Distributed Algorithms

Consistency & Replication

Contents

Contents

Introduction

Definition (Availability)

The probability that a system will provide its required service, or the ratio of the total time a system is capable of being used during a given interval to the length of the interval:

$$A = \frac{E[uptime]}{E[uptime + downtime]}$$

Example

- One single server
- On average, crashes once per week (MTBF: $10.080'$)
- Two minutes to reboot (MTBR: $2'$)

$$A = \frac{10080}{10080 + 2} = 0.\underline{9998}$$

Introduction

Definition (Availability)

The probability that a system will provide its required service, or the ratio of the total time a system is capable of being used during a given interval to the length of the interval:

$$A = \frac{E[uptime]}{E[uptime + downtime]}$$

Example

- Ten servers
- MTBF, MTBR as before
- All needed at the same time to perform the service

$$p_f = \frac{2}{10082}$$

$$A = (1 - p_f)^{10} = 0.\underline{998}$$

Introduction

Definition (Availability)

The probability that a system will provide its required service, or the ratio of the total time a system is capable of being used during a given interval to the length of the interval:

$$A = \frac{E[uptime]}{E[uptime + downtime]}$$

Example

- Ten servers
- MTBF, MTBR as before
- One replica needed to perform the service

$$p_f = \frac{2}{10082}$$

$$A = 1 - (p_f)^{10} = 1 - 10^{-38}$$

Definition (Availability)

The probability that a system will provide its required service, or the ratio of the total time a system is capable of being used during a given interval to the length of the interval:

$$A = \frac{E[uptime]}{E[uptime + downtime]}$$

Example

- Ten servers
- MTBF, MTBR as before
- One replica needed to perform the service

$$p_f = \frac{2}{10082}$$
$$A = 1 - (p_f)^{10} = 1 - 10^{-38}$$

Five-nines availability: 99.999%, means 5 minutes per year.

Replication

How to increase availability:

- Avoid single point of failures
- Use replication (time/space)

Replication in space:

- Run parallel copies
- Vote on replica output
- High-availability, high-cost

Replication in time:

- When a replica fails, restart it (or replace it)
- Lower maintenance, lower availability

Replication

Replication advantages:

- Replicating a service increases its availability
- Performance benefits:
  - Geographical co-location
  - Load-balancing
  - No bottlenecks

Replication drawbacks:

- Trade-off between availability and consistency
- Transparent replication is difficult

Consistency problem

The consistency problem:

- Whenever a copy is modified, that copy becomes different from the rest
- Modifications have to be carried out on all copies to ensure consistency

Conflicting operations - from the world of transactions:

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

Consistency problem

The goal

We generally need to ensure that all conflicting operations are done in the same order everywhere

The problem

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

The solution

Weaken consistency requirements so that hopefully global synchronization can be avoided

Consistency example

Example (Flight reservation database)

- At 9.36, all seats of flight 48 are booked
- At 9.37, Jane cancel its reservation on flight 48
- At 9.38, Michael tries to reserve a seat on flight 48
  - the answer is fully booked
- At 9.39, George tries to reserve a set on flight 48
  - the seat is granted

Contents

Consistency models

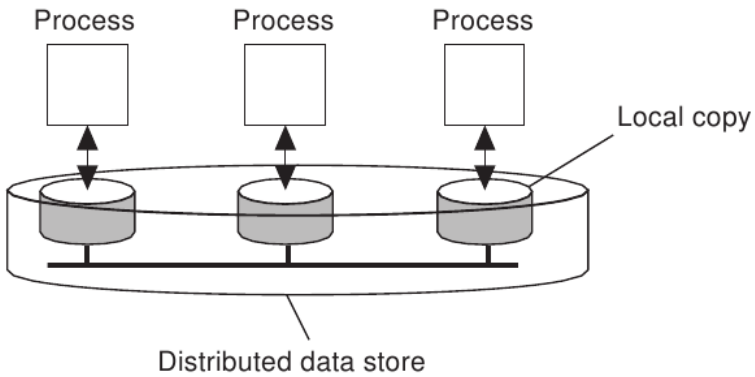
Definition (Consistency model)

A contract between a distributed data store and a set of processes, which specifies what the results of read/write operations are in the presence of concurrency

Definition (Distributed data store)

A distributed collection of storage entities accessible to clients

- Distributed database, file system
- Shared memory in a parallel system



Consistency models

Werner Vogels

Whether or not inconsistencies are acceptable depends on the client application. In all cases the developer must be aware that consistency guarantees are provided by the storage systems and must be taken into account when developing applications.

