Distributed Systems

Consistency, replication, and fault tolerance

TDTS04 – Computer Networks and Distributed Systems

Carl Magnus Bruhner, ADIT/IDA

LINKÖPING UNIVERSITY

Based on slides by M. van Steen and A. S. Tanenbaum,
Distributed Systems, 4th ed., distributed-systems.net

Consistency and replication in distributed systems

LINKÖPING
UNIVERSITY

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

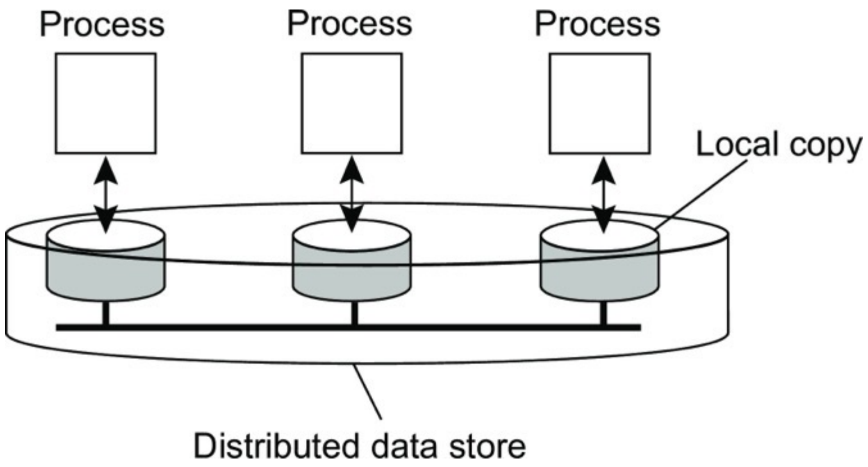
Data-centric consistency models

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:



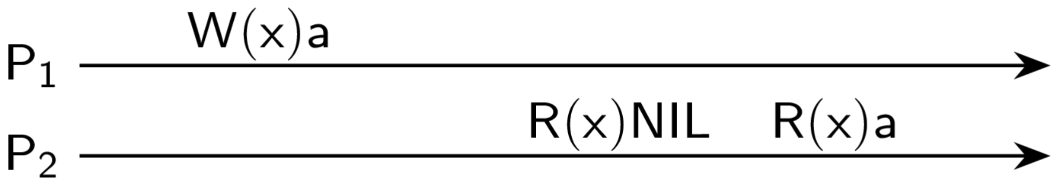
Some notations

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

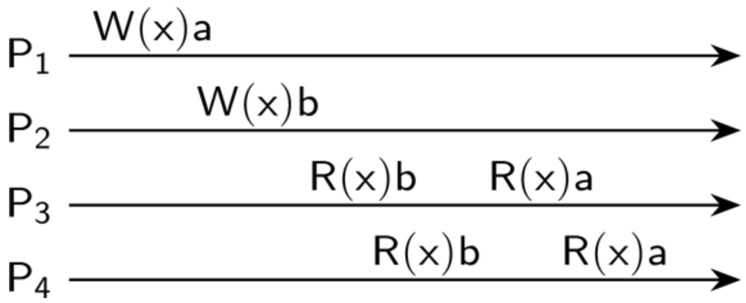
We omit the index when possible and draw according to time (x-axis):



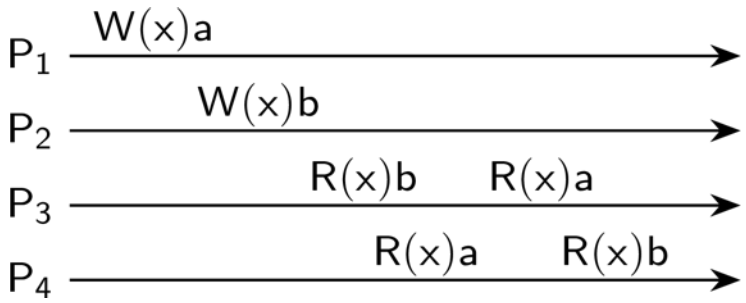
Sequential consistency

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.



A sequentially consistent data store



A data store that is not sequentially consistent

Example

Three concurrent processes (initial values: 0)

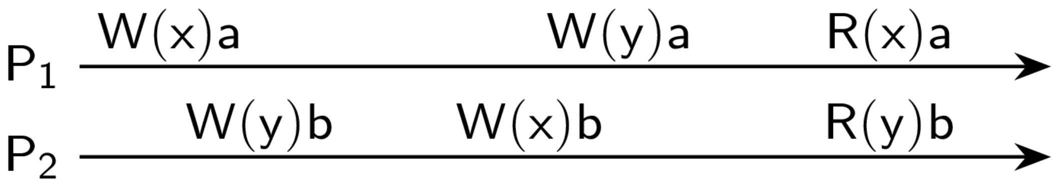
Process P_1	Process P_2	Process P_3
`x ← 1;`	`y ← 1;`	`z ← 1;`
`print(y,z);`	`print(x,z);`	`print(x,y);`

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)

How tricky can it get?

