# Principles of Distributed Systems

inft-3507

Dr. J.Burns

**ADA University**

Autumn 2025

## Section 7: Consistency and Replication

# Introduction

## 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 one copy does not live up to specifications, switch over to the other copy while repairing the failing one.
- Performance: simply spread requests between different replicated parts to keep load balanced, or to ensure quick responses by taking proximity into account.

### 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.

# Performance and scalability

## Main issue

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

## Conflicting operations: From the world of transactions

- Read–write conflict: a read operation and a write operation act concurrently
- Write–write conflict: two concurrent write operations

## Issue

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability.
Solution: weaken consistency requirements so that hopefully global synchronization can be avoided
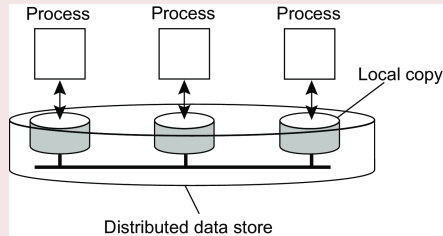
# Data-centric consistency models

## Data-centric consistency models

### Consistency model

A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

### Essential

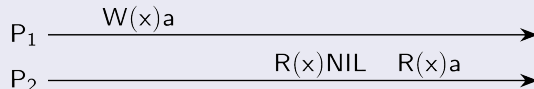A data store is a distributed collection of storages:

# Some notations

## 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 $x$
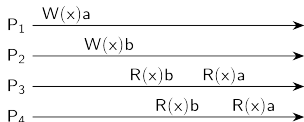- All data items initially have value *NIL*

## Possible behavior

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

$$P_1 \xrightarrow{\quad W(x)a \quad}$$

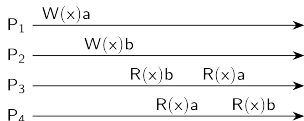$$P_2 \xrightarrow{\quad R(x)NIL \quad R(x)a \quad}$$

# 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.

$$
\begin{array}{ll}
P_1 & \text{---}\underline{W(x)a}\text{---------------------------}\rightarrow \\
P_2 & \text{-------}\underline{W(x)b}\text{-----------------------}\rightarrow \\
P_3 & \text{-----------------}\underline{R(x)b}\quad\underline{R(x)a}\text{-----}\rightarrow \\
P_4 & \text{-----------------------}\underline{R(x)b}\quad\underline{R(x)a}\rightarrow
\end{array}
$$

A sequentially consistent data store

$$
\begin{array}{ll}
P_1 & \text{---}\underline{W(x)a}\text{---------------------------}\rightarrow \\
P_2 & \text{-------}\underline{W(x)b}\text{-----------------------}\rightarrow \\
P_3 & \text{-----------------}\underline{R(x)b}\quad\underline{R(x)a}\text{-----}\rightarrow \\
P_4 & \text{-----------------------}\underline{R(x)a}\quad\underline{R(x)b}\rightarrow
\end{array}
$$

A data store that is not sequentially consistent

# Example

## Three concurrent processes (initial values: 0)

| Process $P_1$ | Process $P_2$ | Process $P_3$ |
|---|---|---|
| $x \leftarrow 1;$ | $y \leftarrow 1;$ | $z \leftarrow 1;$ |
| print(y,z); | print(x,z); | print(x,y); |

