inconsistencies.

  3. In the Web, web pages are updated by a single authority, like a webmaster or the page owner. There are typically no write-write conflicts to worry about. To improve efficiency, browsers and web proxies often cache pages locally and serve the cached version on subsequent requests. This can result in outdated pages being returned, but many users find this acceptable as long as they consistently access the same cache and don't notice the updates they're missing.

Data-centric Consistency Models

- Eventual consistency

Main issue

Conflicting operations: From the world of transactions

- Read–write conflict: a read operation and a write operation act concurrently, where one process wants to update a data item while another is reading it. Lazy propagation of updates is often acceptable, meaning readers might see the update only after some time. Sometimes, large-scale distributed and replicated databases that tolerate a fair amount of inconsistency. They share the feature that if no updates occur for a while, all replicas gradually become consistent – this is known as **eventual consistency**.
- Write–write conflict: two concurrent write operations

Eventual consistency means that as long as there are no write-write conflicts, all replicas will eventually have identical data. This model only requires that updates will eventually reach all replicas. Write-write conflicts are easier to resolve if only a few processes can make updates. Often, in case of conflicts, one write is declared the "winner," and its effects overwrite any other conflicting writes, making eventual consistency relatively cheap to implement.

Data-centric Consistency Models

- Eventual consistency

Main issue

Conflicting operations: From the world of transactions

- However, assuming that write-write conflicts rarely happen isn't realistic. Improvements can be made by grouping operations and using locks, which means processes need to coordinate their actions using mutual-exclusion mechanisms. For many large-scale systems, this coordination can be a real performance bottleneck.

Issue

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability.

Solution

Reducing coordination or weaken consistency requirements so that hopefully global synchronization can be avoided

Data-centric Consistency Models

- Eventual consistency

Definition

Consider a collection of data stores and (concurrent) write operations. The stores are eventually consistent when in lack of updates from a certain moment, all updates to that point are propagated in such a way that replicas will have the same data stored (until updates are accepted again).

Strong eventual consistency

Basic idea: It ensures that if there are conflicting updates, the replicas where those updates happened end up in the same state. If there are conflicting updates, have a globally determined resolution mechanism (for example, using Network Time Protocol (NTP), simply let the "most recent" update win).

Program consistency

Basic idea: It shifts focus from data-centric consistency to program consistency. Coordinating for consistency can be very costly, the question is whether and where coordination is actually necessary. Program consistency is about ensuring that a program produces the expected outcome despite various anomalies.

Data-centric Consistency Models

- Eventual consistency

Program consistency

P is a monotonic problem if for any input sets *S* and *T*, the solution for S is a subset of the solution for T. This means a program solving a monotonic problem can start with incomplete information and still make progress without having to roll back when missing information is later added. A classic example is a digital shopping cart: items can be added in any order by any server without conflicts. However, if items can also be removed, we have a problem. Monotonicity can help if we treat add and delete operations as separate sets that grow independently. We only need to coordinate when these sets reach their final states. The difference between what has been added and what has been deleted determines the final consistent state.

Observation: A program solving a monotonic problem can start with incomplete information, but is guaranteed not to have to roll back when missing information becomes available.

Important observation

In all cases, we are avoiding global synchronization.

Data-centric Consistency Models

- Continuous consistency

We can actually talk about a degree of consistency

- replicas may differ in their numerical value. This is useful for data with numerical meanings, like stock market prices. For example, an application might specify that two copies of stock prices shouldn't differ by more than $0.02 (an absolute numerical deviation) or by more than 0.5% (a relative numerical deviation). So, if a stock price goes up and one replica updates immediately within these limits, the replicas are still considered consistent. It can also refer to the number of updates applied to one replica but not yet seen by others. For instance, a web cache might not have the latest batch of operations from a web server.
- replicas may differ in their relative staleness. It can refer to how old the data in a replica can be. Some applications can handle old data as long as it's not too old. For example, weather reports remain accurate for a few hours. A main server might get timely updates but only send them to replicas occasionally.

Data-centric Consistency Models

- Continuous consistency

We can actually talk about a degree of consistency

- there may be differences regarding (number and order) of performed update operations. Some applications can tolerate different orderings of updates across replicas, as long as these differences are within certain bounds. Think of these updates as being applied tentatively to a local copy while waiting for global agreement from all replicas. Some updates might need to be rolled back and applied in a different order before becoming permanent. Ordering deviations are trickier to understand than numerical or staleness deviations.

Example: Conit

Consistency unit ⇒ specifies the data unit over which consistency is to be measured.

Data-centric Consistency Models

- Continuous consistency

Example: Consistency for mobile users

Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.

- At location *A* you access the database doing reads and updates.
- At location *B* you continue your work, but unless you access the same server as the one at location *A*, you may detect inconsistencies:
  - your updates at *A* may not have yet been propagated to *B*
  - you may be reading newer entries than the ones available at *A*
  - your updates at *B* may eventually conflict with those at *A*

Note

The only thing you really want is that the entries you updated and/or read at *A*, are in *B* the way you left them in *A*. In that case, the database will appear to be consistent to you.

Client-centric Consistency Models

Issue

- Data-centric consistency models ensure that all users see a consistent view of data. These models assume multiple processes might be updating the data store at the same time, so consistency is crucial. For example, with object-based entry consistency, when a process accesses an object, it gets the latest version with all updates, and no other process can interfere during that access. Handling concurrent operations on shared data while maintaining strong consistency is crucial for distributed systems. But for performance reasons, strong consistency is sometimes only guaranteed when using things like transactions or synchronization variables. Sometimes, though, strong consistency isn't possible, so we settle for weaker forms like causal consistency or eventual consistency.

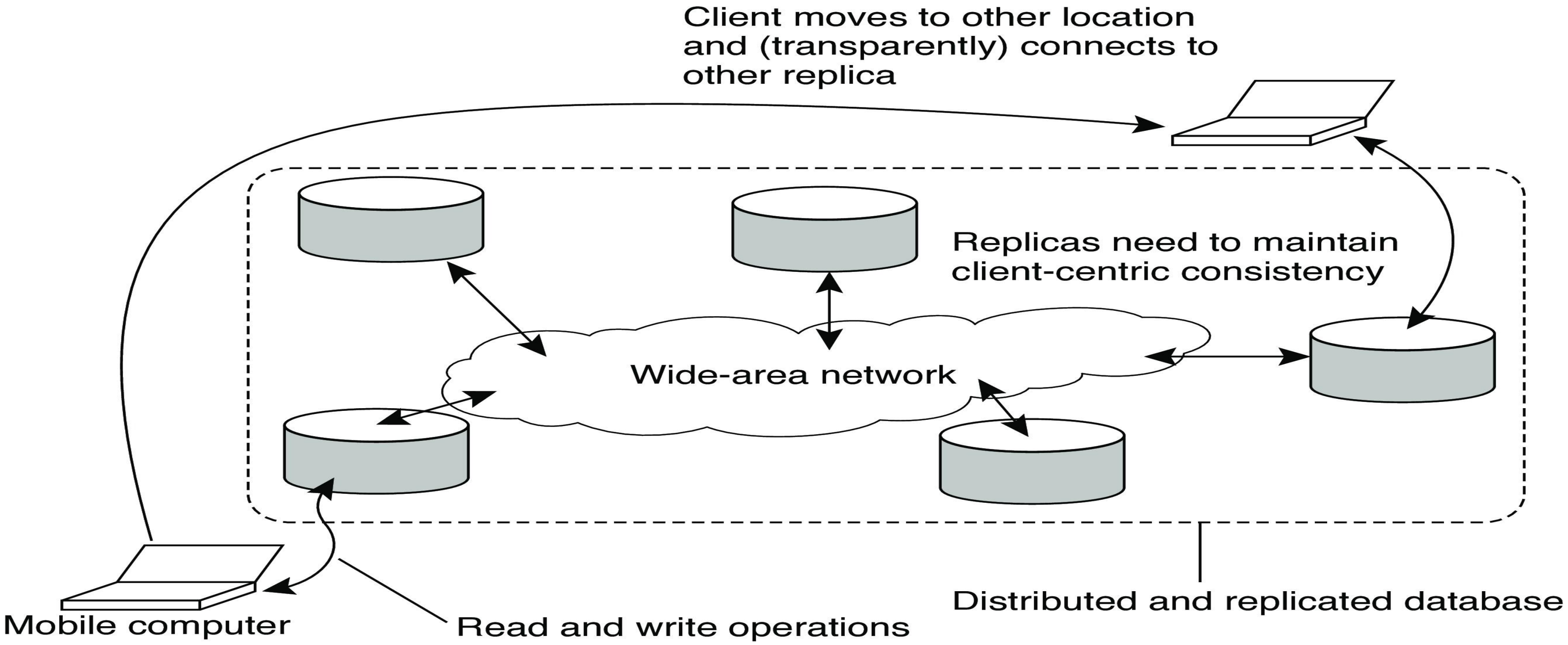
Client-centric Consistency Models

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

It guarantees a single client's interactions with the data store, but not for concurrent accesses by different clients.

It was developed for mobile computing with unreliable network connectivity

A process accesses the data store, it usually connects to the nearest copy, although any copy works. All operations are performed on this local copy, and updates eventually spread to the other copies.



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
Mobile computer
Read and write operations

Client-centric Consistency Models

- Monotonic Reads

A data store provides monotonic-read consistency if it meets this condition: If a process reads a value of a data item x, any later read of x by that process will always return the same value or a more recent one. In other words, once a process has seen a certain value of x, it will never see an older version of x.

Example

Automatically reading your personal calendar updates from different servers. Monotonic reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

Example

Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

Imagine your email is stored on multiple machines. You can add new emails to your mailbox from any location, but updates spread slowly, only when needed. Suppose you read your email in San Francisco. Let's assume reading emails doesn't change the mailbox (like deleting, moving, or tagging emails). When you later fly to New York and check your email again, monotonic-read consistency ensures that all the emails you saw in San Francisco will still be in your mailbox when you open it in New York.

Client-centric Consistency Models

- Monotonic Reads

Notations

- $W_1(x_2)$ is the write operation by process P_1 that leads to version x_2 of x
- $W_1(x_i;x_j)$ indicates P_1 produces version x_j based on a previous version x_i.
- $W_1(x_i|x_j)$ indicates P_1 produces version x_j concurrently to version x_i.
- *R1(x2)* just means that process P1 read version x2.

Client-centric Consistency Models

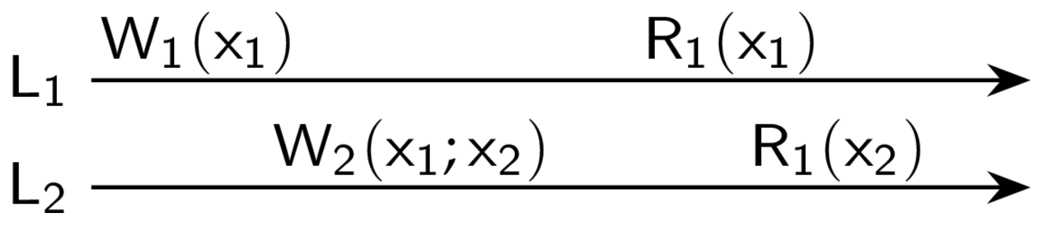
- Monotonic Reads

Definition

If a process reads the value of a data item *x*, any successive read operation on *x* by that process will always return that same or a more recent value.

Example

Process P1 first writes to x at location L1, creating version x1. Then, P1 reads this version. Meanwhile, at location L2, process P2 creates version x2, which is based on x1. When P1 moves to L2 and reads x again, it finds the more recent version x2, which at least considers its previous write.



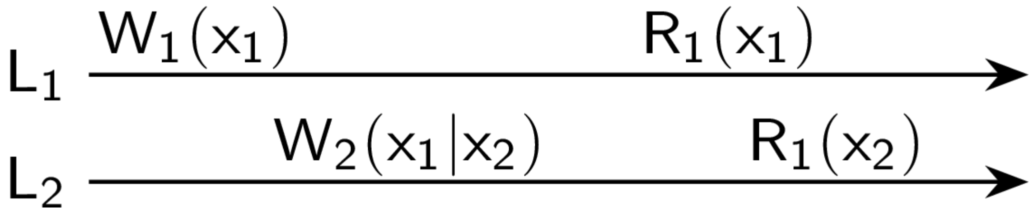
A monotonic-read consistent data store

Client-centric Consistency Models

- Monotonic Reads

Example

Monotonic-read consistency is violated. After P1 reads x1 at L1, it later reads x2 at L2. But the write operation W2(x1|x2) by P2 at L2 produces a version that doesn't follow from x1. So, P1’s read at L2 doesn’t include the changes from when it reads x1 at L1.



A data store that does not provide monotonic reads

Client-centric Consistency Models

- Monotonic Writes

In a system with monotonic-write consistency, if a process writes to a data item x, any following write by the same process has to wait until the first one is completely done. So, if process Pk writes Wk(xi) and then writes Wk(xj), no matter where the second write happens, it must follow the first one. This ensures that the new write takes into account the previous write. In other words, you can only write to a copy of x if it’s up-to-date with all the previous writes by the same process. If it’s not up-to-date, the new write has to wait. Monotonic-write consistency is similar to first input first output (FIFO) consistency, which makes sure that writes by the same process are done in the right order everywhere. The difference is that monotonic writes focus on consistency for just one process rather than for all processes. Therefore, when updating the program on the server, we need to ensure that all components that compilation and linking depend on are also on the same server.

Client-centric Consistency Models

- Monotonic Writes

This ordering isn't always necessary if each write completely replaces the current value of x. But usually, writes only change part of the data item. Take a software library, for example. Updating the library might involve replacing a few functions, creating a new version. With monotonic-write consistency, before applying the new update, the system makes sure all previous updates are applied first. In This way, the library ends up being the most recent version, including all previous updates. In some examples, one process writes to a data item at one location, and then again at another location. If the first write hasn't been shared with the second location, we lose consistency. On the other hand, if all updates are shared correctly, everything stays consistent. In more complex cases, if different processes update at different times and places without sharing updates, it can lead to conflicts. These conflicts must be resolved, often favoring one version over another to maintain consistency. However, future writes could still break this consistency if not managed properly. .

Client-centric Consistency Models

- Monotonic Writes

Example

Updating a program at server, and ensuring that all components on which compilation and linking depends, are also placed at the same server.

Example

Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

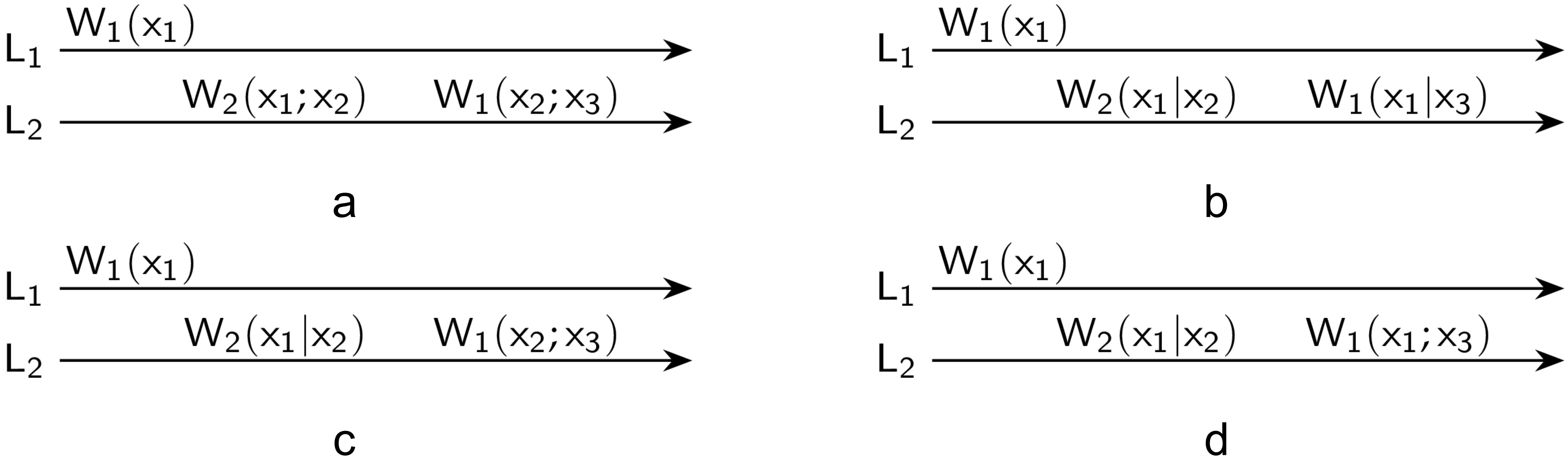
Client-centric Consistency Models

- Monotonic Writes

Definition

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

(a) Process P1 writes to x at location L1, shown as W1(x1). Later, P1 writes to x again, but this time at L2, shown as W1(x2;x3). The version x2 created by P1 at L2 follows an update from process P2, which was based on version x1 (W2(x1;x2)). To keep things consistent with monotonic writes, the first write at L1 must be sent to L2 and possibly updated before P1 can write x2



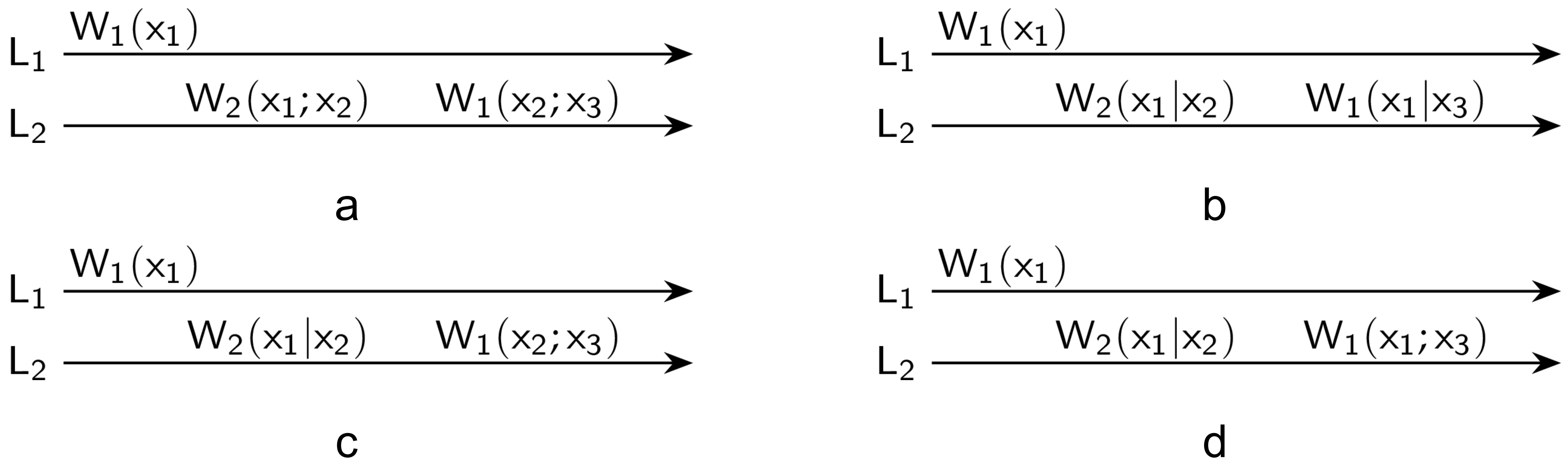
Client-centric Consistency Models

- Monotonic Writes

Definition

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

(b) We see what happens when monotonic-write consistency isn't maintained. Here, x1 wasn't sent to L2 before P2 created a new version, shown as W2(x1|x2). P2 creates a version concurrent to x1, and then P1 produces x3 concurrently to x1 too. This breaks monotonic-write consistency because the writes are out of order.