Seemingly okay



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

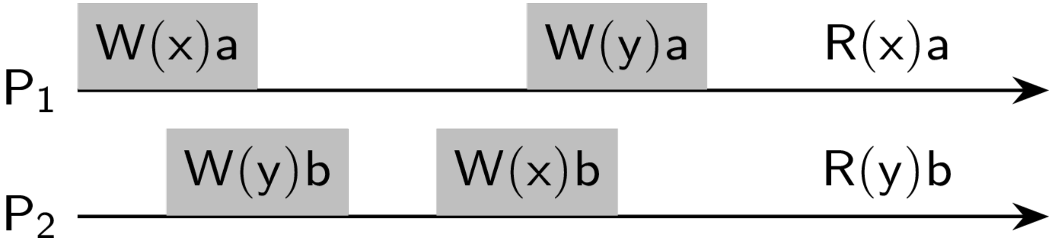
Possible ordering of operations	**Result**	
$W_1(x)a$; $W_1(y)a$; $W_2(y)b$; $W_2(x)b$	$R_1(x)b$	$R_2(y)b$
$W_1(x)a$; $W_2(y)b$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a$; $W_2(y)b$; $W_2(x)b$; $W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$; $W_1(x)a$; $W_1(y)a$; $W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b$;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

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)



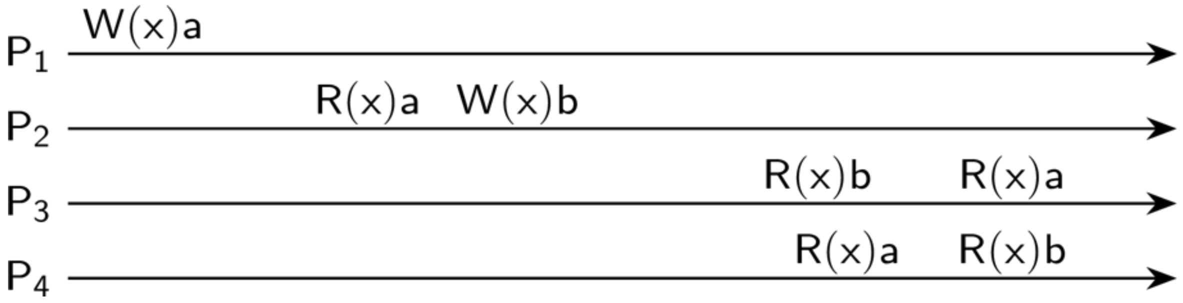
With better results

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

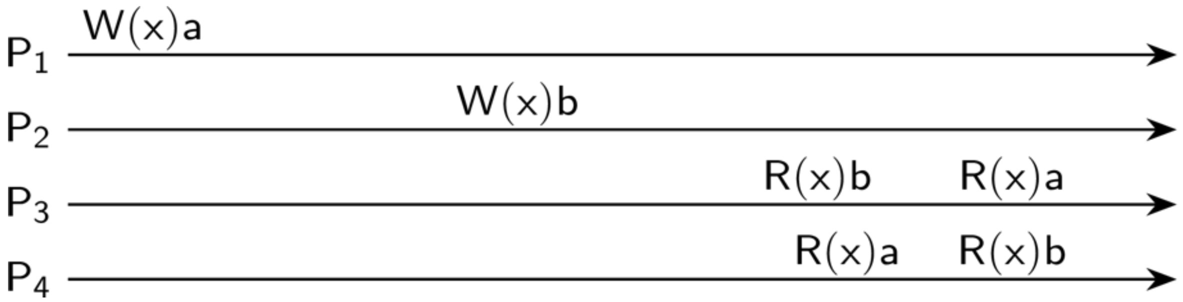
Causal consistency

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.



A violation of a causally-consistent store



A correct sequence of events in a causally-consistent store

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).

Strong eventual consistency

Basic idea: if there are conflicting updates, have a globally determined resolution mechanism (e.g., in NTP, letting 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.

Important observation

In all cases, we are avoiding global synchronization.

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.