## Example execution sequences

| Execution 1 | Execution 2 | Execution 3 | Execution 4 |
|---|---|---|---|
| $P_1$:  $x \leftarrow 1;$ | $P_1$:  $x \leftarrow 1;$ | $P_2$:  $y \leftarrow 1;$ | $P_2$:  $y \leftarrow 1;$ |
| $P_1$:  print(y,z); | $P_2$:  $y \leftarrow 1;$ | $P_3$:  $z \leftarrow 1;$ | $P_1$:  $x \leftarrow 1;$ |
| $P_2$:  $y \leftarrow 1;$ | $P_2$:  print(x,z); | $P_3$:  print(x,y); | $P_3$:  $z \leftarrow 1;$ |
| $P_2$:  print(x,z); | $P_1$:  print(y,z); | $P_2$:  print(x,z); | $P_2$:  print(x,z); |
| $P_3$:  $z \leftarrow 1;$ | $P_3$:  $z \leftarrow 1;$ | $P_1$:  $x \leftarrow 1;$ | $P_1$:  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 | *Prints:* 101011 | *Prints:* 010111 | *Prints:* 111111 |
| *Signature:* 0 0 1 0 1 1 | *Signature:* 1 0 1 0 1 1 | *Signature:* 1 1 0 1 0 1 | *Signature:* 1 1 1 1 1 1 |
| (a) | (b) | (c) | (d) |

## How tricky can it get?

### Seemingly okay

$$P_1 \xrightarrow{\quad W(x)a \qquad\qquad W(y)a \qquad R(x)a \quad}$$
$$P_2 \xrightarrow{\quad W(y)b \qquad\qquad W(x)b \qquad R(y)b \quad}$$
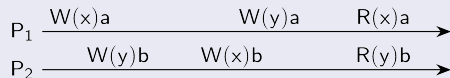
### But not really (don't forget that $P_1$ and $P_2$ act concurrently)

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

## How tricky can it get?

### Linearizability

Each operation should appear to take effect instantaneously at some moment between its start and completion.

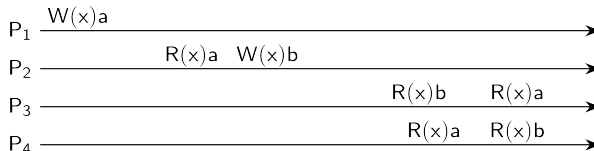### Operations complete within a given time (shaded area)

$P_1$ — W(x)a — W(y)a — R(x)a →

$P_2$ — W(y)b — W(x)b — R(y)b →
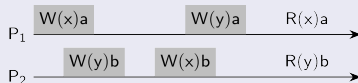
### With better results

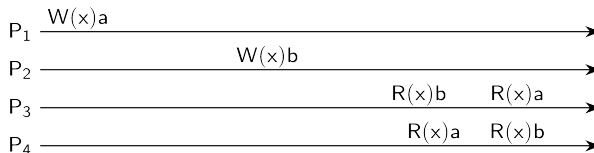| Possible ordering of operations | Result | |
|---|---|---|
| $W_1(x)a$; $W_2(y)b$; $W_1(y)a$; $W_2(x)b$ | $R_1(x)b$ | $R_2(y)a$ |
| $W_1(x)a$; $W_2(y)b$; $W_2(x)b$; $W_1(y)a$ | $R_1(x)b$ | $R_2(y)a$ |
| $W_2(y)b$; $W_1(x)a$; $W_1(y)a$; $W_2(x)b$ | $R_1(x)b$ | $R_2(y)a$ |
| $W_2(y)b$; $W_1(x)a$; $W_2(x)b$;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

## Causal consistency

### Definition

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.

$P_1$ ──── W(x)a ────────────────────────────────►

$P_2$ ──────── R(x)a   W(x)b ────────────────────►

$P_3$ ──────────────────────── R(x)b      R(x)a ──►

$P_4$ ──────────────────────── R(x)a      R(x)b ──►

A violation of a causally-consistent store

$P_1$ ──── W(x)a ────────────────────────────────►

$P_2$ ──────────── W(x)b ──────────────────────────►

$P_3$ ──────────────────────── R(x)b      R(x)a ──►

$P_4$ ──────────────────────── R(x)a      R(x)b ──►

A correct sequence of events in a causally-consistent store

## Consistency models, serializability, transactions

### Overwhelming, but often already known

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? The keyword is serializability.

| 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

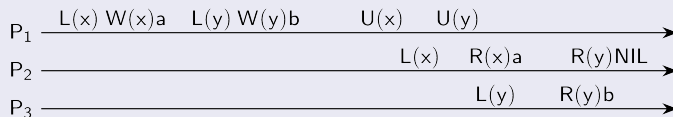|    | Time $\longrightarrow$ |            |            |            |            |            |         |
|----|-------|-----------|-----------|-----------|-----------|-----------|---------|
| S1 | x = 0 | x = x + 1 | x = 0     | x = x + 2 | x = 0     | x = x + 3 | Legal   |
| S2 | x = 0 | x = 0     | 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 |

## Grouping operations

### Entry consistency: Definition

- Accesses to locks are sequentially consistent.