Client-centric Consistency Models

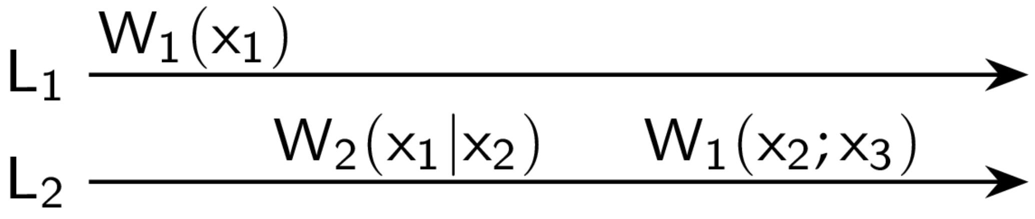
- Monotonic Writes

Definition

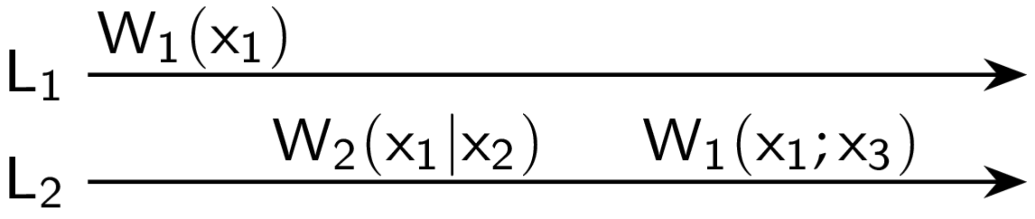
A write operation by a process on a data item *x* is completed before any successive write operation on *x* by the same process.

(c) It shows another violation of monotonic-write consistency. P1 creates x3 based on x2, but x2 didn't include the changes from x1. So, we end up with W1(x1|x3), meaning the writes are not in order.

(d) It presents an interesting case. Here, P2 creates x2 concurrently with x1. Later, P1 produces x3, but it appears based on the fact that x1 was somehow available at L2. There’s a write-write conflict with x2 that gets resolved in favor of x1. This setup follows the rules of monotonic-write consistency. However, any new write by P2 at L2 without knowing about x1 will violate the consistency again.



c



d

Client-centric Consistency Models

- Read Your Writes

  Definition

  Whenever a process writes to a data item x, any later read by the same process will see the changes made by that write, no matter where the read happens. In other words, after you write something, you should be able to read it right away, and it will show your changes. You've probably noticed a lack of this consistency when updating web pages. For instance, you might update a document with an editor or word processor, save the new version, and then check the webpage to see your changes. However, sometimes you don’t see the update because the browser or server caches an old version of the page. The same thing happens when you change a password. If you change your password to access a digital library, it might take a few minutes for the change to take effect. This delay happens because the new password needs to be propagated across all servers managing the library. It ensures that the cache is updated so you see your changes immediately

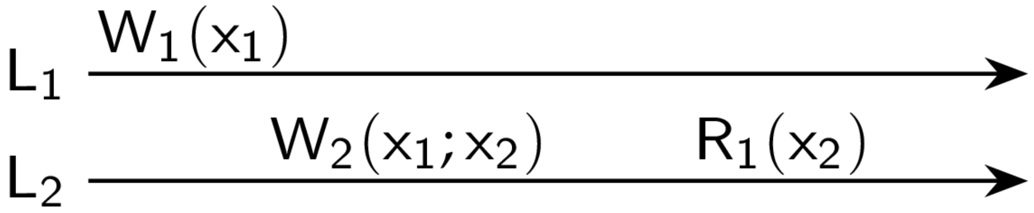
  The effect of a write operation by a process on a data item *x*, will always be seen by a successive read operation on *x* by the same process.

Client-centric Consistency Models

- Read Your Writes

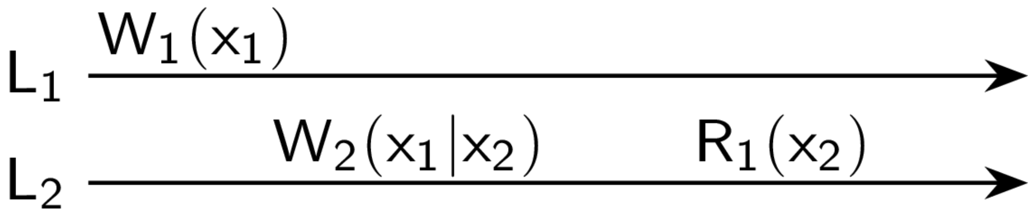
Definition

Process P1 writes to x (W1(x1)) and then reads x at a different location. Read-your-writes consistency makes sure the read sees the write's effects. This is shown by W2(x1;x2), meaning process P2 created a new version of x based on x1.



OK

Process P2 creates a version of x at the same time as x1 (W2(x1|x2)), meaning P1's write at L1 hasn't reached L2 yet. So, when P1 reads x2, it won't see the changes made by its own write at L1.



Not OK

Example

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

Client-centric Consistency Models

- Example: client-centric consistency in ZooKeeper

Yet another model?

ZooKeeper’s consistency model mixes elements of data-centric and client-centric models.

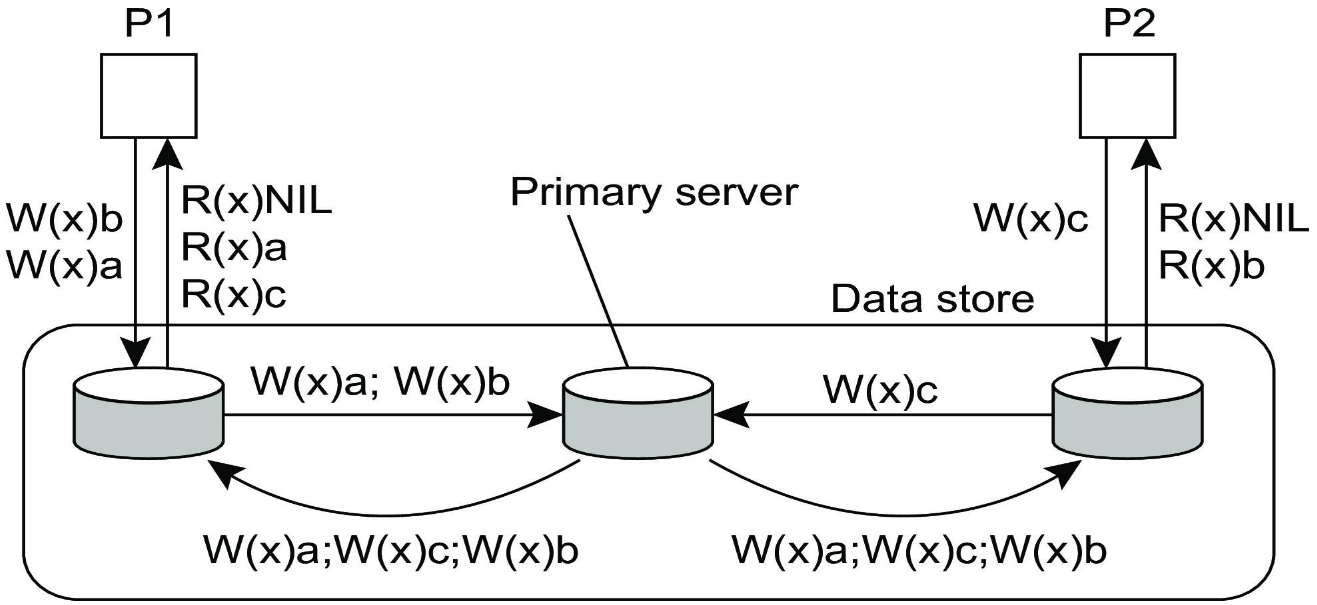
ZooKeeper guarantees that update operations are serializable and keep their order. This means that the state of ZooKeeper can always be explained by some linear order of all the update operations, while maintaining the order of writes from each client. In other words, even though multiple clients can update ZooKeeper at the same time, the final state can be understood by some sequence of those updates where each client’s order is preserved. ZooKeeper also guarantees monotonic reads, but it doesn’t guarantee read-your-writes or writes-follow-reads consistency. Think of ZooKeeper as a group of servers, with each client connected to its own server. There’s one main server (the primary server) that handles all write operations and processes them in the order they’re received from the clients. Read operations are handled by the server the client is connected to, in the order they were submitted by the client. However, ZooKeeper doesn’t specify when the primary server updates the other servers. This means clients don’t know when they’ll see their own updates or updates from other clients.

Client-centric Consistency Models

- Example: client-centric consistency in ZooKeeper

Take a naive example

If process P1 submits two writes, W(x)a and then W(x)b, and process P2 submits W(x)c, the primary server receives these writes and decides to forward them in the order W(x)a, W(x)c, and W(x)b to the other servers. The timing of when these writes are submitted or performed isn't specified, but the order must be followed. If P1 submits read requests R(x), it might first read the initial value NIL, then R(x)a, and later R(x)c (eventually, it will read only R(x)b). Similarly, P2 might read NIL initially and later read the final value R(x)b. Both P1 and P2 might read NIL even after submitting their write operations. This shows that ZooKeeper doesn't provide read-your-writes consistency. Also, if P1 first read NIL and then submitted W(x)a and W(x)b, it might still read R(x)c if its latest write (R(x)b) hasn't been processed yet. This demonstrates that ZooKeeper doesn't provide writes-follow-reads consistency either.



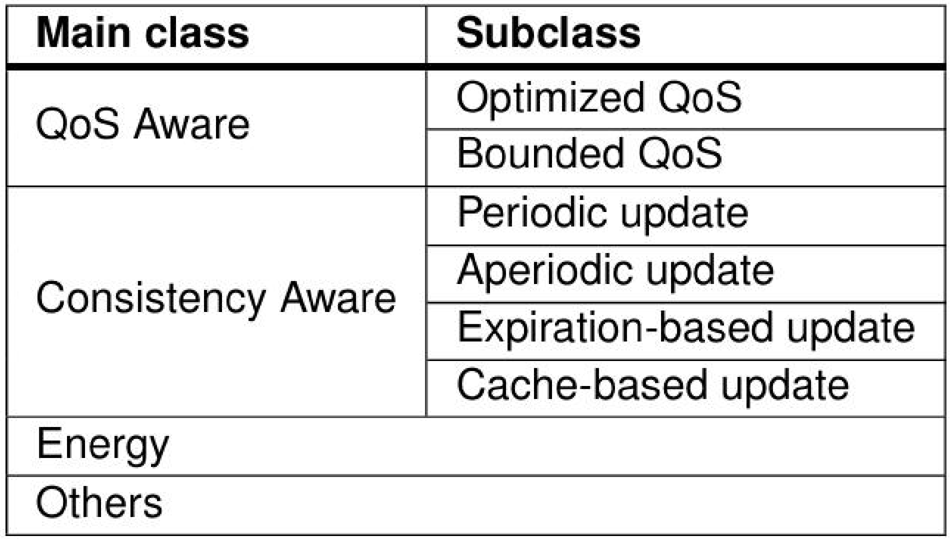
Replica Management

- Finding the best server location

  Essence

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

  - Select best location out of $N - K$ for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location minimizes the average distance to all clients.) Computationally expensive.
  - Select the K-th largest autonomous system and place a server at the best-connected host. Computationally expensive.
  - Position nodes in a d-dimensional geometric space, where distance reflects latency. Identify the K regions with highest density and place a server in every one. Computationally cheap.



Main class	Subclass
QoS Aware	Optimized QoS
	Bounded QoS
Consistency Aware	Periodic update
	Aperiodic update
	Expiration-based update
	Cache-based update
Energy	
Others	

Replica Management

- Finding the best server location

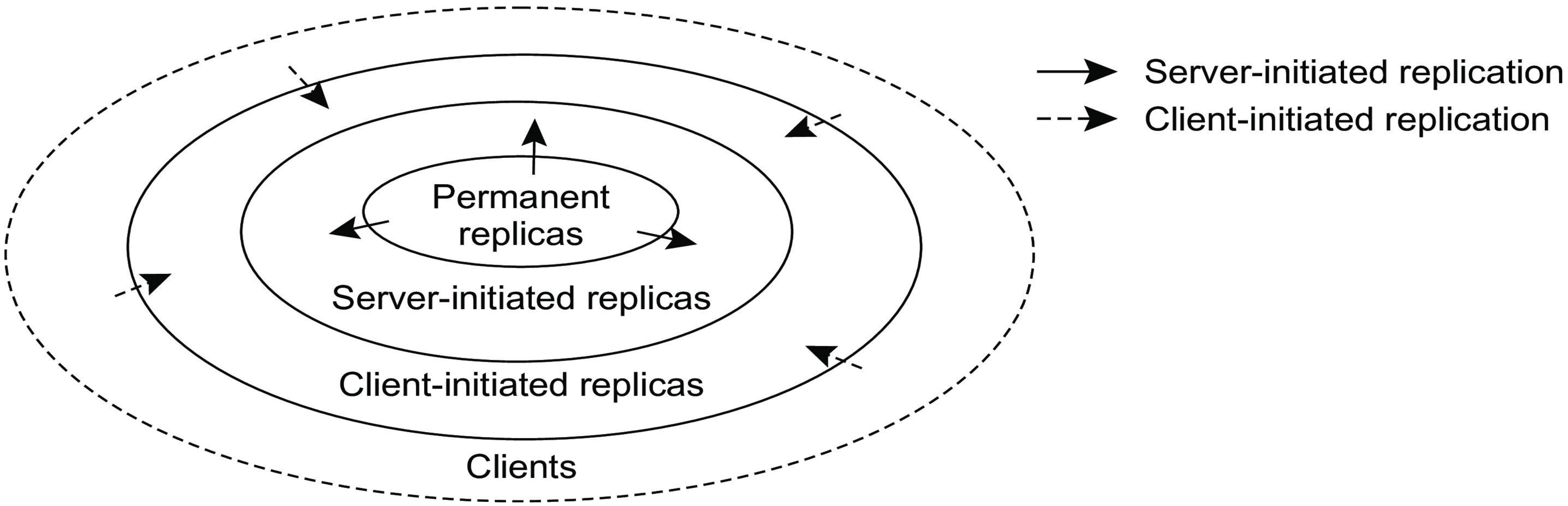
Essence

- **Quality of Service (QoS):** Server placement is optimized for QoS parameters, like guaranteed bandwidth. However, solving for QoS is difficult, often requiring heuristics and approximations since accurately measuring parameters like latency and bandwidth is challenging.
- **Consistency-Aware Algorithms:** These focus on the costs of keeping replicas up to date, considering when and how updates are propagated. These models often rely on knowing read and update patterns, which can be difficult to predict.
- **Energy-Aware Algorithms:** These consider energy consumption, aiming to place servers where they can be energy efficient. Factors include switching power modes and balancing the workload among servers to save energy. Again, this requires understanding access patterns and network capabilities.
- **Other Models:** These include considerations for Content Delivery Networks (CDNs) spread across multiple organizations. Decisions here involve monetary costs and connectivity, taking into account QoS parameters and the financial aspects of working with different organizations.