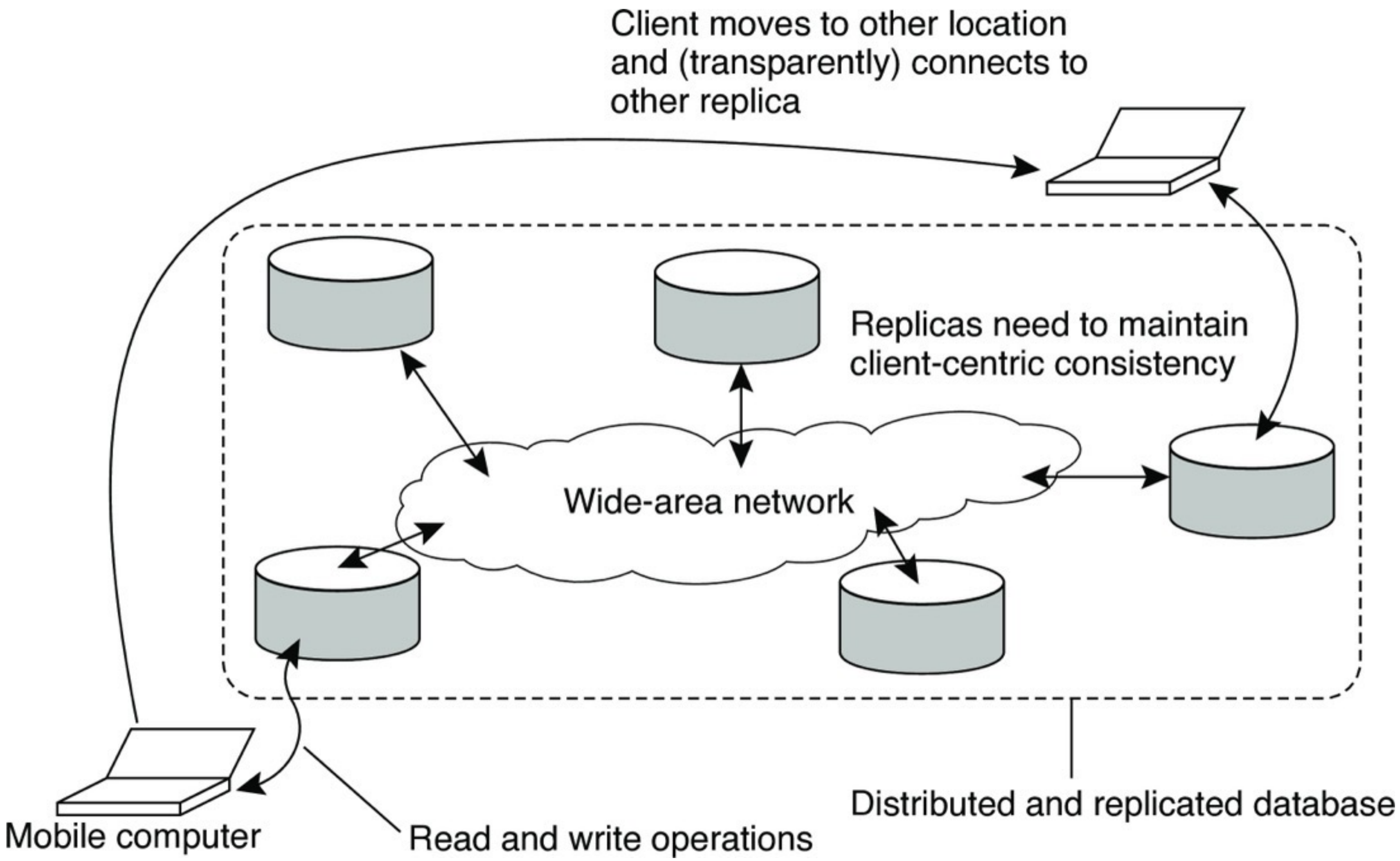
- At location *A* you access the database doing reads and updates.
- At location *B* you continue your work, but unless you access the same server as the one at location *A*, you may detect inconsistencies:
  - your updates at *A* may not have yet been propagated to *B*
  - you may be reading newer entries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

- Monotonic reads
  - *If a process reads the value of a data item ×, any successive read operation on × by that process will always return that same value or a more recent value.*
- Monotonic writes
  - *A write operation by a process on a data item × is completed before any successive write operation on × by the same process.*
- Read your writes
  - *The effect of a write operation by a process on data item × will always be seen by a successive read operation on × by the same process.*
- Writes follow reads
  - *A write operation by a process on a data item × following a previous read operation on × by the same process is guaranteed to take place on the same or a more recent value of × that was read.*