- No access to a lock is allowed to be performed until all previous writes have completed everywhere.

- No data access is allowed to be performed until all previous accesses to locks have been performed.

### Basic idea

You don't care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.

# Grouping operations

## A valid event sequence for entry consistency

$P_1$ — L(x) W(x)a    L(y) W(y)b        U(x)    U(y) →

$P_2$ ————————————————————— L(x)    R(x)a        R(y)NIL →

$P_3$ ————————————————————————— L(y)        R(y)b →

## Observation

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

## Question

What would be a convenient way of making this consistency more or less transparent to programmers?

## Eventual consistency

### Definition

Consider a collection of data stores and (concurrent) write operations. The strores 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).

### Srong eventual consistency

Basic idea: if there are conflicting updates, have a globally determined resolution mechanism (for example, using NTP, simply let the "most recent" update win).

### Program consistency

$P$ is a monotonic problem if for any input sets $S$ and $T$, $P(S) \subseteq P(T)$.
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. Example: filling a shopping cart. NOTE: *In all cases, we are avoiding global synchronization.*

## Continuous Consistency

### We can actually talk about a degree of consistency

- replicas may differ in their numerical value
- replicas may differ in their relative staleness
- there may be differences regarding (number and order) of performed update operations

### Conit

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

# Example: Conit



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)

## Conit (contains the variables $g$, $p$, and $d$)

- Each replica has a vector clock: ([known] time @ A, [known] time @ B)
- $B$ sends $A$ operation [$\langle 5, B \rangle : g \leftarrow d + 45$]; $A$ has made this operation permanent (cannot be rolled back)

# Example: Conit



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)

## Conit (contains the variables $g$, $p$, and $d$)

- $A$ has three pending operations $\Rightarrow$ order deviation = 3
- $A$ missed two operations from $B$; max diff is $70 + 412$ units $\Rightarrow (2, 482)$

# Client-centric consistency models

## Consistency for mobile users

### Example

Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.
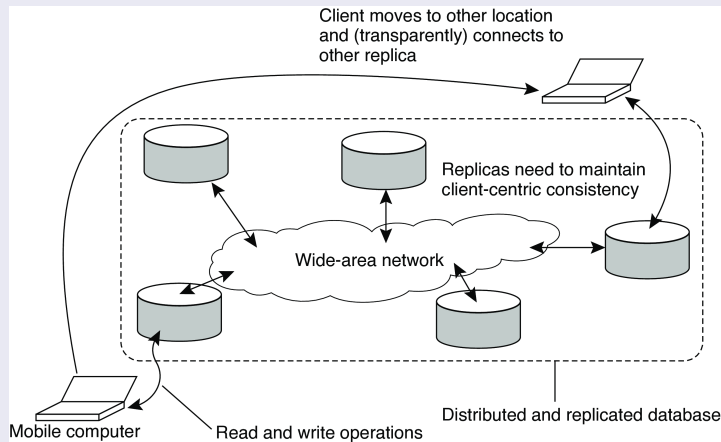
- At location $A$ you access the database doing reads and updates.
- At location $B$ you continue your work, but unless you access the same server as the one at location $A$, you may detect inconsistencies:
  - your updates at $A$ may not have yet been propagated to $B$
  - you may be reading newer entries 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.