Replica Management

- Content replication and placement

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


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

Distinguish different processes

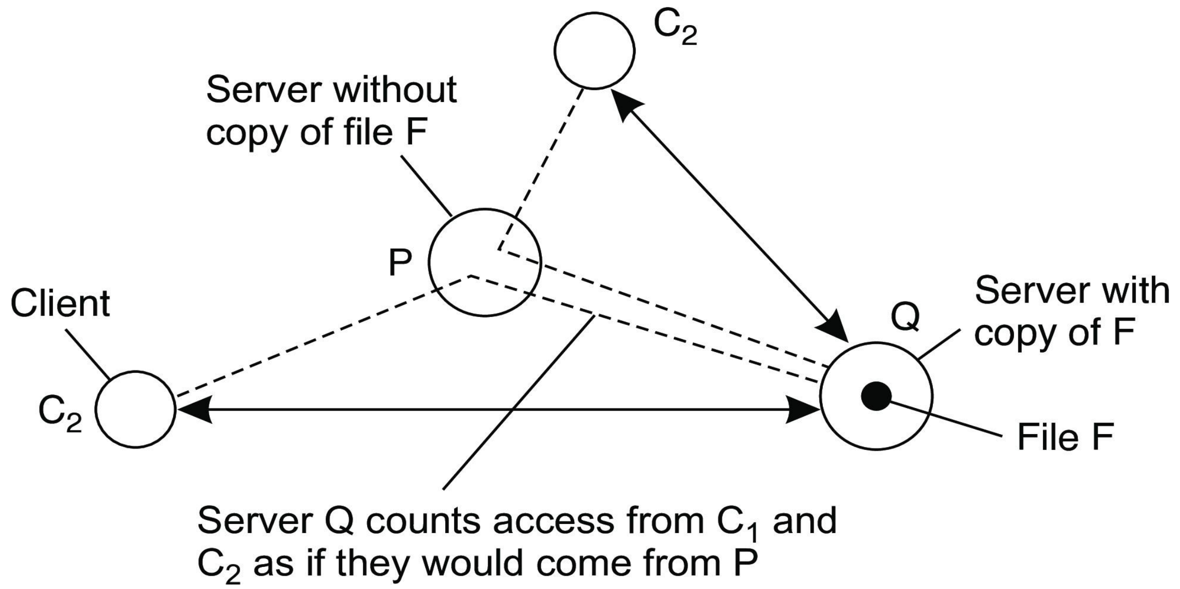
A process is capable of hosting a replica of an object or data:

- Permanent replicas: Process/machine always having a replica
- Server-initiated replica: Process that can dynamically host a replica on request of another server in the data store
- Client-initiated replica: Process that can dynamically host a replica on request of a client (client cache)

Replica Management

- Content replication and placement: Dynamic replication
  - **Reducing server load:** By replicating files to lighten the load on busy servers.
  - **Proximity to clients:** By moving or replicating files to servers near clients that request those files often.

Counting access requests from different clients

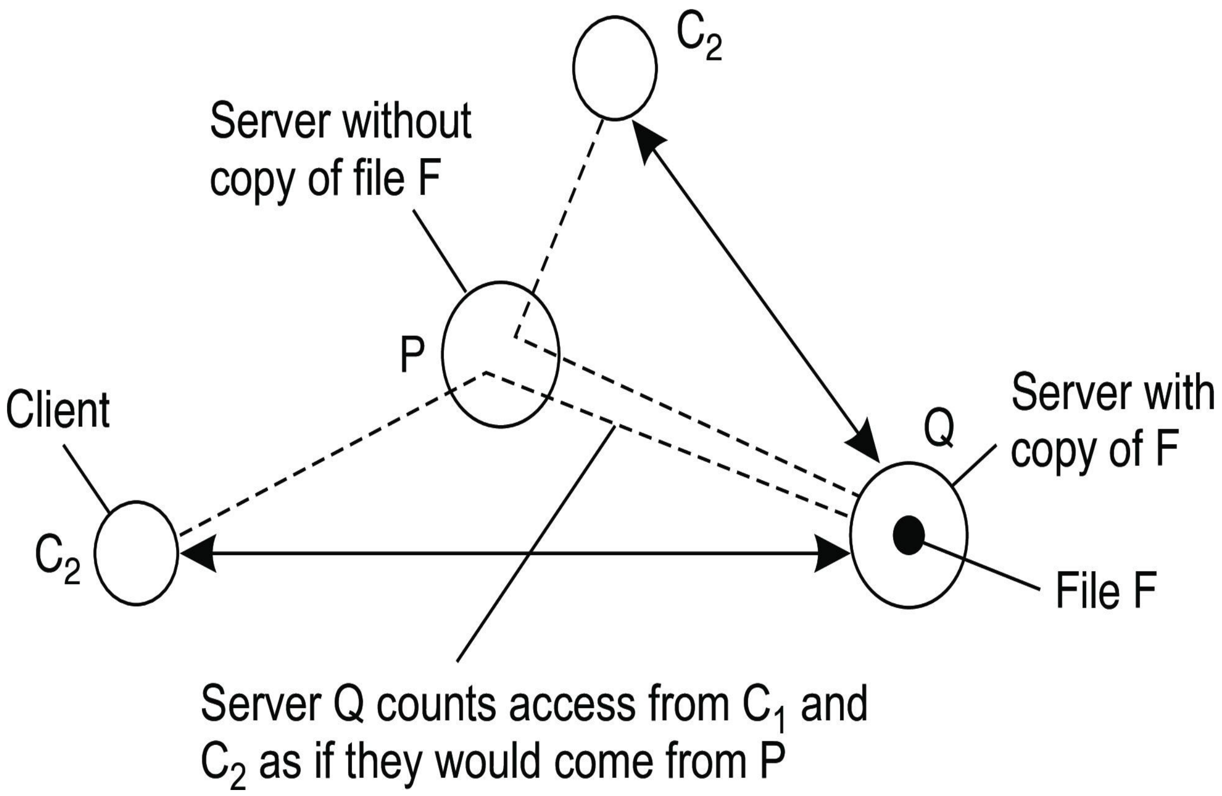

C2
Server without copy of file F
P
Client
C2
Q
Server with copy of F
File F
Server Q counts access from C1 and C2 as if they would come from P

- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold *D* ⇒ drop file
- Number of accesses exceeds threshold *R* ⇒ replicate file
- Number of access between *D* and *R* ⇒ migrate file

Replica Management

- Content replication and placement

  Counting access requests from different clients

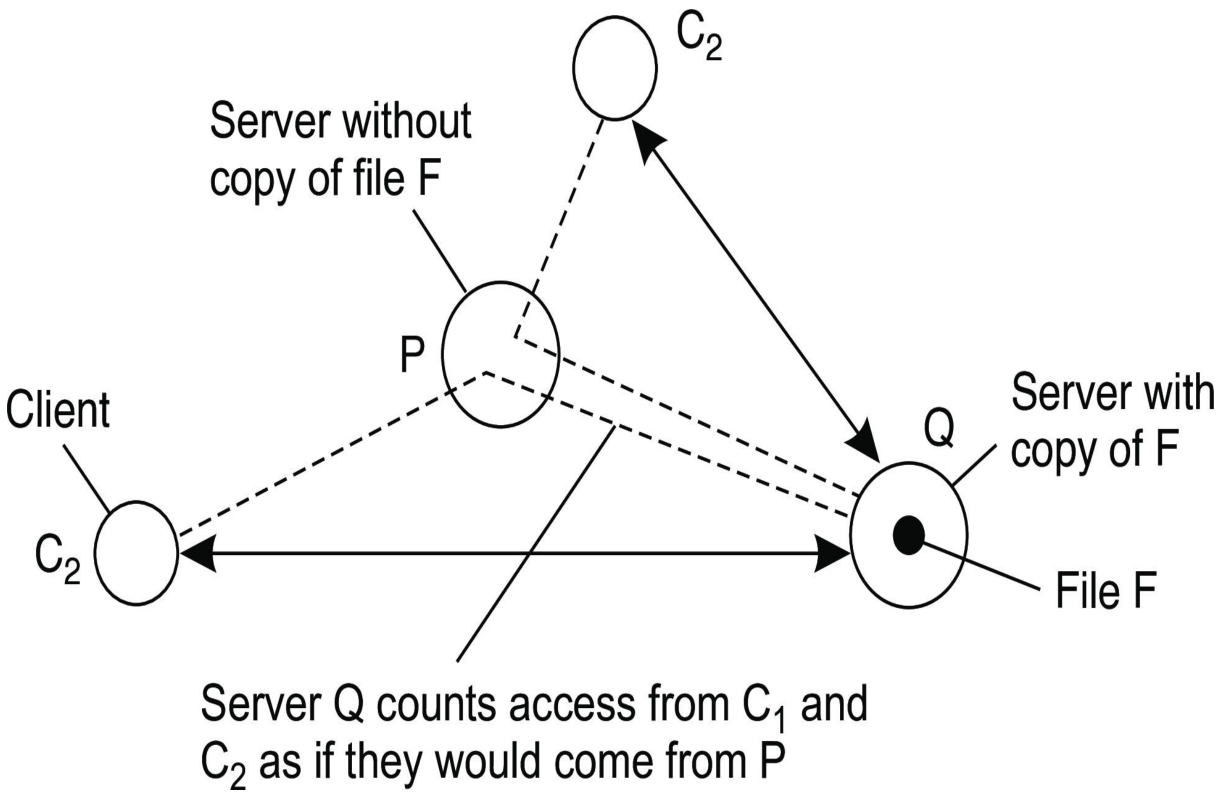

C$_2$
Server without copy of file F
P
Client
C$_2$
Q
Server with copy of F
File F
Server Q counts access from C$_1$ and C$_2$ as if they would come from P

- Each server keeps track of how often each file is accessed and where those requests come from. For example, if clients C1 and C2 both use the same closest server P to request file F from server Q, these requests are counted together at Q. When requests for a file F on server S reach a certain deletion threshold (del(S,F)), the file can be removed from S, reducing the number of replicas and possibly increasing the workload on other servers.

Replica Management

- Content replication and placement

  Counting access requests from different clients



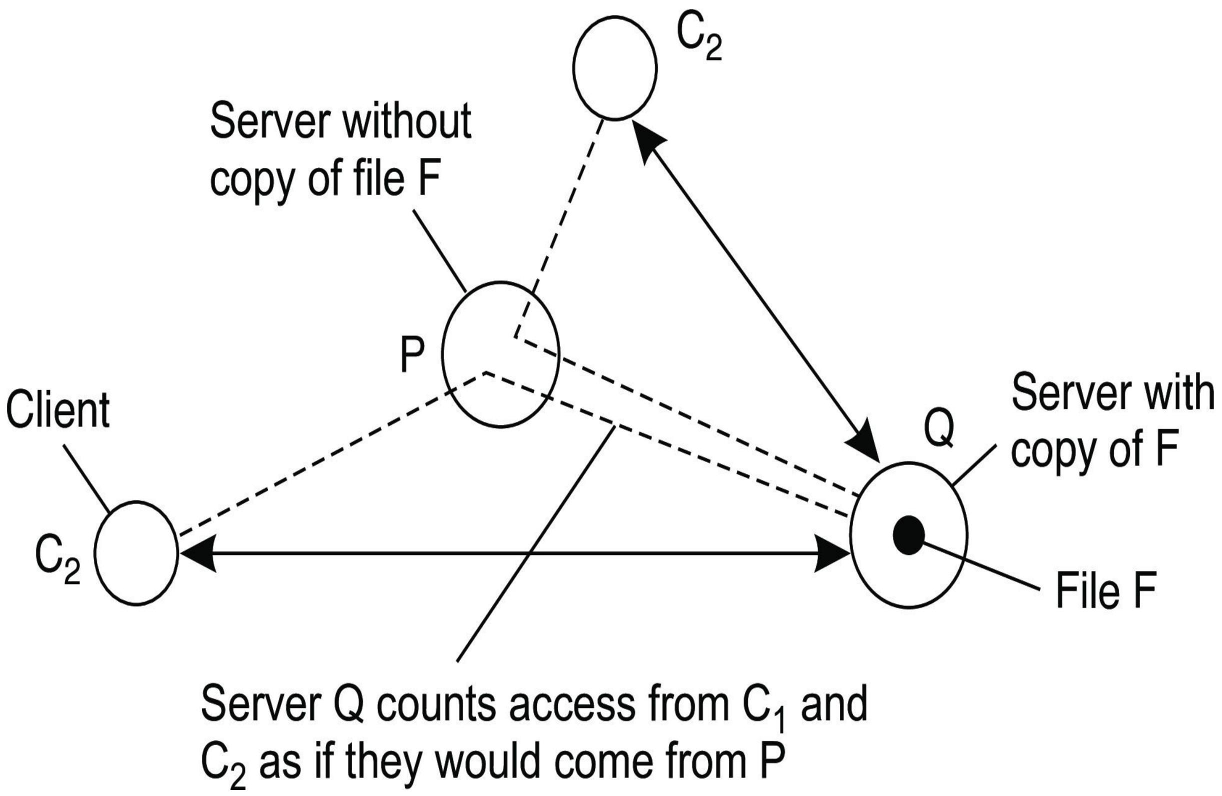
C_2
Server without copy of file F
P
Client
C_2
Q
Server with copy of F
File F
Server Q counts access from C_1 and C_2 as if they would come from P

- However, there’s always at least one copy of each file kept somewhere. A replication threshold (rep(S,F)), which is higher than the deletion threshold, indicates that a file is getting so many requests that it should be replicated on another server. If requests fall between the deletion and replication thresholds, the file can be moved (migrated) but not removed. When server Q reevaluates its files, it checks access counts. If requests for file F reach below del(Q,F), it deletes the file unless it’s the last copy.

Replica Management

- Content replication and placement

Counting access requests from different clients


C_2
Server without copy of file F
P
Client
C_2
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

- If more than half the requests for F at Q come from server P, Q tries to migrate F to P. Sometimes, migration fails because server P might be too busy or out of space. If that happens, Q tries to replicate F on other servers, but only if requests for F exceed rep(Q,F). Q starts with the farthest server and checks if requests from server R for F are significant, then tries to replicate F to R.

Replica Management

- Content Distribution

Consider only a client-server combination

- Propagate only notification/invalidation of update (often used for caches)
- Transfer data from one copy to another (distributed databases): passive replication
- Propagate the update operation to other copies: active replication

Note

No single approach is the best, but depends highly on available bandwidth and read-to-write ratio at replicas.

Replica Management

- Content Distribution

Consider only a client-server combination

- **Propagate a Notification:** This is what invalidation protocols do. They inform other copies that an update has happened and that their data is now outdated. The notification might specify which part of the data is updated, so only that part is invalidated. The big advantage here is that it uses very little network bandwidth since you're just sending a notification. This works best when there are lots of updates compared to reads, meaning the read-to-write ratio is low. For example, if you have a data store where updates occur frequently and the updated data is large, sending the actual data each time could be wasteful. Imagine two updates happening in quick succession without any reads in between; sending the first update is meaningless because it's soon overwritten by the second. A simple notification that data has changed would be more efficient.

Replica Management

- Content Distribution

Consider only a client-server combination

- **Transfer Modified Data:** This method is useful when the read-to-write ratio is high. If updates are likely to be read before the next update occurs, sending the modified data makes sense. Sometimes, instead of sending the full data, you can log the changes and send those logs to save bandwidth. Often, multiple modifications are packed into a single message to reduce communication overhead.

- **Propagate the Update Operation:** Instead of transferring data, you tell each replica which update operation to perform, sending only the necessary parameters. This is called active replication. Here, each replica actively updates its data by executing the operations. The advantage is minimal bandwidth usage, as the operation parameters are usually small. Plus, you can perform complex operations that help keep replicas consistent. The downside is that this requires more processing power at each replica, especially if the operations are complex.

Replica Management

- Content Distribution

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

- Pushing updates: server-initiated approach, in which update is propagated regardless whether target asked for it.
- Pulling updates: client-initiated approach, in which client requests to be updated.

Issue	Push-based	Pull-based
1:	List of client caches	None
2:	Update (and possibly fetch update)	Poll and update
3:	Immediate (or fetch-update time)	Fetch-update time
1: State at server *2: Messages to be exchanged* *3: Response time at the client*		

Replica Management

- Content Distribution

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

- Pushing updates: In a push-based approach, also known as server-based protocols, updates are sent to replicas without those replicas even asking for the updates. This method is often used between permanent replicas and server-initiated replicas, but it can also push updates to client caches. Push-based protocols are great when you need strong consistency. The need for strong consistency is because these replicas are often shared by many clients who mostly perform read operations. So, the read-to-update ratio at each replica is pretty high. Push-based protocols are efficient here because each update is likely to be useful for at least one readers. In addition, they make consistent data immediately available when requested.