Content replication

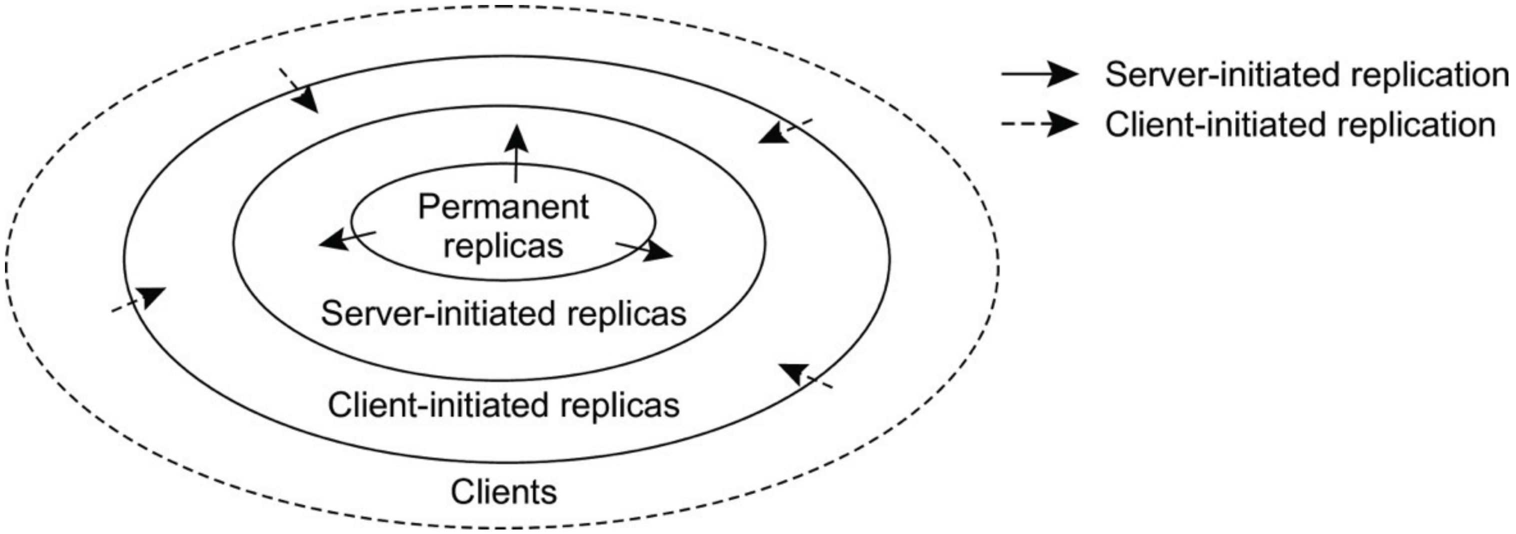
Distinguish different processes

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 (CDN)
- 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



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

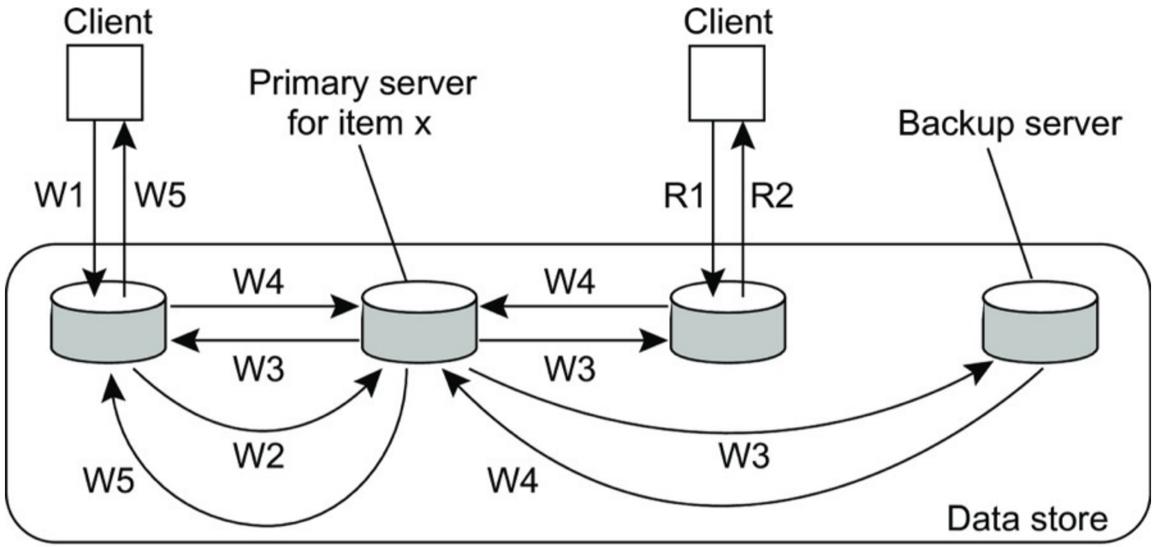
A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems

- 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
State at server	List of client caches	None
Messages to be exchanged	Update (and possibly fetch update)	Poll and update
Response time at the client	Immediate (or fetch-update time)	Fetch-update time