## Basic architecture

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



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: notation

## 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 $x_i$.

# Monotonic reads

### 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.

## 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.

$$L_1 \xrightarrow{\quad W_1(x_1) \qquad\qquad R_1(x_1) \quad}$$
$$L_2 \xrightarrow{\quad\quad W_2(x_1;x_2) \qquad\quad R_1(x_2) \quad}$$
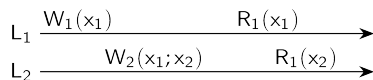
A monotonic-read consistent data store

$$L_1 \xrightarrow{\quad W_1(x_1) \qquad\qquad R_1(x_1) \quad}$$
$$L_2 \xrightarrow{\quad\quad W_2(x_1|x_2) \qquad\quad R_1(x_2) \quad}$$
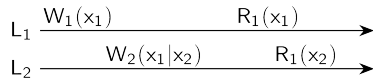
A data store that does not provide monotonic reads

## Monotonic writes

### Example

Updating a program at server $S_2$, and ensuring that all components on which compilation and linking depends, are also placed at $S_2$.

### Example

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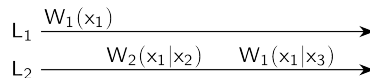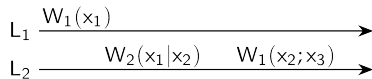
## Monotonic writes

### Definition

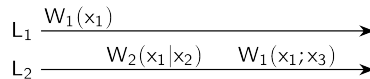A write operation by a process on a data item $x$ is completed before any successive write operation on $x$ by the same process.

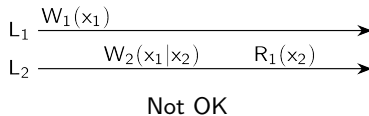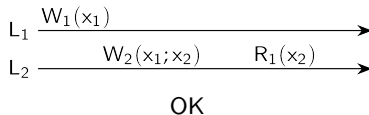$$L_1 \xrightarrow{\quad W_1(x_1) \quad}$$
$$L_2 \xrightarrow{\quad W_2(x_1;x_2) \qquad W_1(x_2;x_3) \quad}$$

OK

$$L_1 \xrightarrow{\quad W_1(x_1) \quad}$$
$$L_2 \xrightarrow{\quad W_2(x_1|x_2) \qquad W_1(x_1|x_3) \quad}$$

Not OK

$$L_1 \xrightarrow{\quad W_1(x_1) \quad}$$
$$L_2 \xrightarrow{\quad W_2(x_1|x_2) \qquad W_1(x_2;x_3) \quad}$$

Not OK

$$L_1 \xrightarrow{\quad W_1(x_1) \quad}$$
$$L_2 \xrightarrow{\quad W_2(x_1|x_2) \qquad W_1(x_1;x_3) \quad}$$
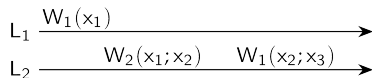
OK

## Read your writes

### Definition

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.
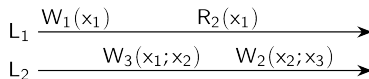
$$L_1 \xrightarrow{\quad W_1(x_1) \quad}$$

$$L_2 \xrightarrow{\quad W_2(x_1;x_2) \qquad R_1(x_2) \quad}$$

OK

$$L_1 \xrightarrow{\quad W_1(x_1) \quad}$$

$$L_2 \xrightarrow{\quad W_2(x_1|x_2) \qquad R_1(x_2) \quad}$$

Not OK

### Example

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

## Writes follow reads

### Definition

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.