Replica Management

- Content Distribution

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

- Pulling updates: In a pull-based approach, a server or client requests updates from another server when needed. Also called client-based protocols, these are often used by client caches. For example, web caches might first check if their cached data is still up-to-date by asking the original web server. If the data has changed, the modified data is sent to the cache and then to the client. If not, the cached data is used. Essentially, the client checks with the server to see if an update is needed. Content delivery networks often use this method, where replica servers cache content but verify updates with the origin server before delivering it. Pull-based approaches are efficient when the read-to-update ratio is low, which is often the case with client caches that serve only one client. Even when a cache is shared by many clients, a pull approach can be efficient if the cached data isn't frequently shared. The disadvantage is that response time increases when a cache miss happens.

Replica Management

- Content Distribution

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

- Comparing Push and Pull: There are several trade-offs between push-based and pull-based solutions, as shown in table. For simplicity, let’s consider a client-server system with a single server and multiple client caches. In push-based protocols, the server needs to keep track of all client caches, whether they are end-user clients or replica servers. This can create a significant overhead for the server. For example, a web server might need to track tens of thousands of client caches. Each time a page is updated, the server has to notify all those caches. If a client deletes a page due to lack of space, it has to inform the server, adding more communication. In terms of messaging, in a push-based approach, the server sends updates to each client. If updates are just invalidations, the client has to fetch the modified data. In a pull-based approach, the client polls the server and fetches the modified data if necessary. Finally, the response time for the client differs. With push-based protocols, if the server pushes modified data, the client's response time is zero. If the server pushes invalidations, the response time is the same as the pull-based approach, determined by how long it takes to fetch the updated data from the server.

Replica Management

- Content Distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time adaptive

- Age-based leases: An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- Renewal-frequency based leases: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- State-based leases: The more loaded a server is, the shorter the expiration times become

Question

Why are we doing all this?

leases provide a smart way to manage updates by balancing the push and pull methods based on specific criteria. This helps in maintaining efficiency and consistency across the system.

Replica Management

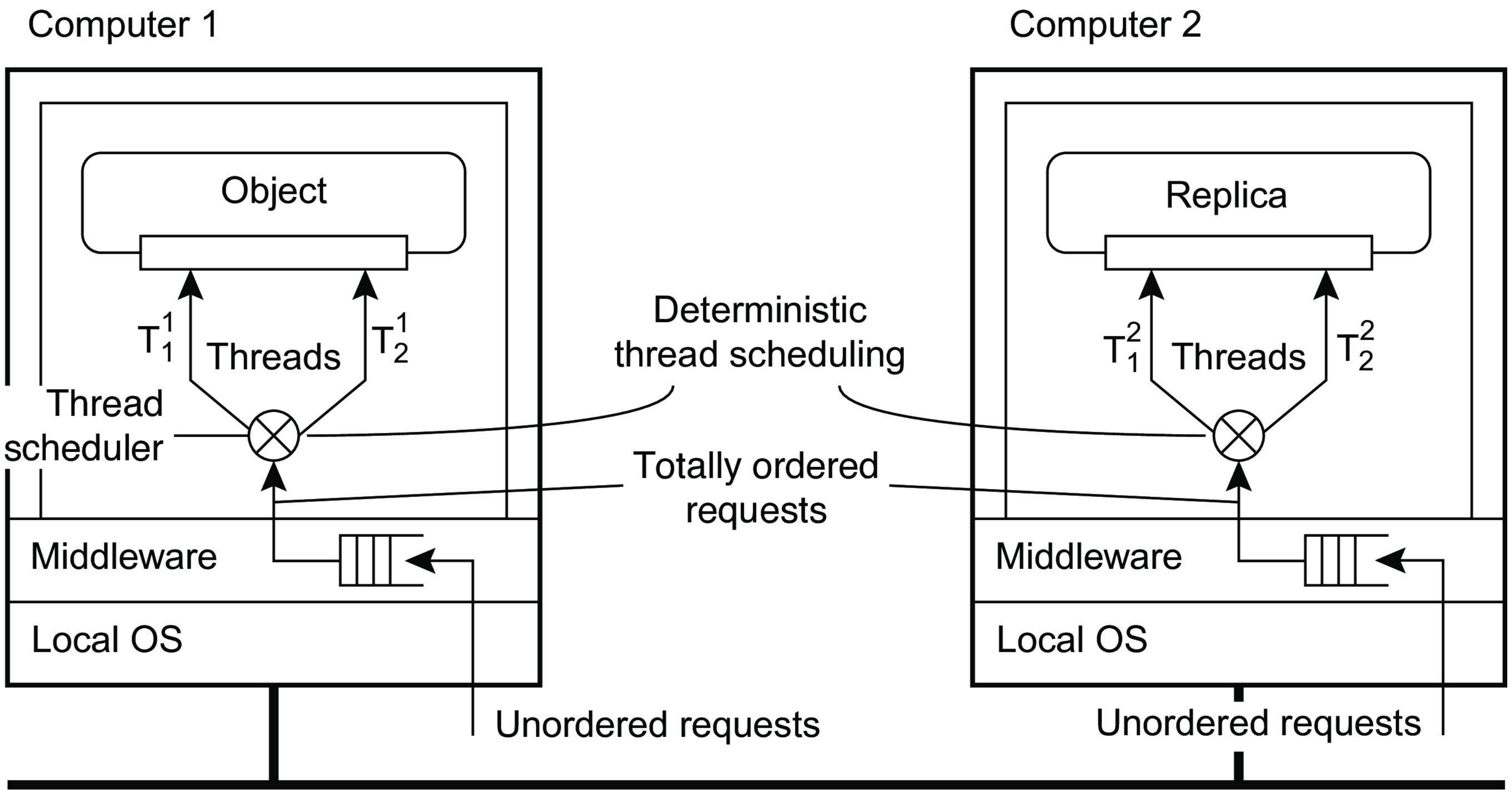
- Managing replicated objects
  - In distributed systems, data consistency often relies on entry consistency, which groups operations on shared data using locks. This means that when an object is locked during a call, access is serialized, keeping everything consistent. However, simply associating a lock with an object can be tricky when that object is replicated.

Issues:

- Preventing Concurrent Execution: We need to ensure that when a method on an object is running, no other methods can execute simultaneously. This is handled by local locking, which serializes access to the object’s data.
- Consistency Across Replicas: When an object is replicated, all replicas must reflect the same changes. This means we need to ensure that all method calls on replicas happen in the same order everywhere.

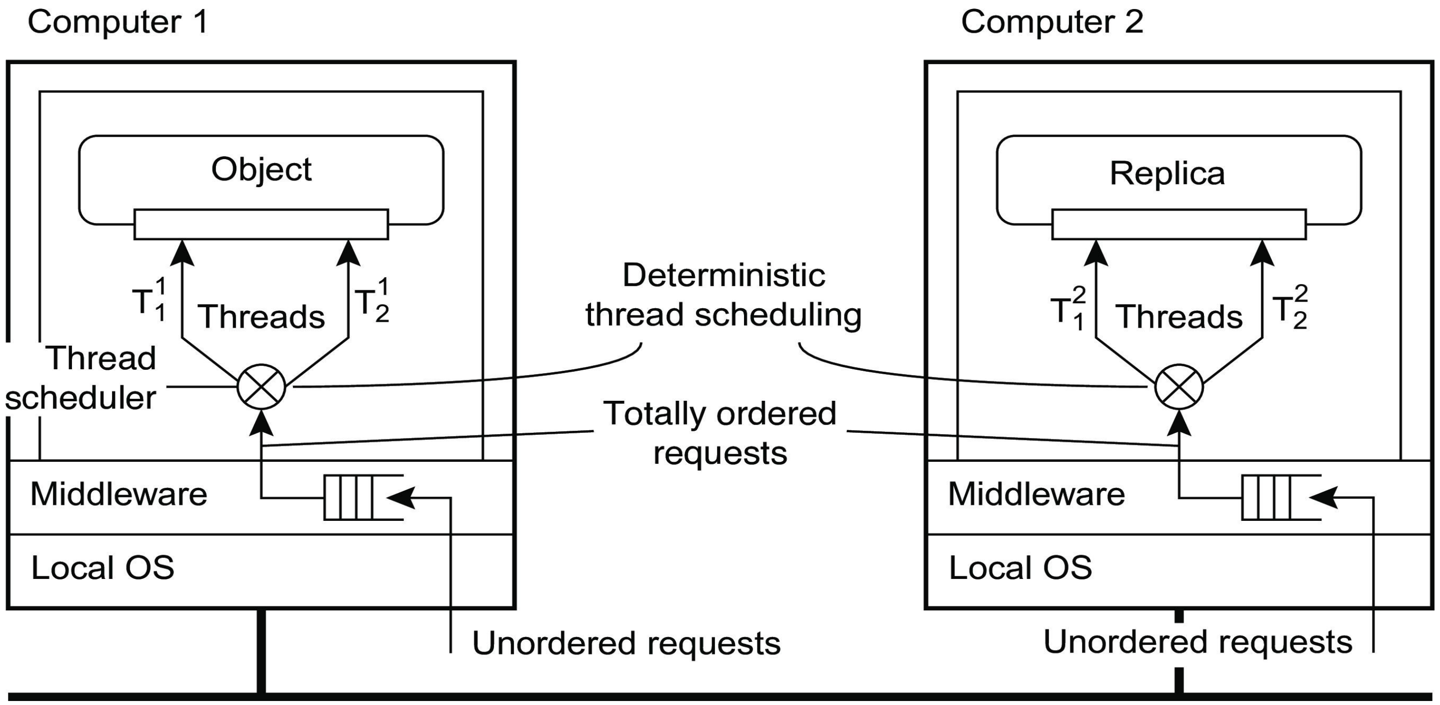
Replica Management

- Managing replicated objects
  - Designing replicated objects starts with a single object, protecting it against concurrent access with local locking. Then, we replicate it. Middleware ensures that if a client invokes a replicated object, the invocation is passed to all replicas and handled in the same order everywhere. We also need to make sure that all threads in those servers process the requests in the correct order.



Replica Management

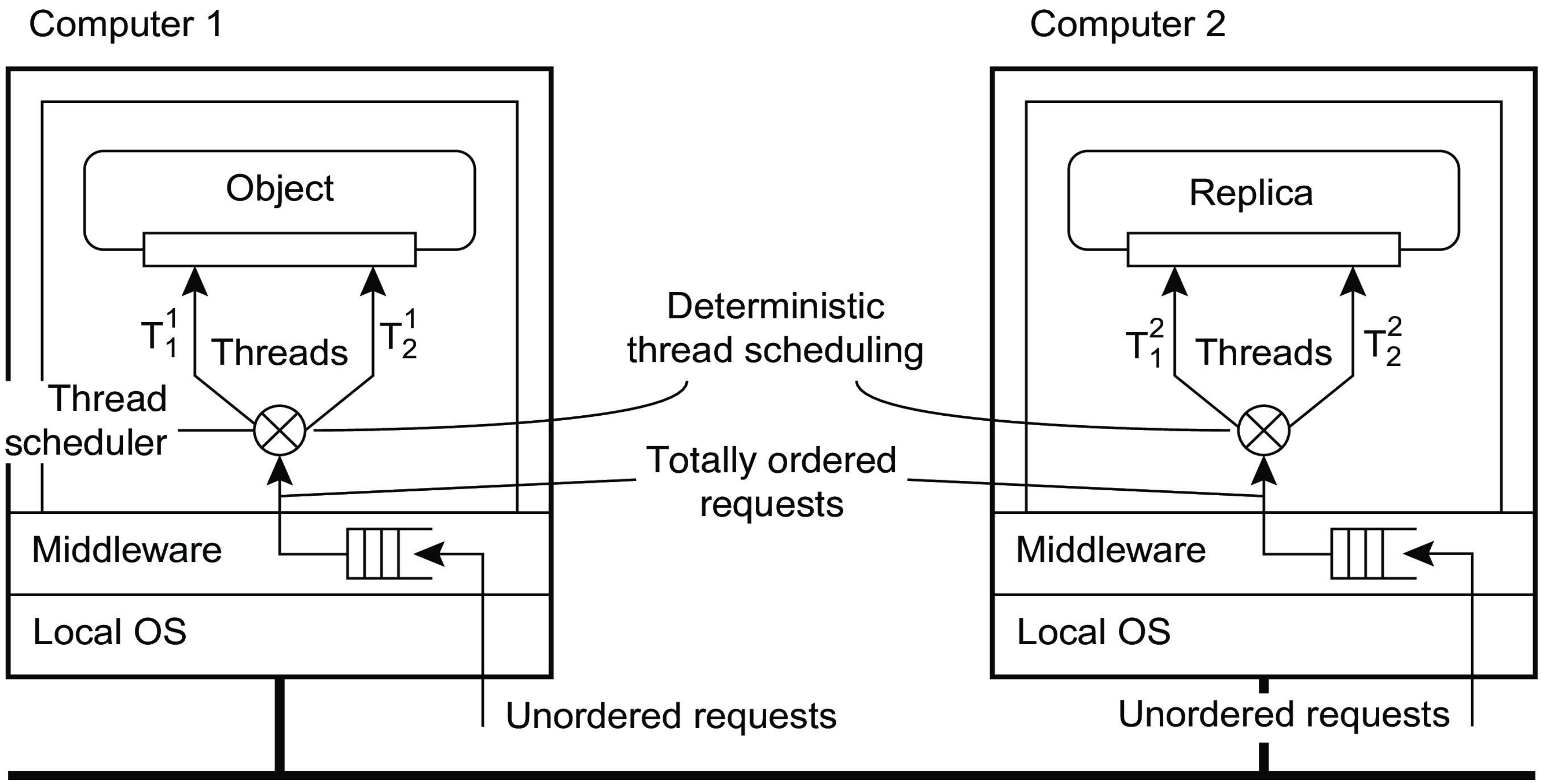
- Managing replicated objects
  - **Multithreaded Servers:** Multithreaded (object) servers pick up incoming requests, pass them to available threads, and wait for the next request. The server's thread scheduler then allocates the CPU to runnable threads. Even if the middleware ensures a total ordering for request delivery, thread schedulers should operate deterministically to maintain the order on the same object. Namely, we need **deterministic thread scheduling**. For example, if threads T1 and T2 handle the same incoming request, they should both be scheduled before T1 and T2 handle subsequent requests.



Replica Management

- Managing replicated objects

- However, Not all threads need to be scheduled deterministically. If we have totally ordered request delivery, we only need to ensure that requests for the same replicated object are handled in the order they were delivered. This allows invocations for different objects to be processed concurrently without additional restrictions from the thread scheduler. Few systems support this level of concurrency.

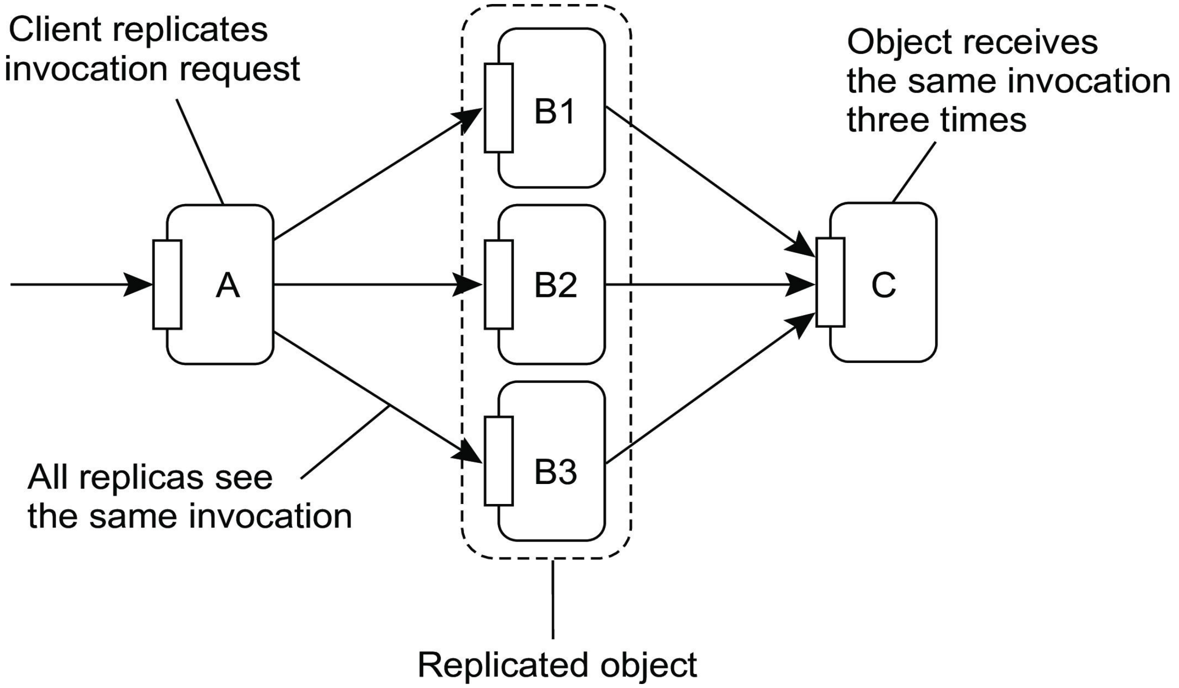


Replica Management

- Replicated-object invocations

Problem when invocating a replicated object

- Imagine you have an object A that calls another object B, as shown in Figure. Now, let's assume object B calls another object C. If B is replicated, each replica of B will call C independently. The problem here is that C ends up being called multiple times instead of just once. For example, suppose the method called on C results in a $100,000 transfer. If C gets called multiple times, it's pretty obvious that someone will eventually notice and complain about the unexpected extra transactions.
- This highlights a significant challenge in managing replicated objects and ensuring that certain operations, like critical transactions, are not unintentionally duplicated due to multiple invocations from replicas.