Primary-based protocols

Primary-backup protocol



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

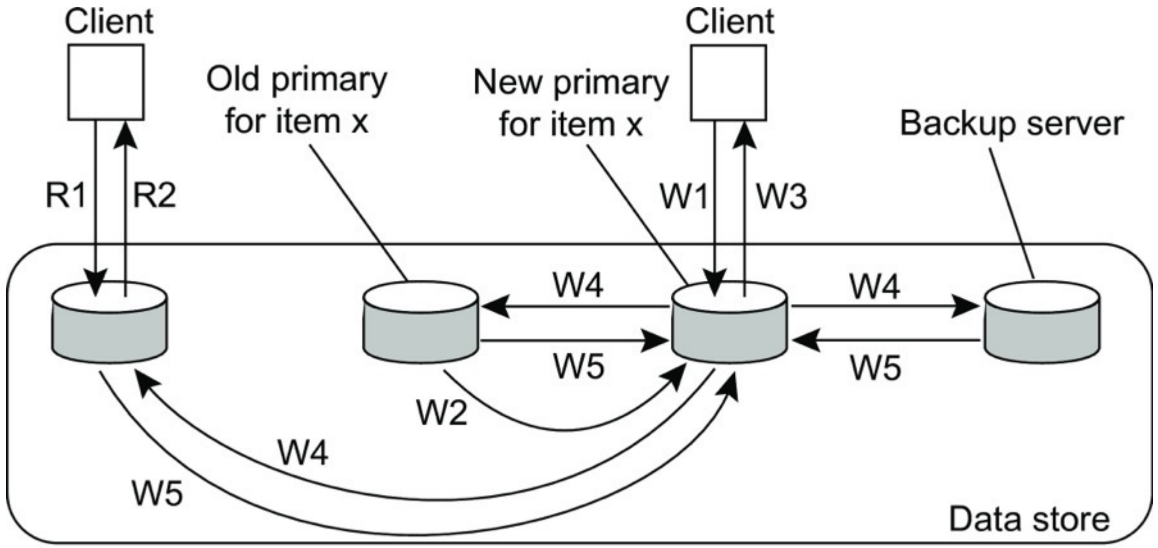
R1. Read request
R2. Response to read

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



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).

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.

Web-cache consistency

How to guarantee freshness?

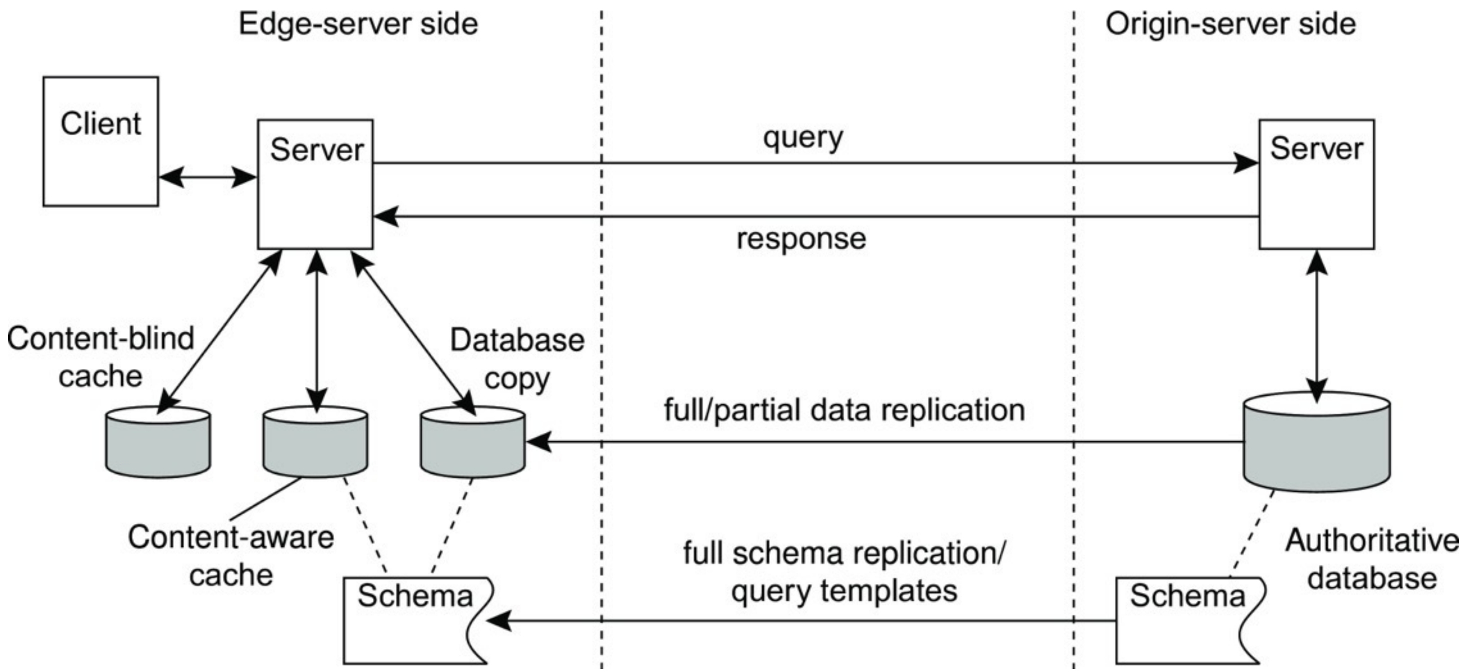
To prevent that stale information is returned to a client:

- 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



- Content-blind cache: store a query, and its result. When the exact same query is issued again, return the result from the cache.
- 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.
- Database copy: the edge has the same as the origin server

Fault tolerance in distributed systems

Dependability

Basics

A component provides services to clients. To provide services, the component may require the services from other components ⇒ a component may depend on some other component.

Specifically

A component C depends on $C*$ if the correctness of C's behavior depends on the correctness of $C*$'s behavior. (Components are processes or channels.)

Requirements related to dependability

Requirement	**Description**
Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	How easy can a failed system be repaired

Reliability versus availability

Reliability $R(t)$ of component C

Conditional probability that C has been functioning correctly during $[0, t)$ given C was functioning correctly at time $T = 0$.

Traditional metrics

- Mean Time To Failure (*MTTF*): The average time until a component fails.
- Mean Time To Repair (*MTTR*): The average time needed to repair a component.
- Mean Time Between Failures (*MTBF*): Simply $MTTF + MTTR$.

Reliability versus availability

Availability $A(t)$ of component C

Average fraction of time that C has been up-and-running in interval $[0, t)$.

- Long-term availability A: $A(\infty)$

  Note: $A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF+MTTR}$

Observation

Reliability and availability make sense only if we have an accurate notion of what a failure actually is.

Terminology

Failure, error, fault

Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

Terminology

Handling faults

Term	**Description**	**Example**
Fault prevention	Prevent the occurrence of a fault	Don’t hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

Failure models

Types of failures

Type	Description of server's behavior
Crash failure	Halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	Fails to respond to incoming requests Fails to receive incoming messages Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure *Value failure* *State-transition failure*	Response is incorrect The value of the response is wrong Deviates from the correct flow of control
Arbitrary failure	May produce arbitrary responses at arbitrary times

Halting failures

Scenario

C no longer perceives any activity from *C** — a halting failure? Distinguishing between a crash or omission/timing failure may be impossible.

Asynchronous versus synchronous systems

- Asynchronous system: no assumptions about process execution speeds or message delivery times → cannot reliably detect crash failures.
- Synchronous system: process execution speeds and message delivery times are bounded → we can reliably detect omission and timing failures.
- In practice we have partially synchronous systems: most of the time, we can assume the system to be synchronous, yet there is no bound on the time that a system is asynchronous → can normally reliably detect crash failures.

Redundancy for failure masking

Types of redundancy

- Information redundancy: Add extra bits to data units so that errors can recovered when bits are garbled.
- Time redundancy: Design a system such that an action can be performed again if anything went wrong. Typically used when faults are transient or intermittent.
- Physical redundancy: add equipment or processes in order to allow one or more components to fail. This type is extensively used in distributed systems.

Consensus

Prerequisite

In a fault-tolerant process group, each nonfaulty process executes the same commands, and in the same order, as every other nonfaulty process.

Reformulation

Nonfaulty group members need to reach consensus on which command to execute next.

Examples: Flooding-based consensus, Raft, Paxos (details in book)

Failure detection

Issue

How can we reliably detect that a process has actually crashed?

General model

- Each process is equipped with a failure detection module
- A process *P* probes another process *Q* for a reaction
- If *Q* reacts: *Q* is considered to be alive (by *P*)
- If *Q* does not react with *t* time units: *Q* is suspected to have crashed

Observation for a synchronous system

a suspected crash ≡ a known crash

Practical failure detection

Implementation

- If *P* did not receive heartbeat from Q within time *t*: *P* suspects Q.
- If *Q* later sends a message (which is received by *P*):
  - *P* stops suspecting *Q*
  - *P* increases the timeout value *t*
- Note: if *Q* did crash, *P* will keep suspecting *Q*.

Reliable remote procedure calls

What can go wrong?