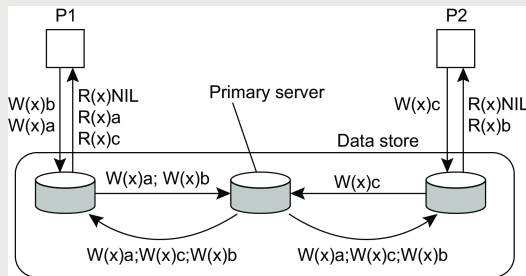$$L_1 \xrightarrow{\quad W_1(x_1) \qquad\qquad R_2(x_1) \quad}$$
$$L_2 \xrightarrow{\quad\qquad W_3(x_1;x_2) \qquad W_2(x_2;x_3) \quad}$$

OK

$$L_1 \xrightarrow{\quad W_1(x_1) \qquad\qquad R_2(x_1) \quad}$$
$$L_2 \xrightarrow{\quad\qquad W_3(x_1|x_2) \qquad W_2(x_2|x_3) \quad}$$
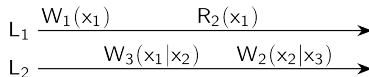
Not OK

### Example

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

# Example: ZooKeeper consistency

### Yet another model?

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

### Take a naive example

# Replica management

## Replica placement

### 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.

# Content replication

## 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)

# Content replication

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



Permanent replicas

Server-initiated replicas

Client-initiated replicas

Clients

→ Server-initiated replication
--→ Client-initiated replication
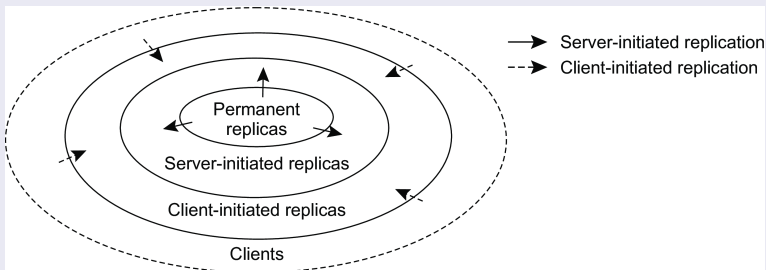
## Server-initiated replicas

### Counting access requests from different clients
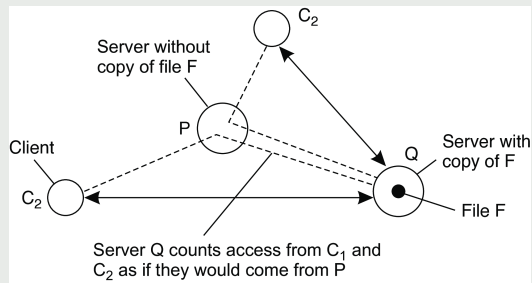


- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold $D \Rightarrow$ drop file
- Number of accesses exceeds threshold $R \Rightarrow$ replicate file
- Number of access between $D$ and $R \Rightarrow$ migrate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

# 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.

## Content distribution: client/server system

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* | | |

## 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.
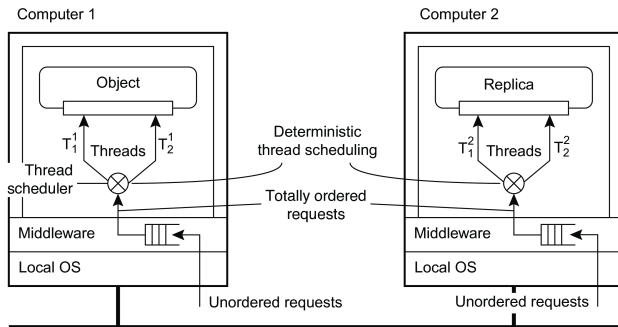
### Make lease expiration time adaptive

- 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
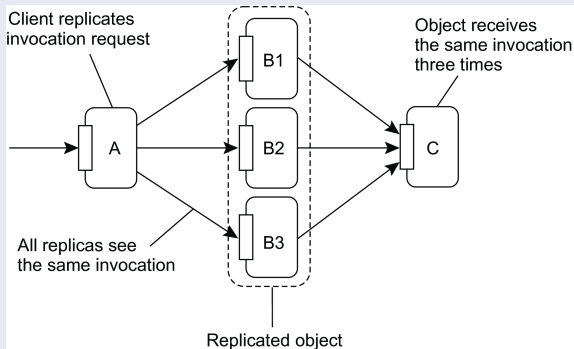
### Question

Why are we doing all this?