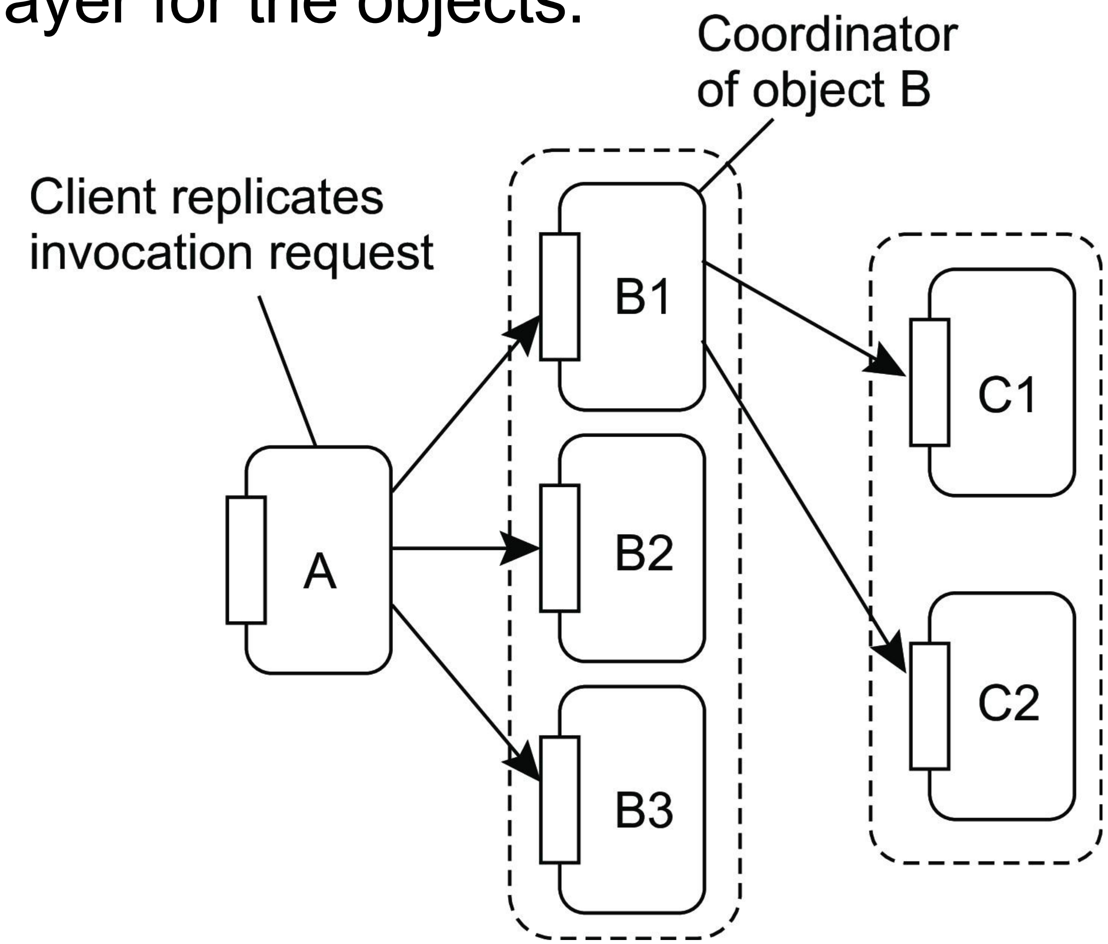

Client replicates invocation request
Object receives the same invocation three times
B1
A
B2
C
B3
All replicas see the same invocation
Replicated object

Replica Management

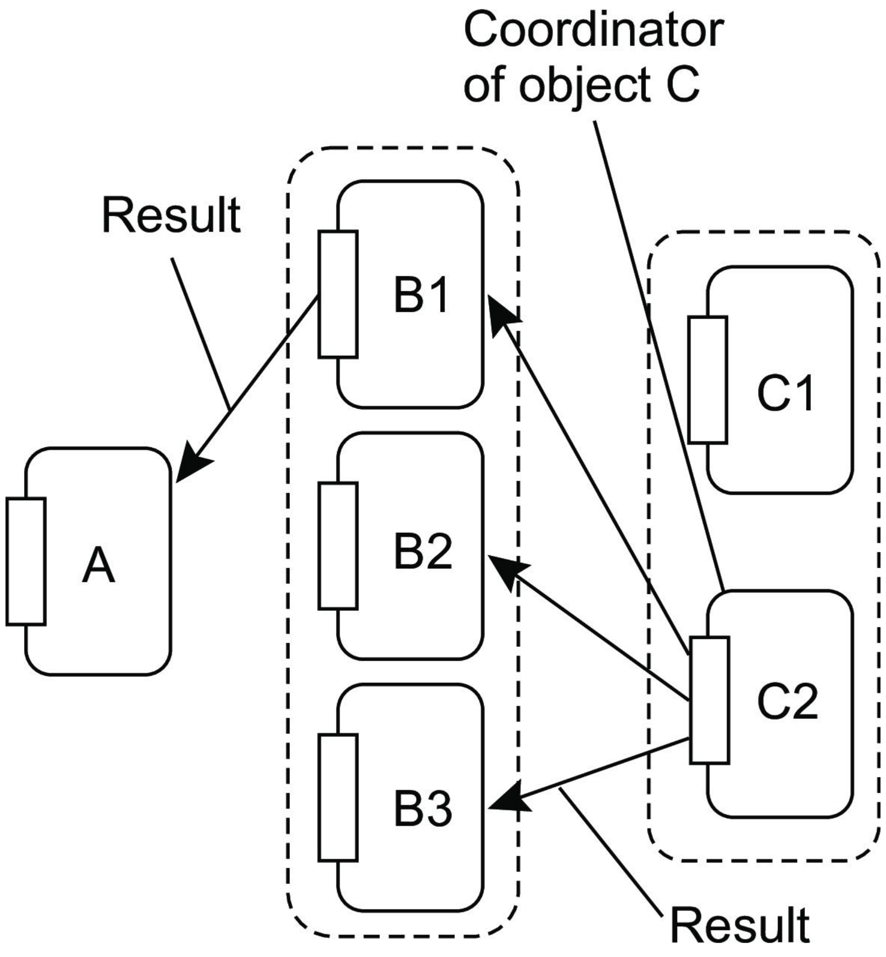
- Replicated-object invocations

Solution

- One simple solution is to just avoid replicated invocations. This makes sense when performance is important and you want to avoid the overhead. But if you're replicating for fault tolerance, there's a different approach. This solution doesn't depend on the specific replication policy, meaning it works no matter how you keep your replicas consistent. The key idea is to use a replication-aware communication layer for the objects.



Forwarding a request



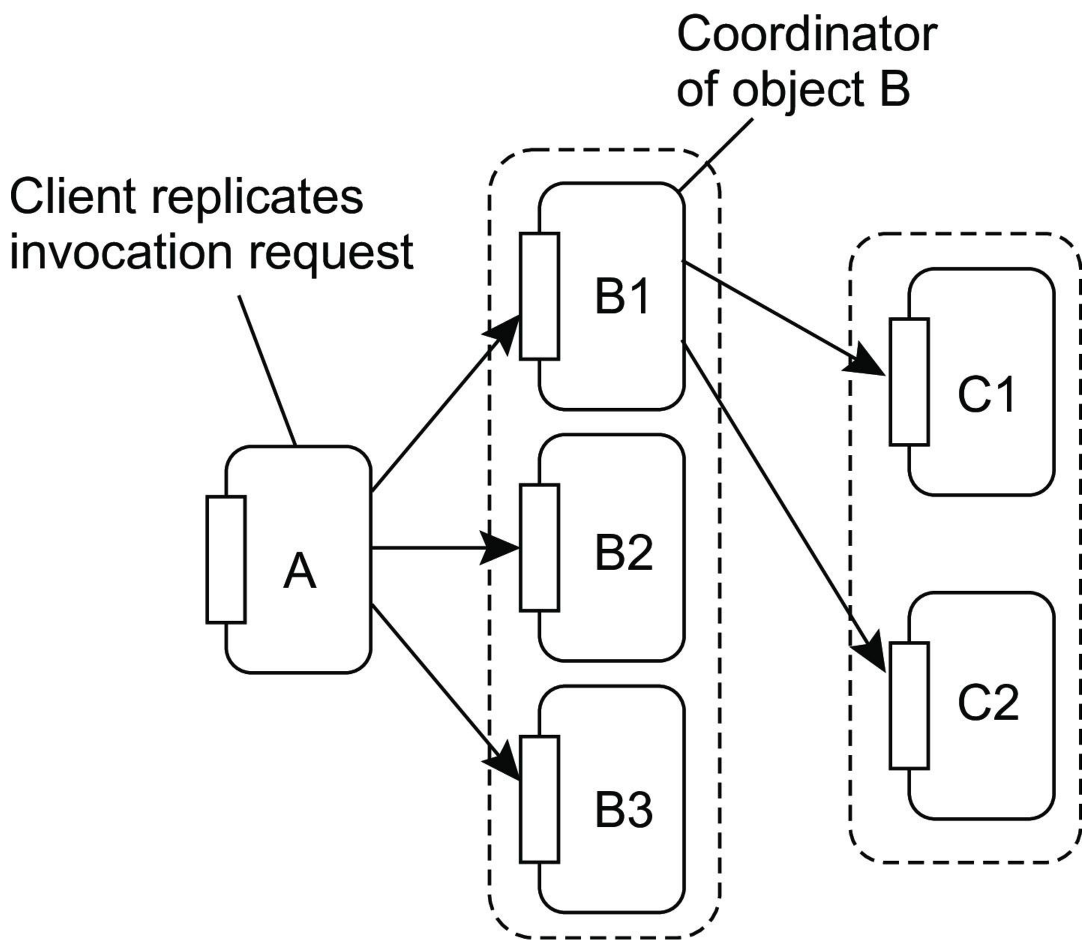
Returning the reply

Replica Management

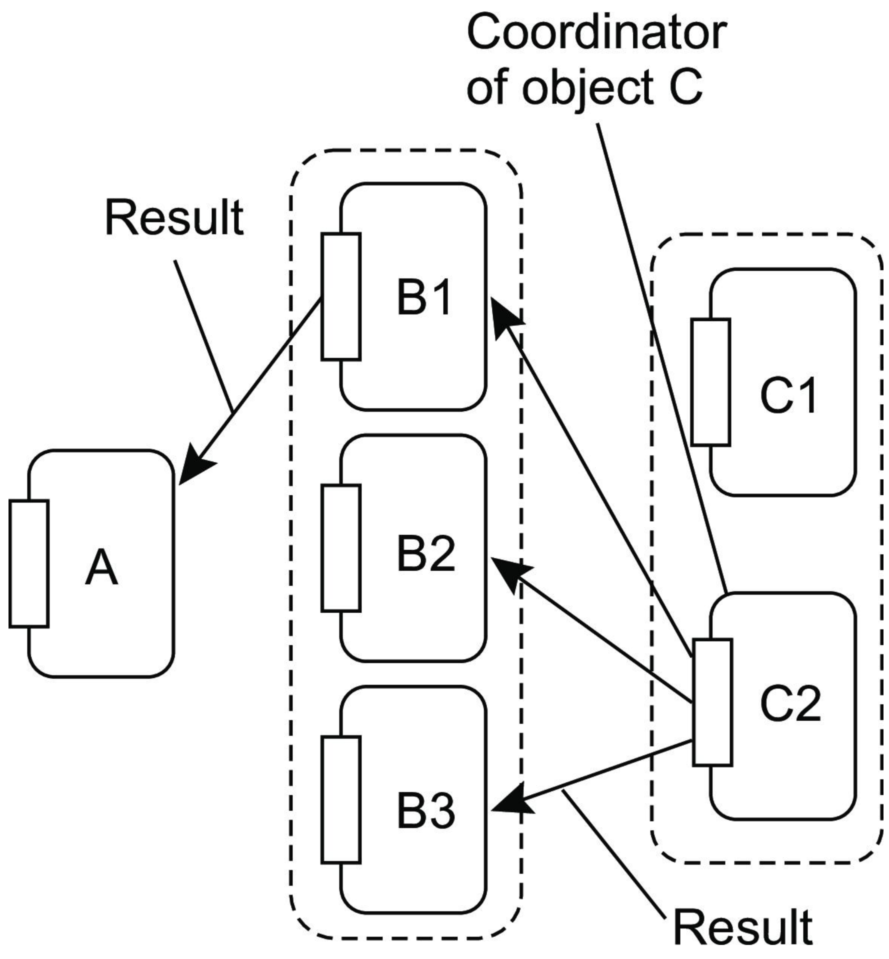
- Replicated-object invocations

Solution

- When a replicated object B calls another replicated object C, each replica of B assigns the same unique identifier to the invocation request. Then, a coordinator among B's replicas sends the request to all replicas of C, while the other B replicas hold back their copies of the request. This way, only one request is forwarded to each replica of C.



Forwarding a request



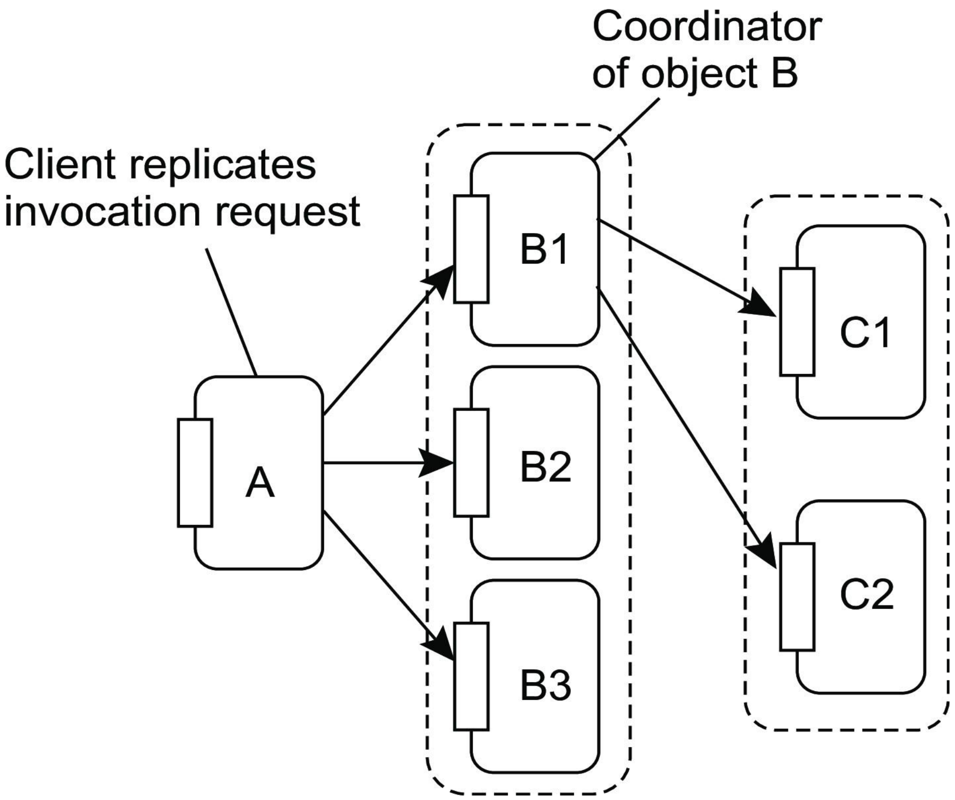
Returning the reply

Replica Management

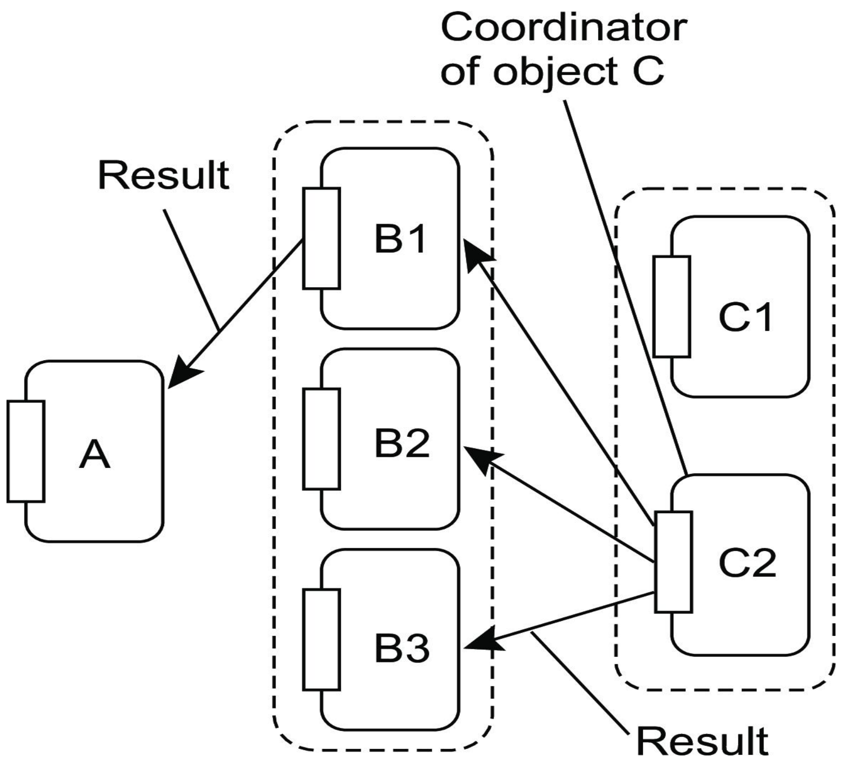
- Replicated-object invocations

  Solution

- The coordinator among C’s replicas handles the reply, forwarding it to B’s replicas, while the other C replicas hold back their replies. So, when a replica of B receives a reply to an invocation, it either sent to C or held back, it hands the reply to the actual object. In essence, this scheme uses multicast communication but avoids having the same message multicast by different replicas. It’s a sender-based scheme. An alternative would be to let a receiving replica detect multiple copies of the same message and only pass one copy to its object.