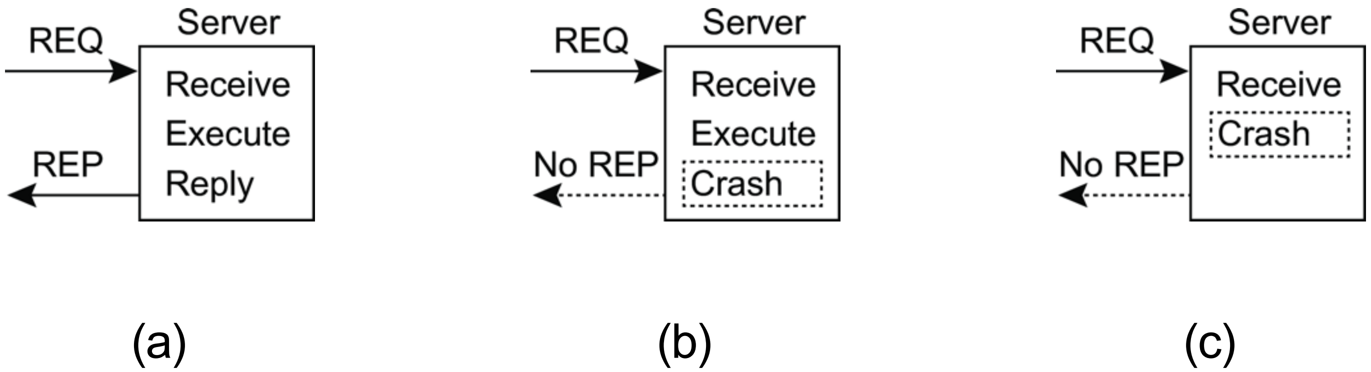
1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

Two “easy” solutions

1: (cannot locate server): just report back to client

2: (request was lost): just resend message

Reliable RPC: server crash



Problem

Where (a) is the normal case, situations (b) and (c) require different solutions. However, we don’t know what happened. Two approaches:

- At-least-once-semantics: The server guarantees it will carry out an operation at least once, no matter what.
- At-most-once-semantics: The server guarantees it will carry out an operation at most once (but possibly not at all).

Reliable RPC: lost reply messages

The real issue

What the client notices, is that it is not getting an answer. However, it cannot decide whether this is caused by a lost request, a crashed server, or a lost response.

Partial solution

Design the server such that its operations are idempotent: repeating the same operation is the same as carrying it out exactly once:

- pure read operations
- strict overwrite operations

Many operations are inherently nonidempotent, such as many banking transactions.

Reliable RPC: client crash

Problem

The server is doing work and holding resources for nothing (called doing an orphan computation).

Solution

- Orphan is killed (or rolled back) by the client when it recovers
- Client broadcasts new epoch number when recovering ⇒ server kills client's orphans
- Require computations to complete in a *T* time units. Old ones are simply removed.

Recovery: Background

Essence

When a failure occurs, we need to bring the system into an error-free state:

- Forward error recovery: Find a new state from which the system can continue operation
- Backward error recovery: Bring the system back into a previous error-free state

Practice

Use backward error recovery, requiring that we establish recovery points

Observation

Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover

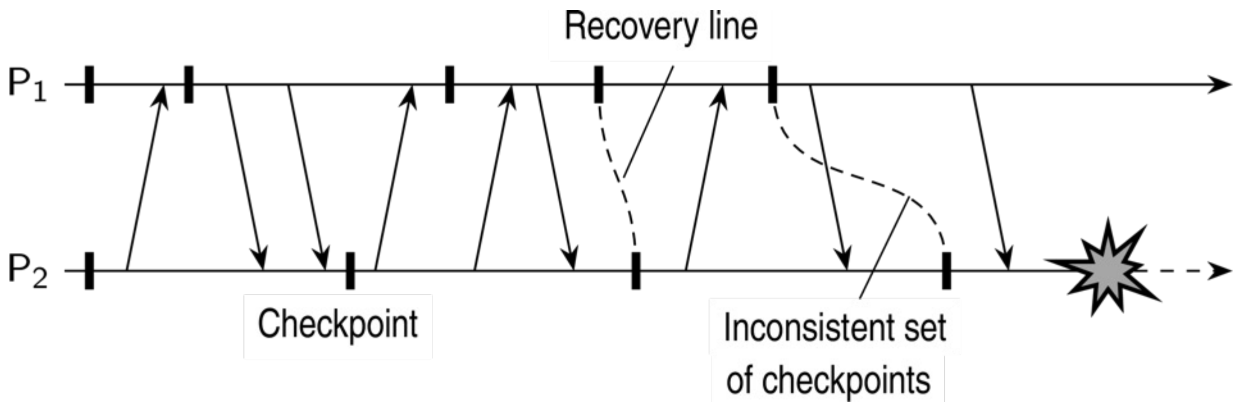
Consistent recovery state

Requirement

Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line

Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.



Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a checkpoint request message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a checkpoint done message to allow all processes to continue

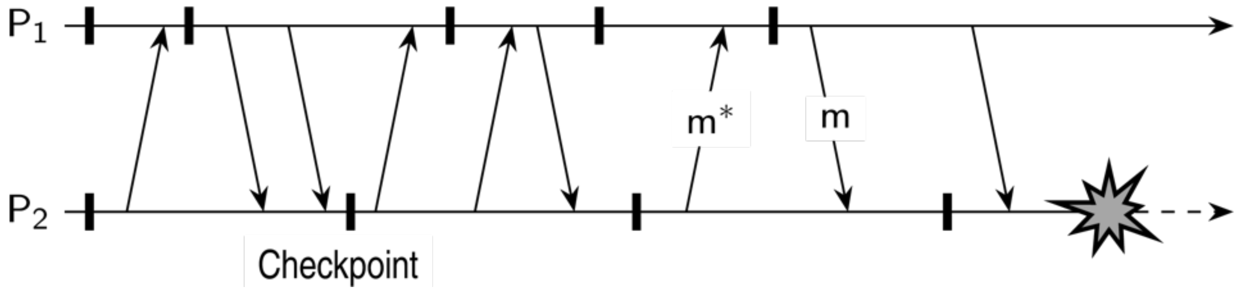
Observation

It is possible to consider only those processes that depend on the recovery of the coordinator, and ignore the rest

Independent checkpointing and cascaded rollback

Observation

If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time. We have a so-called cascaded rollback or domino effect.



Exam

Some guidance for the exam

- Questions are now published on the course webpage.
- To best answer the questions, make sure to read up on each in the book: Distributed Systems (M. van Steen & A. S. Tanenbaum)
  Available for free at: https://www.distributed-systems.net/index.php/books/ds4/
- Slides might only give a shallow idea of each concept (but enough to pass).
- Questions can be answered by using terminology and examples from the slides/books and by connecting to your knowledge of computer networks.

It’s a wrap!

Questions?

Questions/feedback: carl.magnus.bruhner@liu.se

LINKÖPING
UNIVERSITY

Extras

(Not part of exam.)

MapReduce

- MapReduce is an example of a programming model used in parallel and distributed algorithms, working with large datasets (big data).
- For more, courses on Big Data, Machine Learning, etc. are recommended.

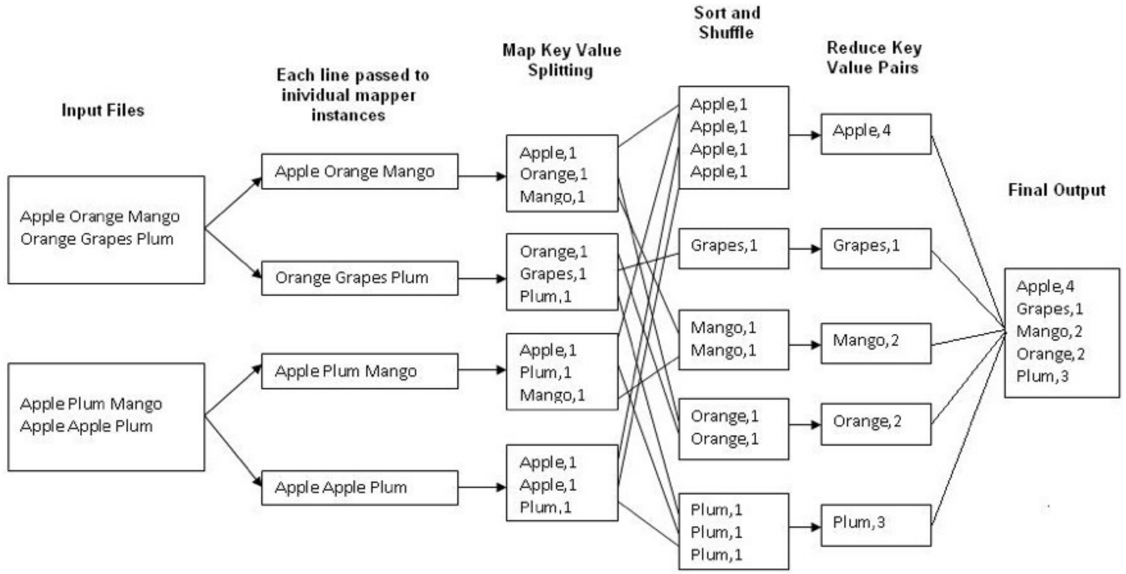


Illustration: CC BY-SA 4.0 Magnai17/Wikimedia