# Managing replicated objects

- Prevent concurrent execution of multiple invocations on the same object: access to the internal data of an object has to be serialized. Using local locking mechanisms are sufficient.

- Ensure that all changes to the replicated state of the object are the same: no two independent method invocations take place on different replicas at the same time: we need deterministic thread scheduling.
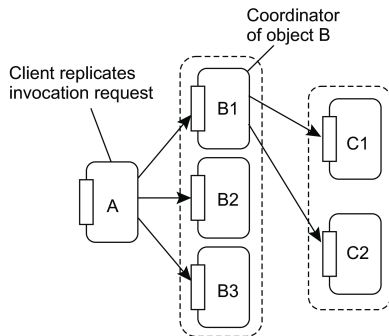
# Replicated-object invocations

## Problem when invoking a replicated object

Client replicates
invocation request

Object receives
the same invocation
three times

All replicas see
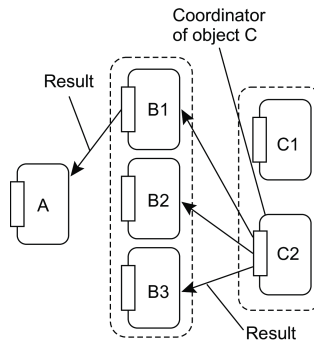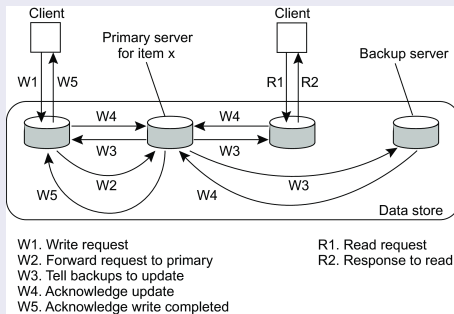the same invocation

B1

B2

B3

A

C

Replicated object

# Replicated-object invocations



Forwarding a request

Returning the reply

Consistency protocols
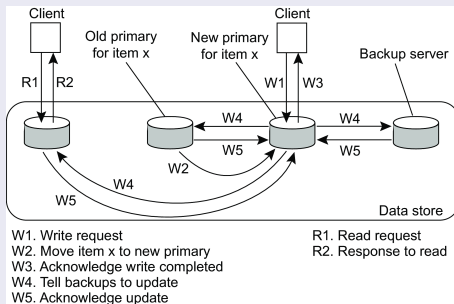
# Primary-based protocols

## Primary-backup protocol



Client

Primary server
for item x

Client

Backup server

W1  W5

W4

W4

R1  R2

W3

W3

W2

W3

W5

W4

Data store

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

## 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.

# Primary-based protocols

## Primary-backup protocol with local writes



Client

Old primary
for item x

New primary
for item x

Client

Backup server

R1 | R2

W1 | W3

W4

W5

W4

W5

W2

W4

W5

Data store

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).
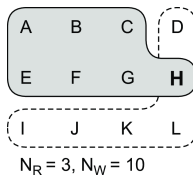
## Replicated-write protocols

### Quorum-based protocols

Assume $N$ replicas. Ensure that each operation is carried out in such a way that a majority vote is established: distinguish read quorum $N_R$ and write quorum $N_W$. Ensure:

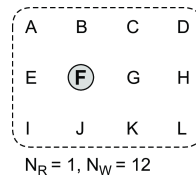1. $N_R + N_W > N$ (prevent read-write conflicts)
2. $N_W > N/2$ (prevent write-write conflicts)



$N_R = 3, N_W = 10$

Correct

$N_R = 3, N_W = 10$

Write-write conflict

$N_R = 1, N_W = 12$
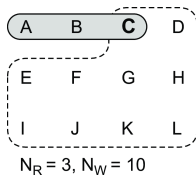
Correct (ROWA)

## Continuous consistency: Numerical errors

### Principal operation

- Every server $S_i$ has a log, denoted as $L_i$.