Forwarding a request



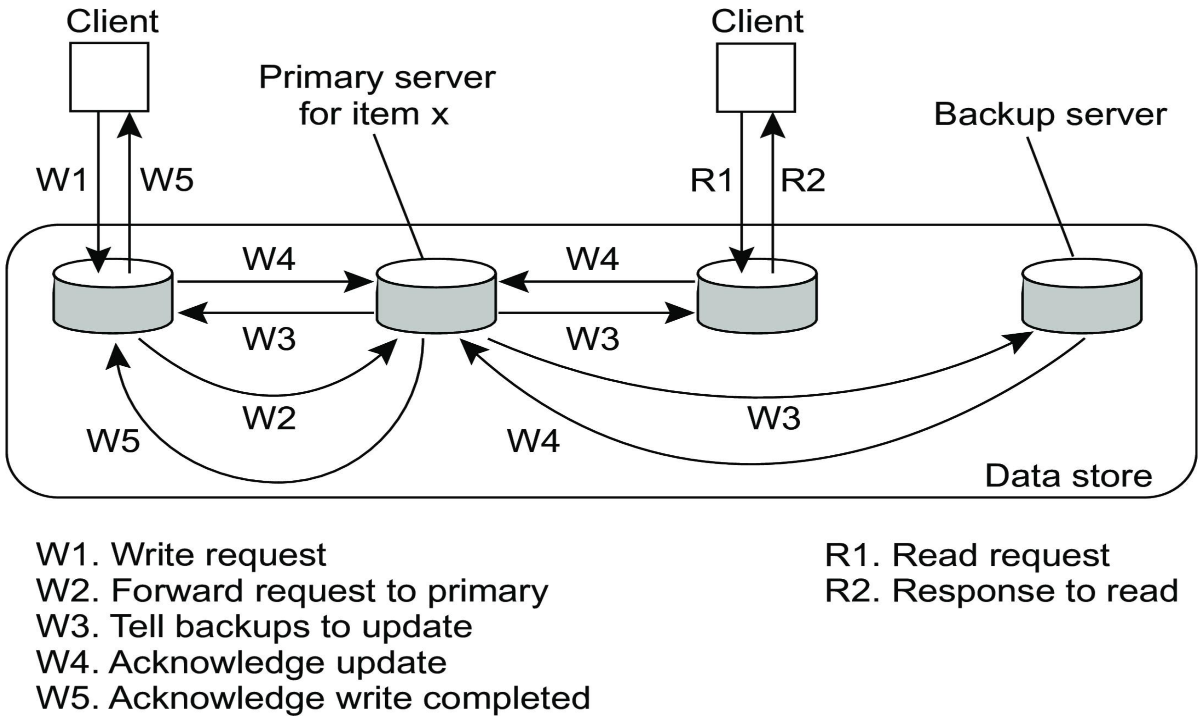
Returning the reply

Consistency Protocols

- Primary-based protocols

Primary-backup protocol

- For sequential consistency, primary-based protocols are common., each data item has an associated primary server that coordinates write operations. The primary can either be fixed at a remote server, or it can be moved to the process initiating the write. 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.



Example primary-backup protocol

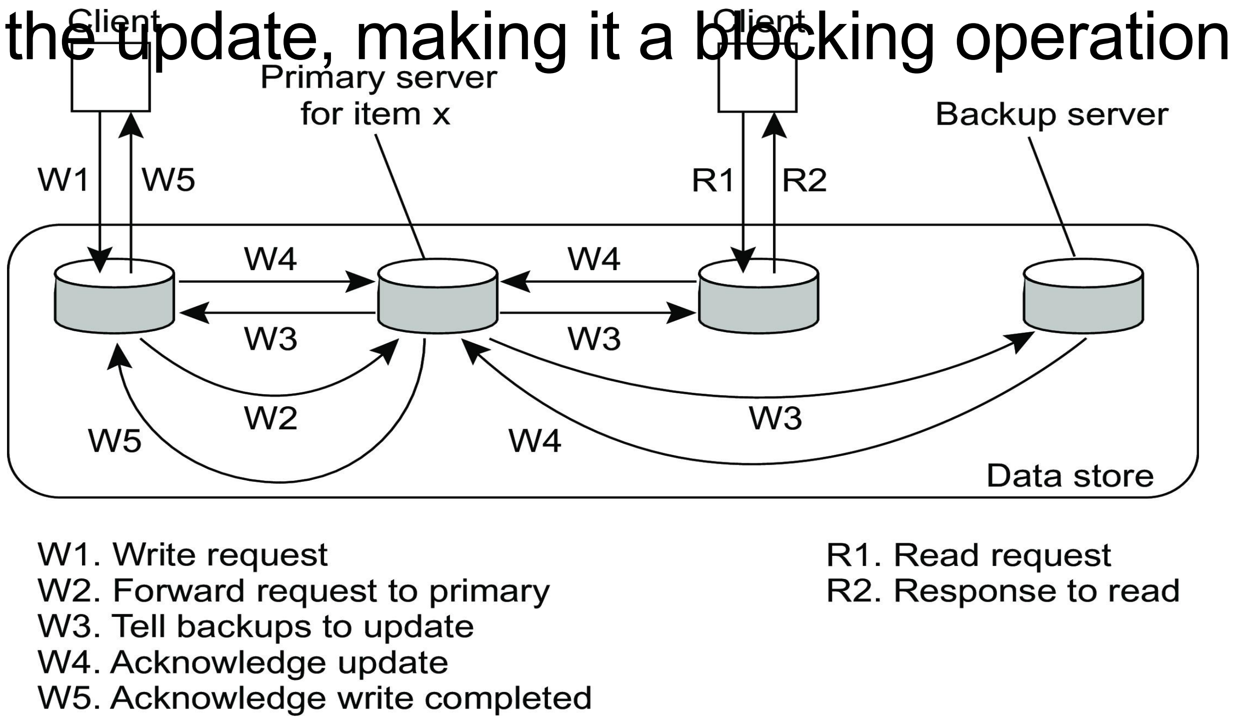
Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on the same LAN.

Consistency Protocols

- Primary-based protocols

Write Operations

- A process wanting to write to a data item sends the operation to the primary server.
- The primary updates its local copy and then forwards the update to backup servers.
- Each backup server updates its copy and sends an acknowledgment to the primary.
- Once all backups have updated, the primary sends an acknowledgment to the initial process, which informs the client.
- This method can be slow because the initiating process has to wait for all backups to acknowledge the update, making it a blocking operation.



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

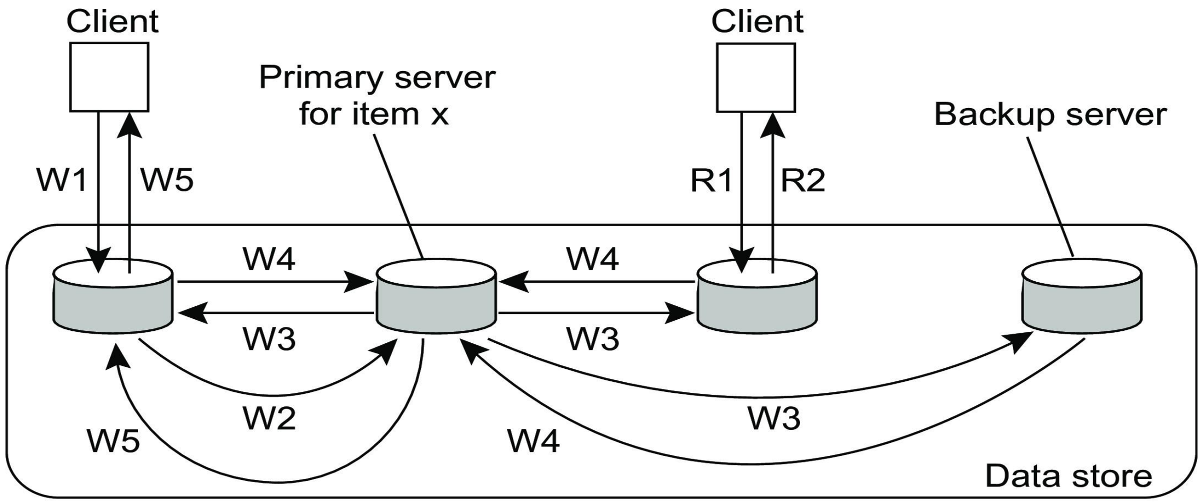
R1. Read request
R2. Response to read

Consistency Protocols

- Primary-based protocols

  Nonblocking Approach:

- As soon as the primary updates its copy, it sends an acknowledgment to the process.
- It then tells the backups to update their copies.
- This speeds up the write operations but sacrifices some fault tolerance. In the blocking scheme, the client knows for sure that the update is backed up by multiple servers. With the nonblocking approach, this certainty isn't there, but write operations are faster.



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

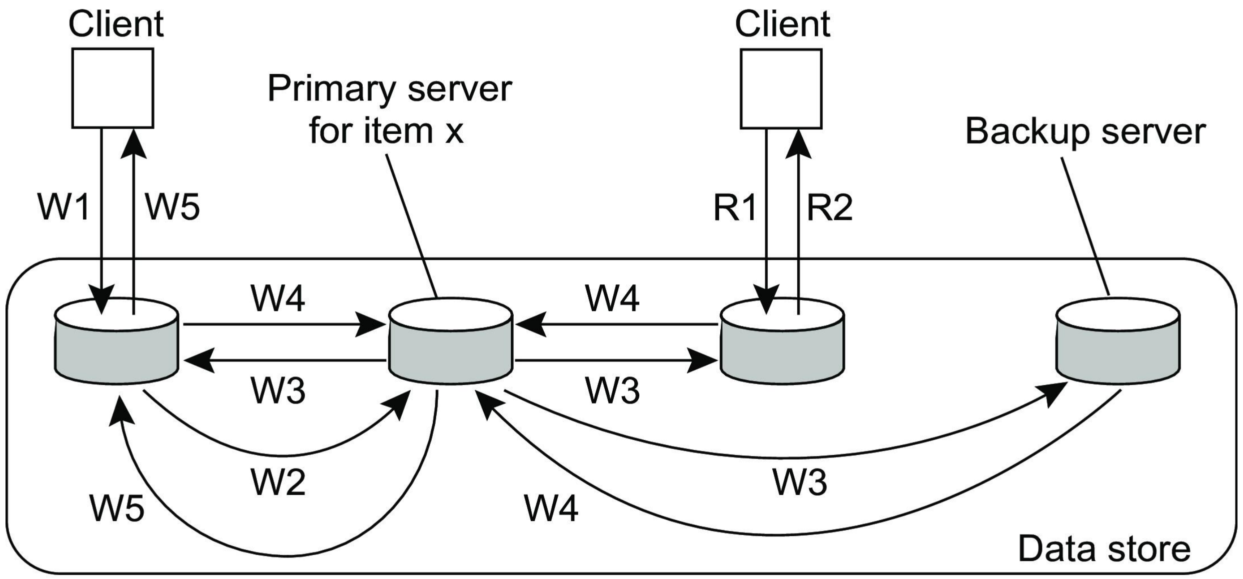
R1. Read request
R2. Response to read

Consistency Protocols

- Primary-based protocols

  Advantages and Disadvantages:

- Primary-backup protocols are straightforward for implementing sequential consistency because the primary can order all incoming writes uniquely. All processes see the writes in the same order, no matter which backup they read from. Blocking protocols ensure processes always see the effects of their most recent writes, which isn't guaranteed with nonblocking protocols without additional measures.



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

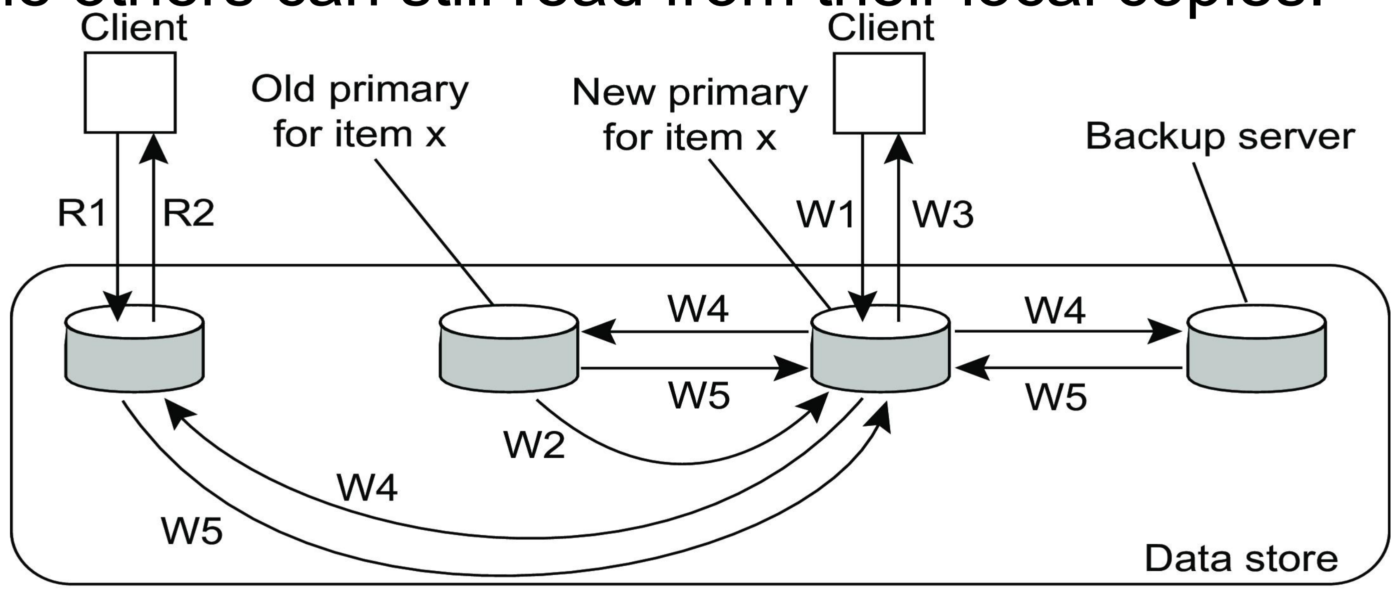
R1. Read request
R2. Response to read

Consistency Protocols

- Primary-based protocols

Primary-backup protocol with local writes

- In a variant of primary-backup protocols, the primary copy moves between processes that need to write data. When a process wants to update an item, it finds the primary copy and moves it to its location. This setup allows multiple local write operations, while others can still read from their local copies.



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

Example primary-backup protocol with local writes

Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).

Consistency Protocols

- Primary-based protocols

  ### Primary-backup protocol with local writes

  - This method works well if a nonblocking protocol is used, meaning updates are sent to replicas after the primary finishes local changes. It’s great for mobile devices that can work offline. Before disconnecting, the device becomes the primary for the data it will update. While offline, it handles all updates locally, and other processes can read but not write. Once reconnected, updates are sent to the backups, syncing everything up.
  - Another variant of this scheme is used in distributed file systems. Normally, there's a central server where all write operations happen, similar to the remote-write primary backup method. However, the central server can temporarily allow one of the replicas to perform a series of local updates, which can significantly speed up performance. Once the replica server is done, it sends the updates back to the central server, which then distributes them to the other replicas.

Consistency Protocols

- Implementing client-centric consistency

Keeping it simple

Each write operation *W* is assigned a globally unique identifier by its origin server. For each client, we keep track of two sets of writes:

- Read set: the (identifiers of the) writes relevant for that client's read operations
- Write set: the (identifiers of the) client's write operations.

Monotonic-read consistency

When a client wants to perform a read operation at a server, that server checks the client's read set to ensure all the identified writes have happened locally. If not, it contacts other servers to update itself before doing the read operation. Alternatively, the read operation can be forwarded to a server where the necessary writes have already been performed. After the read operation, any relevant writes from the selected server are added to the client's read set.

Consistency Protocols

- Implementing client-centric consistency

Keeping it simple

It’s important to know exactly where the writes in the read set have occurred. For example, the write identifier could include the server’s identifier where the operation was submitted. This server logs the write operation so it can be replayed at another server if needed. Also, write operations should be executed in the order they were submitted. This can be managed by letting the client generate a globally unique sequence number included in the write identifier. If each data item can only be modified by its owner, that owner provides the sequence number.

Monotonic-write consistency

It works similarly to monotonic reads. When a client starts a new write operation at a server, it hands over its write set to the server. The server then ensures the identified writes are performed first and in the correct order. After the new operation, the write identifier is added to the write set. However, updating the current server with the client’s write set can significantly increase the client’s response time because the client waits for the operation to fully complete.

Consistency Protocols

- Implementing client-centric consistency

Read-your-writes consistency

When a client reads from a server, that server has seen all the writes in the client's write set. One way to do this is for the server to fetch these writes from other servers before performing the read operation, although this can slow down response time. Alternatively, the client-side software can search for a server where all the necessary writes have already been completed.

When client *C* wants to read at server *S*, *C* passes its write set. *S* can pull in any updates before executing the read operation, after which the read set is updated.

Writes-follows-reads consistency

We first need to update the selected server with the writes in the client's read set. After that, the new write operation's identifier is added to the write set, along with the identifiers in the read set that are now relevant for the new write.

When client *C* wants to write at server *S*, *C* passes its read set. *S* can pull in any updates, executes them in the correct order, and then executes the write operation, after which the write set is updated.

Example: Caching and replication in the Web

Client-side caches

- In the browser. Most browsers have a simple caching system. When a document is fetched, it's stored in the browser's cache and loaded from there the next time.
- At a client's site, notably through a Web proxy. A client's site often runs a web proxy, which accepts requests from local clients and passes them to web servers. When the response comes back, the proxy caches the result and can return it to other clients if needed. This means a web proxy can act as a shared cache.
- Given the number of documents generated dynamically, servers often provide documents in pieces and instruct clients to cache only the parts that are unlikely to change on subsequent requests.

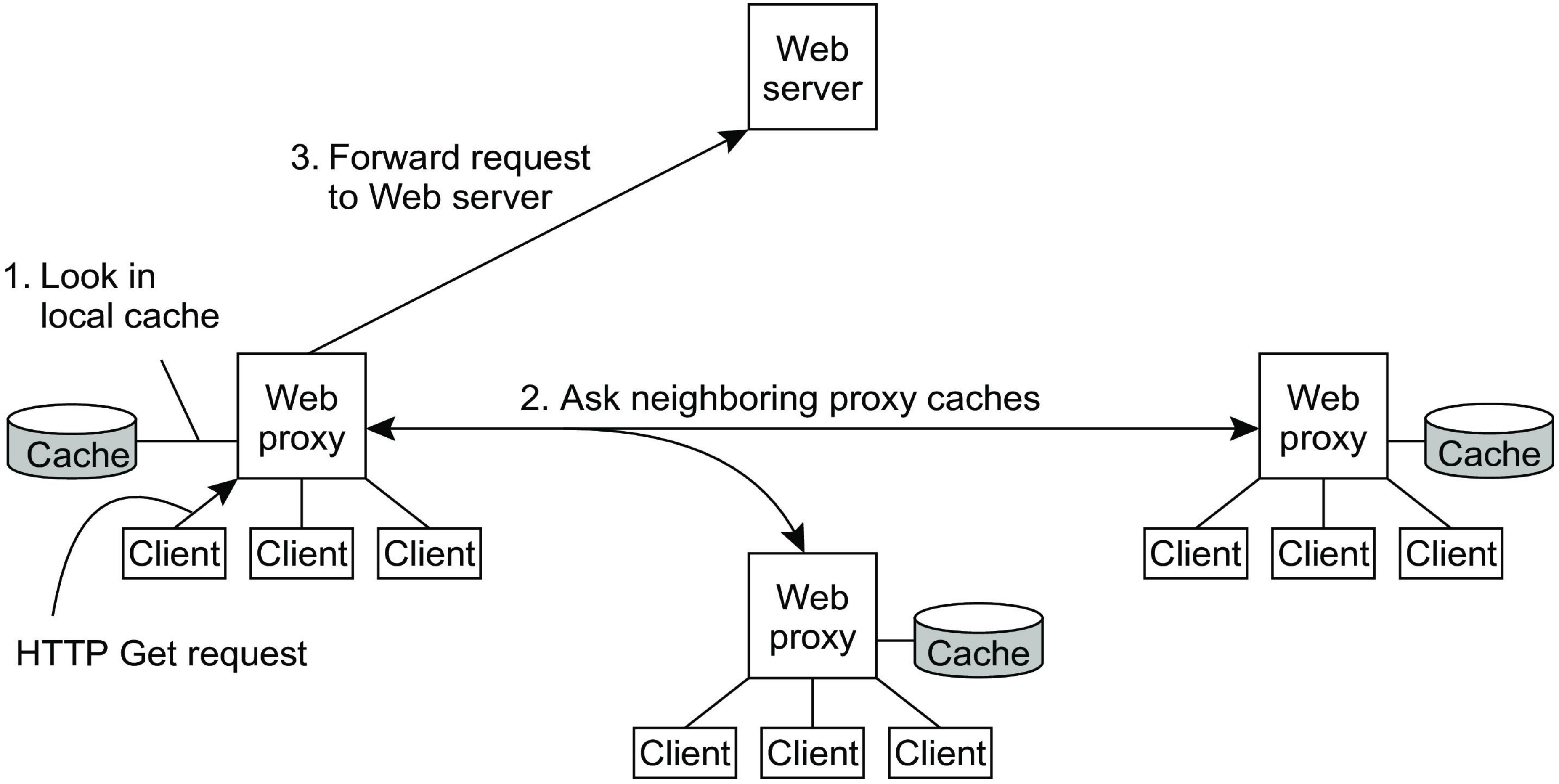
Caches at ISPs

Internet Service Providers also place caches to (1) reduce cross-ISP traffic and (2) improve client-side performance.). However, with multiple caches along the request path from client to server, there's a risk of increased latency if the caches don't contain the requested information.

Example: Caching and replication in the Web

- Cooperative caching

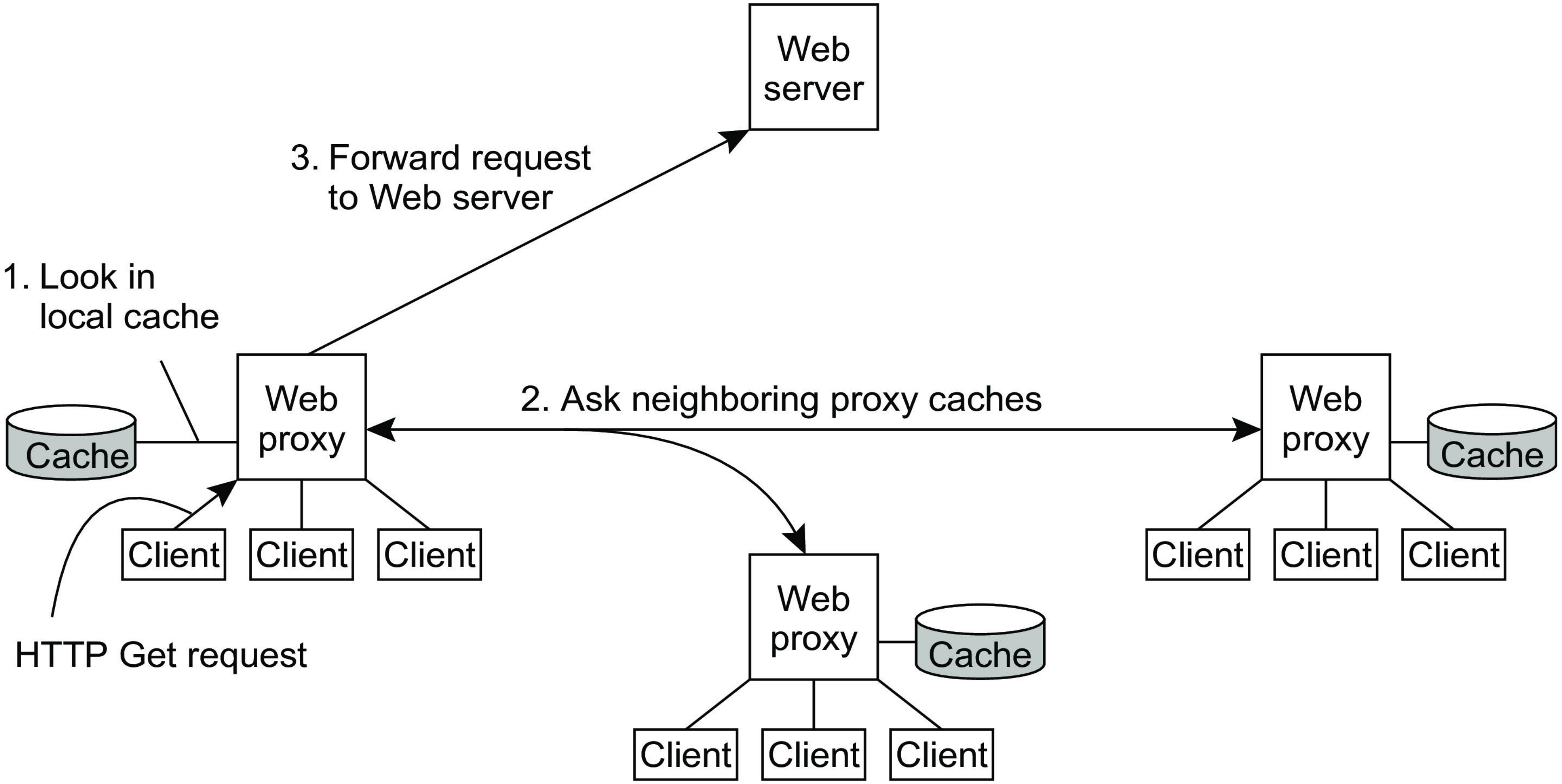
- Instead of building hierarchical caches, you can also set up caches for cooperative deployment, as shown in Figure . In cooperative or distributed caching, when a web proxy has a cache miss, it first checks with neighboring proxies to see if they have the requested document. If none of them have it, the proxy then forwards the request to the web server responsible for the document. This method is typically used with web caches that belong to the same organization or institution.



Example: Caching and replication in the Web

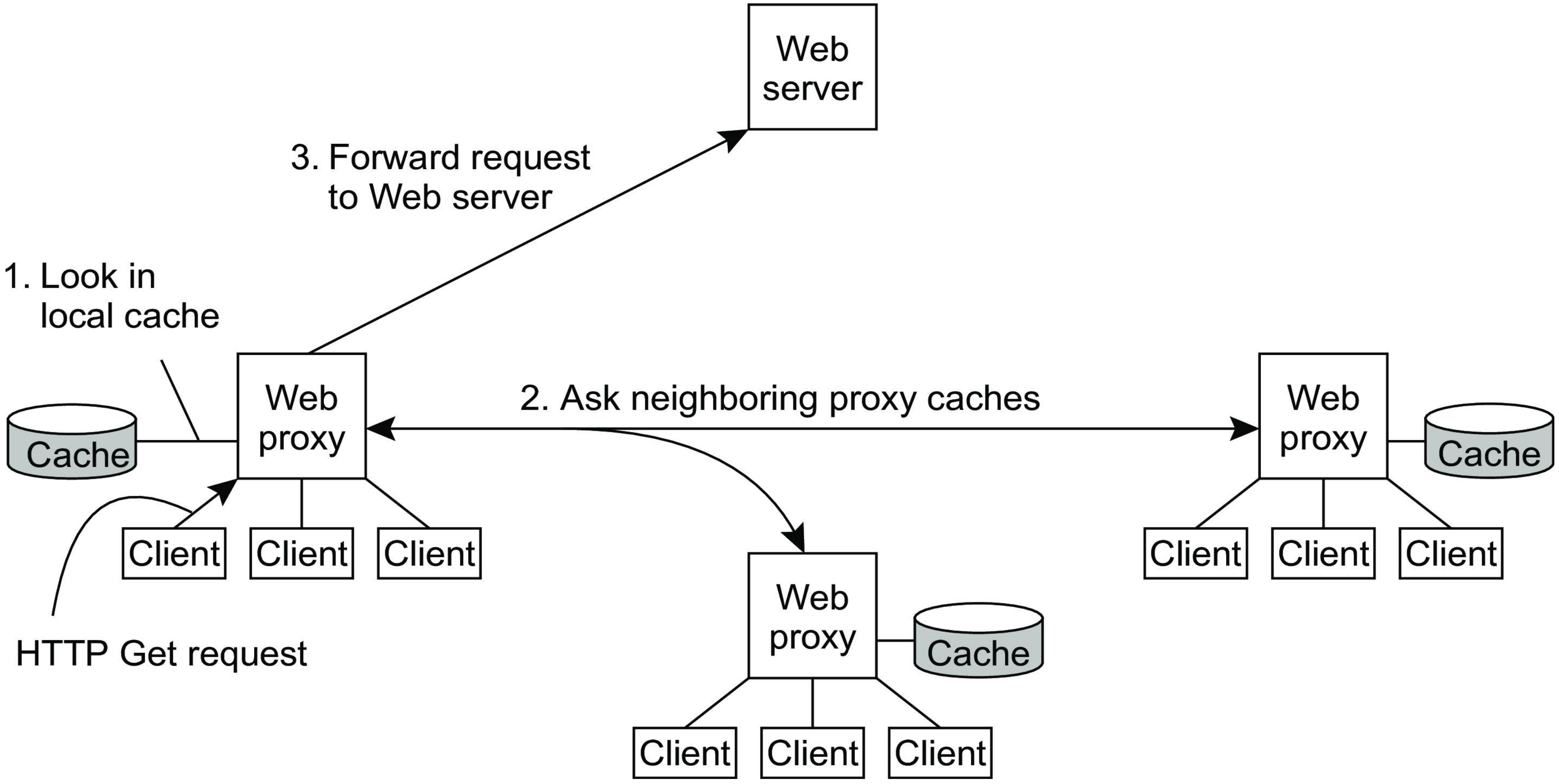
- Cooperative caching

- Cooperative caching works well for relatively small groups of clients, like tens of thousands of users. However, these groups could also be served by a single proxy cache, which is cheaper in terms of communication and resource usage. In a highly decentralized system, cooperative caching was very effective. These studies don't necessarily contradict each other; they show that the effectiveness of cooperative caching depends heavily on client demands.



Example: Caching and replication in the Web

- Cooperative caching
- **Comparison with Hierarchical Caching:** Compared hierarchical and cooperative caching, there several trade-offs. For example, since cooperative caches are usually connected through high-speed links, the time needed to fetch a document is much lower than with a hierarchical cache. Also, storage requirements are less strict for cooperative caches compared to hierarchical ones.



Example: Caching and replication in the Web

- Web-cache consistency

How to guarantee freshness?

To prevent that stale information is returned to a client:

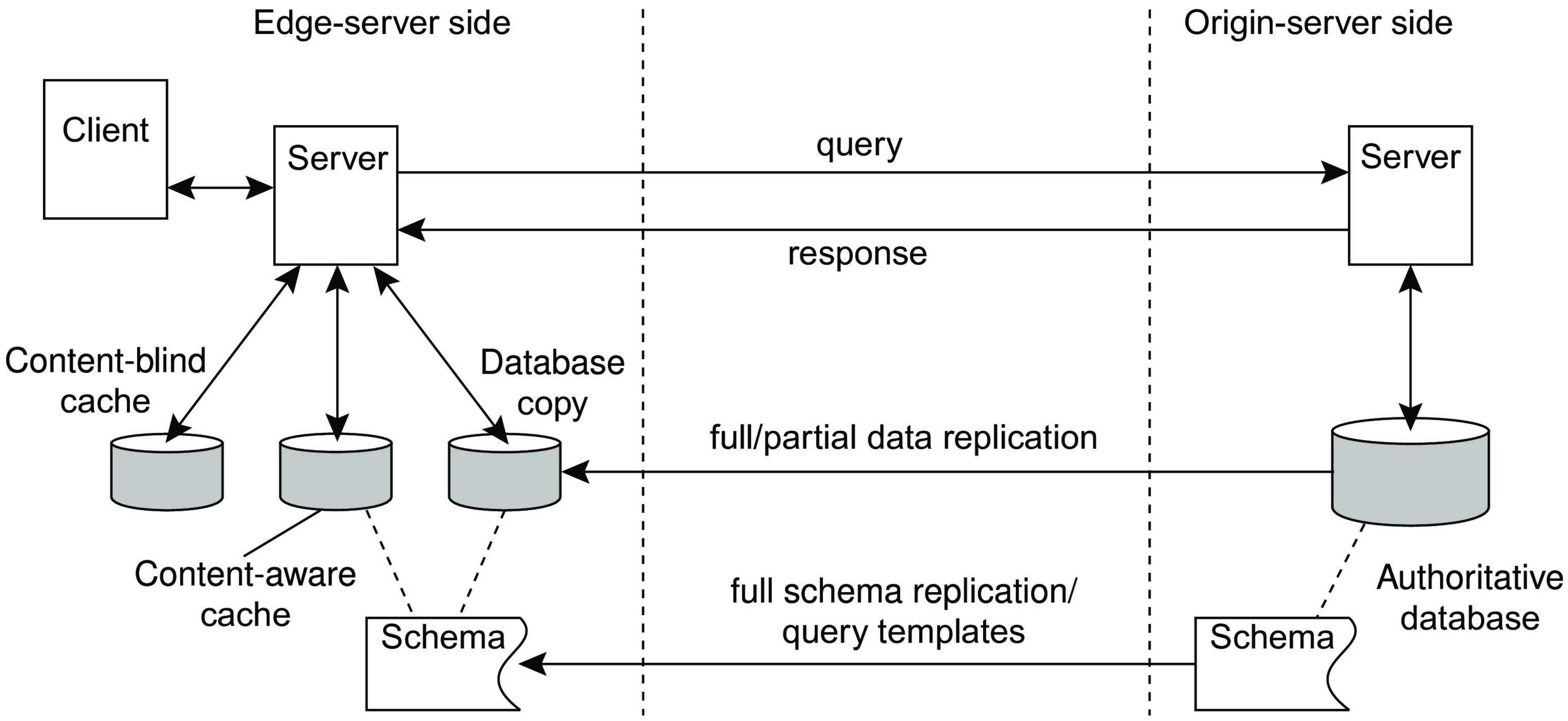
- Option 1: let the cache contact the original server to see if content is still up to date.
- Option 2: Assign an expiration time T_{expire} that depends on how long ago the document was last modified when it is cached. If $T_{last\ modified}$ is the last modification time of a document (as recorded by its owner), and T_{cached} is the time it was cached, then

$$T_{expire} = \alpha(T_{cached} - T_{last\ \ modified}) + T_{cached}$$

with $\alpha = 0.2$. Until T_{expire}, the document is considered valid.