- Consider a data item $x$ and let $val(W)$ denote the numerical change in its value after a write operation $W$. Assume that

$$\forall W : val(W) > 0$$

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

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

## Continuous consistency: Numerical errors

> ### Note
>
> Actual value $v(t)$ of $x$:
>
> $$v(t) = v_{init} + \sum_{k=1}^{N} TW[k, k]$$
>
> value $v_i$ of $x$ at server $S_i$:
>
> $$v_i = v_{init} + \sum_{k=1}^{N} TW[i, k]$$

# Continuous consistency: Numerical errors

### Problem

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

### Approach

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

### Note

$$0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j]$$

# Continuous consistency: Numerical errors

## Solution

$S_k$ sends operations from its log to $S_i$ when it sees that $TW_k[i,k]$ is getting too far from $TW[k,k]$, in particular, when

$$TW[k,k] - TW_k[i,k] > \delta_i/(N-1)$$

## Question

To what extent are we being pessimistic here: where does $\delta_i/(N-1)$ come from?

## Note

Staleness can be done analogously, by essentially keeping track of what has been seen last from $S_i$ (see book).

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

### Monotonic-write consistency

When client $C$ wants to write at server $S$, $C$ passes its write 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.

# Implementing client-centric consistency

### Read-your-writes consistency

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

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.

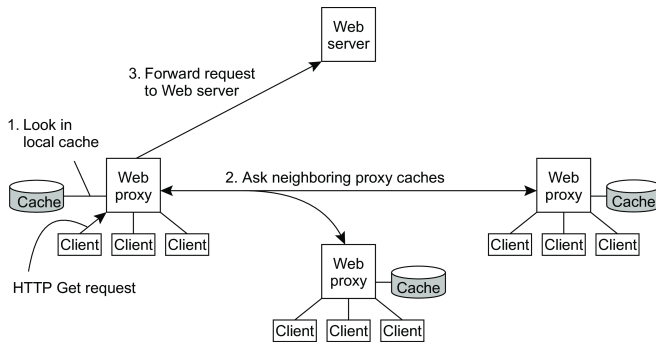# Example: Caching and replication in the Web

## Example: replication in the Web

### Client-side caches

- In the browser

- At a client's site, notably through a Web proxy

### Caches at ISPs

Internet Service Providers also place caches to (1) reduce cross-ISP traffic and (2) improve client-side performance. May get nasty when a request needs to pass many ISPs.

# Cooperative caching

## 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.

## Alternatives for caching and replication
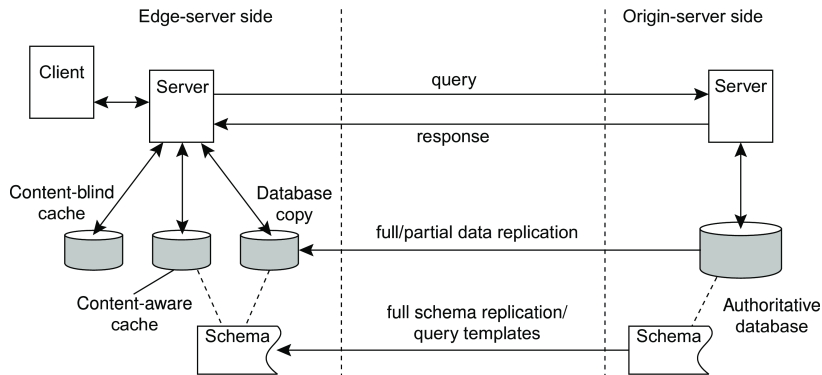


- Database copy: the edge has the same as the origin server
- Content-aware cache: check if a (normal query) can be answered with cached data. Requires that the server knows about which data is cached at the edge.
- Content-blind cache: store a query, and its result. When the exact same query is issued again, return the result from the cache.

# Summary

## Summary

In this section on *Consistency and Replication*, we discussed the following topics

- Data-centric consistency models
- Client-centric consistency models
- Replica Management
- Consistency Protocols
- Caching Example