Example: Caching and replication in the Web

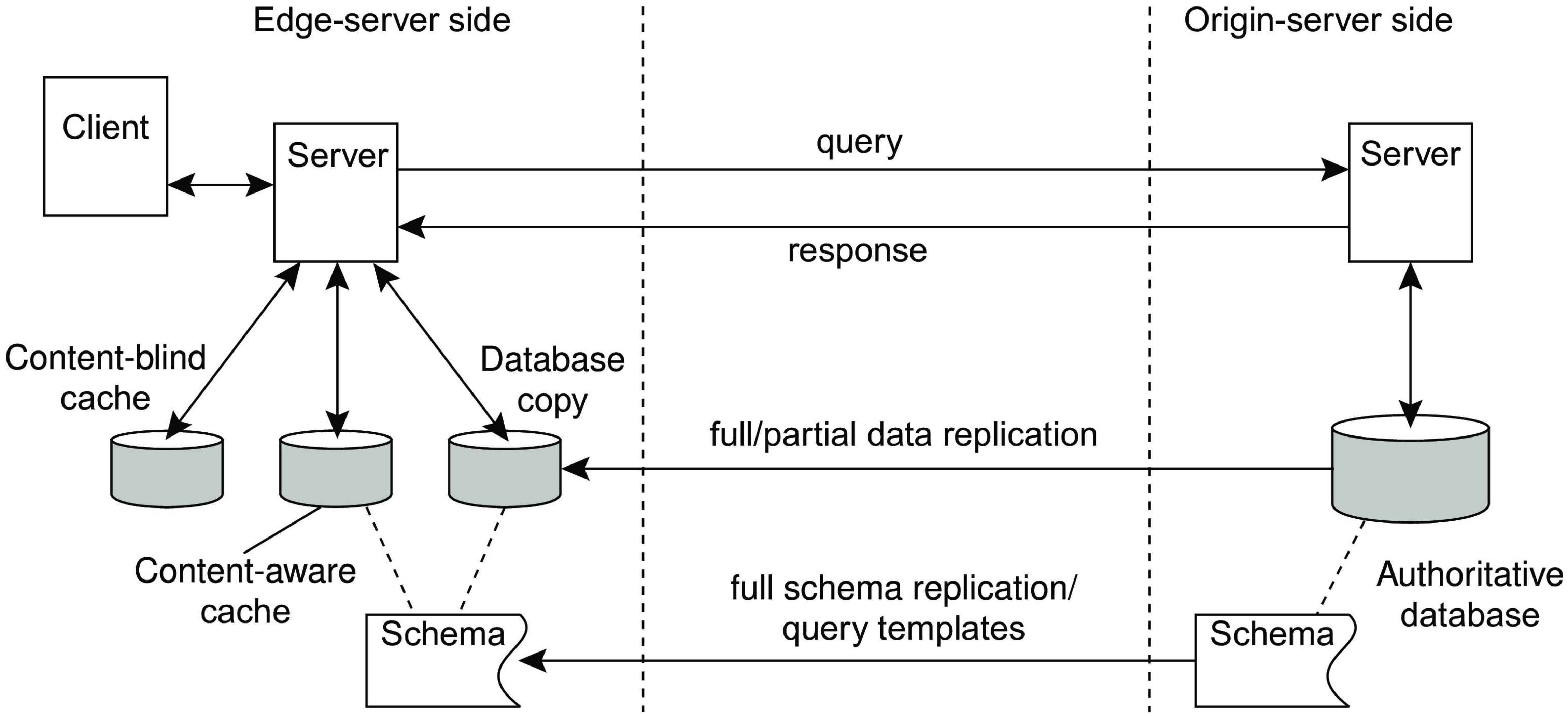
- Alternatives for caching and replication



- In an edge-server architecture, an edge server handles client requests and can store some of the information from the origin server. Web clients request data through an edge server, which then gets its information from the origin server. The origin server often consists of a database that dynamically creates responses. Typically, each server is organized in a multitiered architecture, as we've discussed before.

Example: Caching and replication in the Web

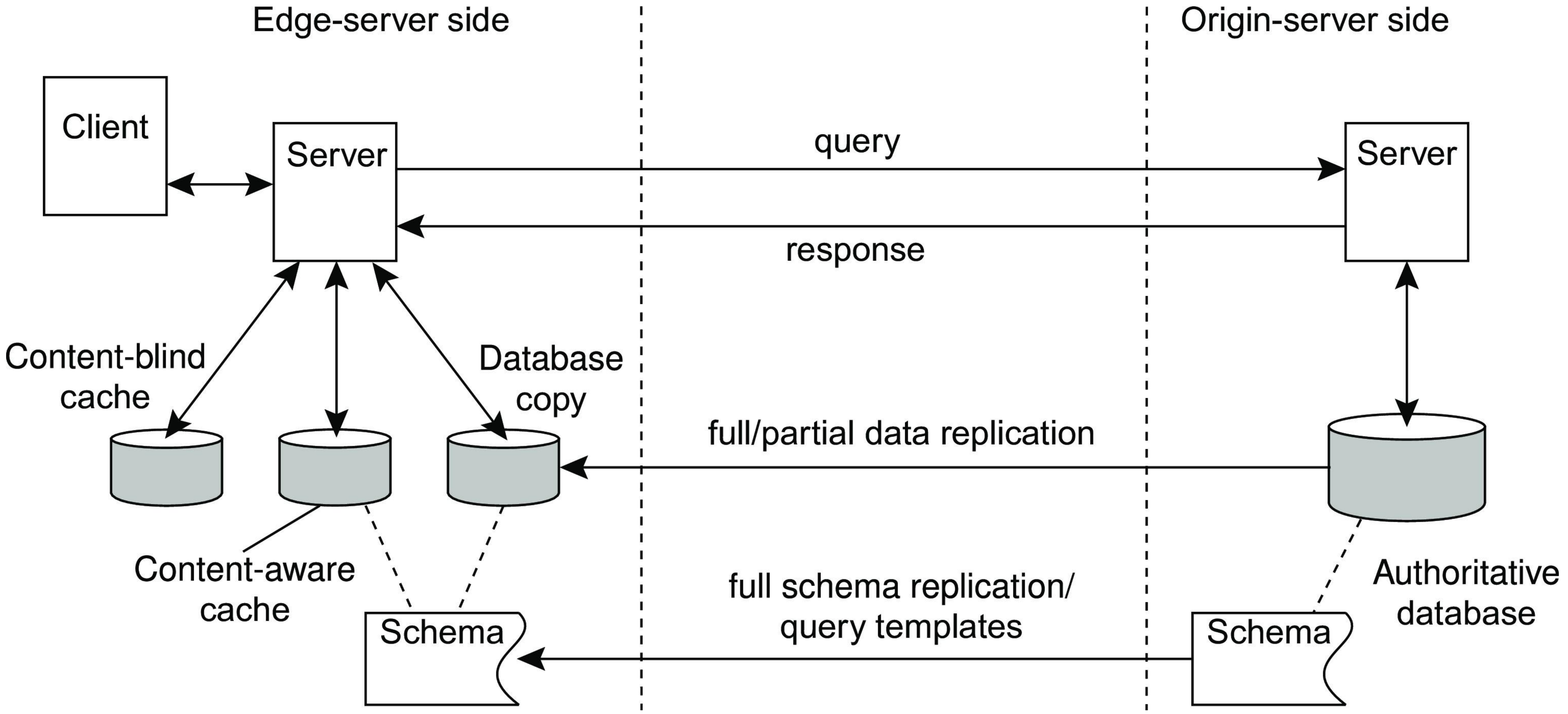
- Alternatives for caching and replication



- Full Replication: You can fully replicate the data stored at the origin server. This works well when the update ratio is low and queries require extensive database searches. All updates happen at the origin server, which then keeps the replicas and edge servers consistent. Read operations happen at the edge servers. However, if the update ratio is high, replicating for performance fails because each update incurs communication over a wide-area network to keep replicas consistent.

Example: Caching and replication in the Web

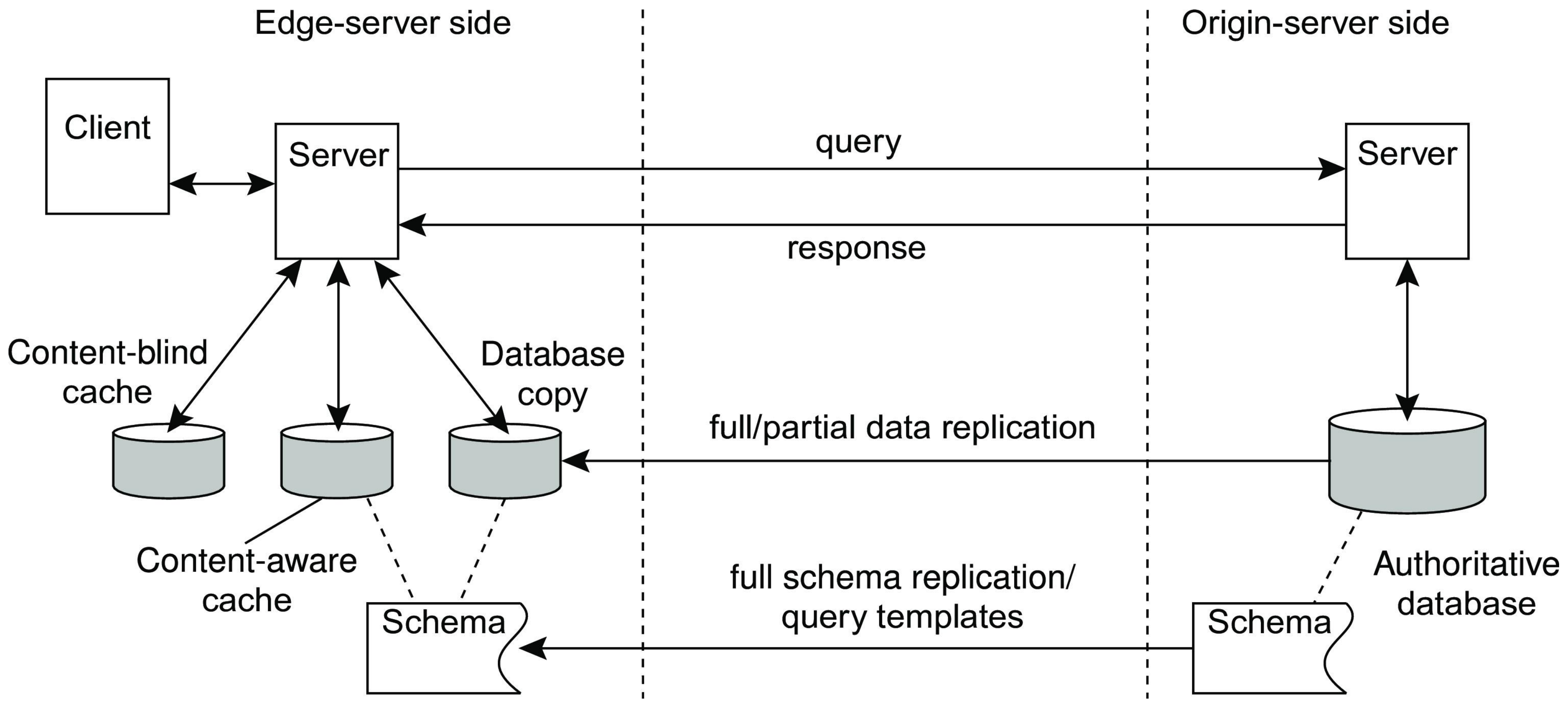
- Alternatives for caching and replication



- Partial Replication: If queries are generally complex, involving multiple tables and operations like joins, full replication might be needed. But for simpler queries, which generally access a single table, partial replication (storing only a subset of data at the edge server) might suffice.

Example: Caching and replication in the Web

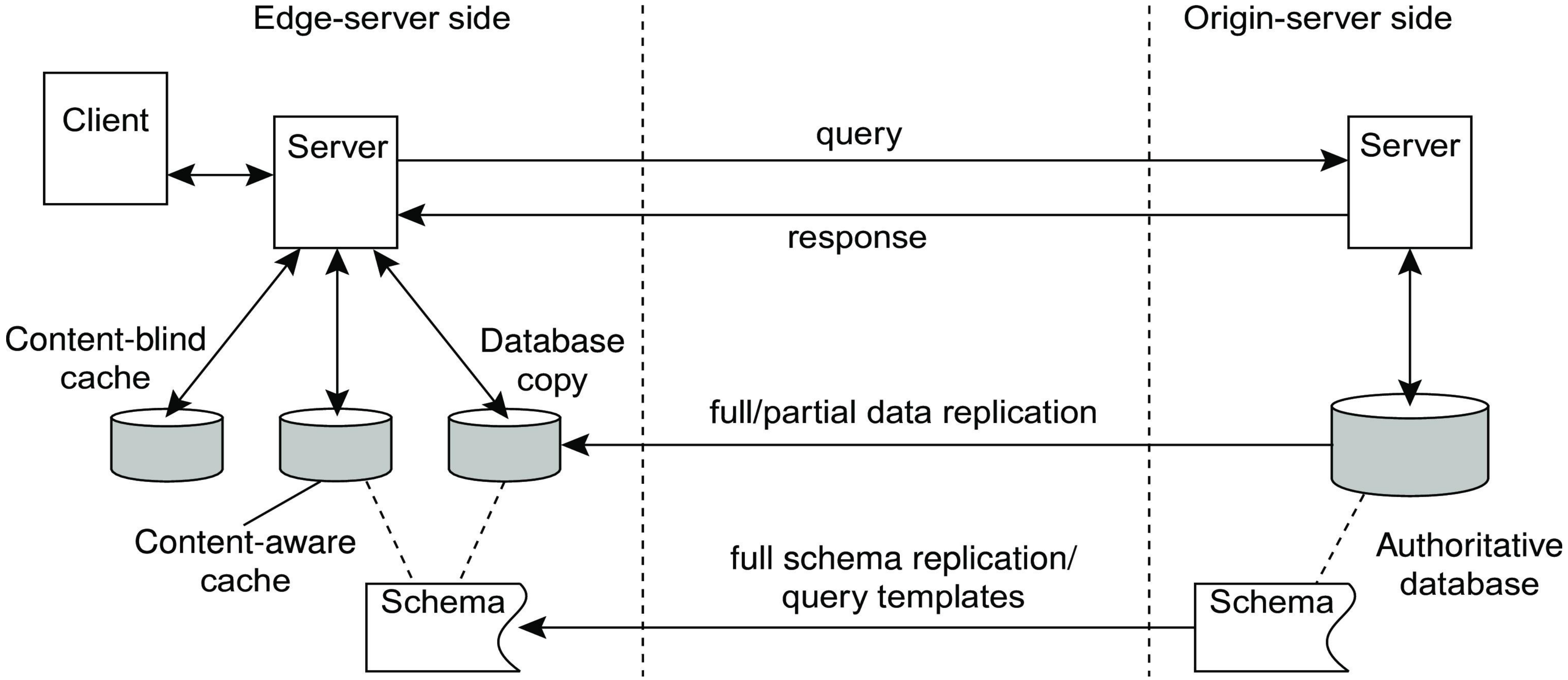
- Alternatives for caching and replication



- Content-Aware Caches: The edge server maintains a local database tailored to the type of queries handled by the origin server. A full-fledged database system organizes data into tables to minimize redundancy (normalization). However, with content-aware caches, the edge server organizes its local database according to query structures, assuming queries follow a limited number of templates. When a query arrives, the edge server matches it against available templates and checks its local database for a response. If the data isn't available locally, the query is forwarded to the origin server, and the response is cached for future use. This method works best when queries are often repeated.

Example: Caching and replication in the Web

- Alternatives for caching and replication



- Content-Blind Caching: When a client submits a query, the edge server computes a unique hash value for that query and checks its cache for a match. If the query hasn’t been processed before, it’s forwarded to the origin server, and the result is cached. If the query has been processed, the cached result is returned to the client. This method reduces the computational effort required from the edge server but can be wasteful in terms of storage since the cache may contain redundant data. Keeping such a cache up-to-date can be challenging, but this can be mitigated by assuming queries match a limited set of predefined templates.

Thank You