W. Vogels. Eventual consistent.
Comm. of the ACM, 52(1):40–44, 2009

Amazon's vice-president and Chief Scientific Officer

Consistency models

Strong consistency models

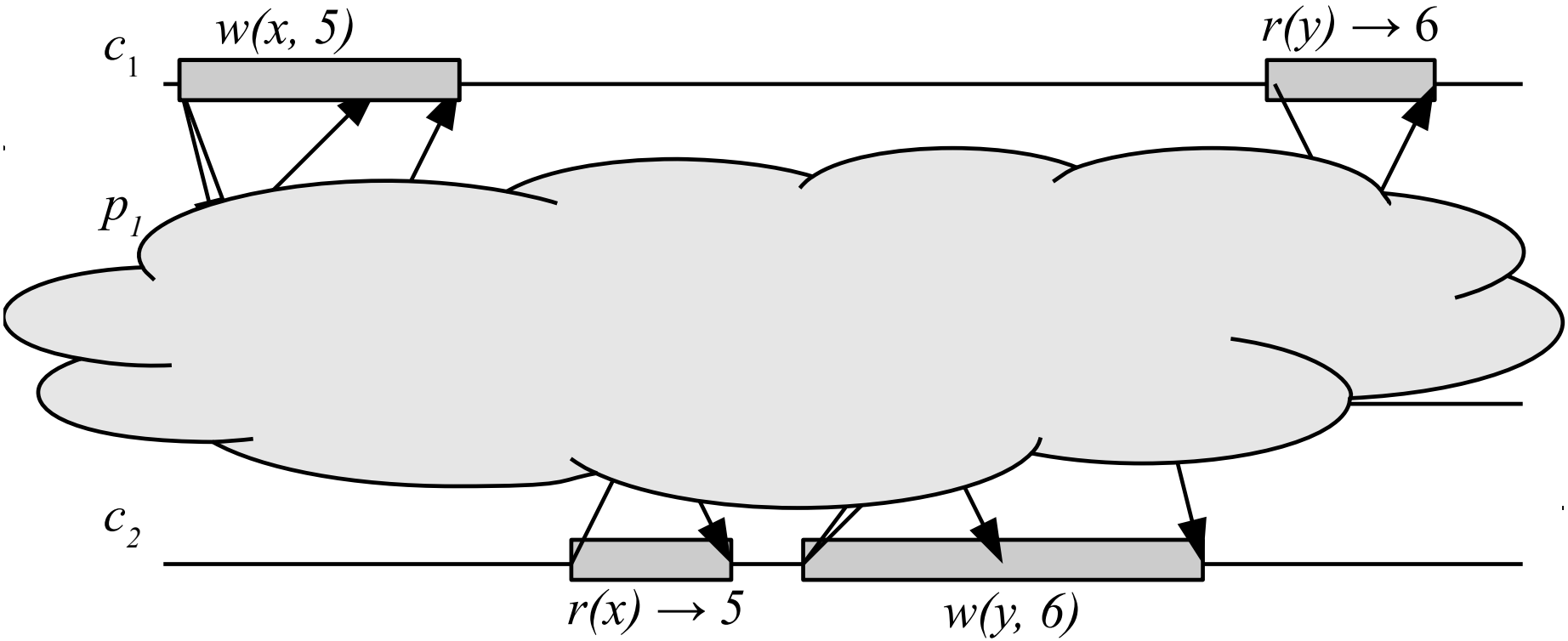
- Strict consistency
- Linearizability
- Sequential consistency

Weak consistency models

- Eventual consistency
- Client-centric consistency models
  - Read-after-read (monotonic read)
  - Read-after-write (read your writes)
- Causal consistency

Notation

- Write operation: $w(x, v)$
- Read operation: $r(x) \rightarrow v$



Strict consistency

Definition (Strict consistency)

A read operation must return the result of the latest write operation which occurred on the data item

Implementation:

- Only possible with a global, perfectly synchronized clock
- Only possible if all writes instantaneously visible to all

It makes sense, though:

- it is the model of uniprocessor systems!

Sequential Consistency

Definition (Sequential Consistency, Lamport, 1978)

An execution E is sequential consistent provided that there exists a sequence H such that

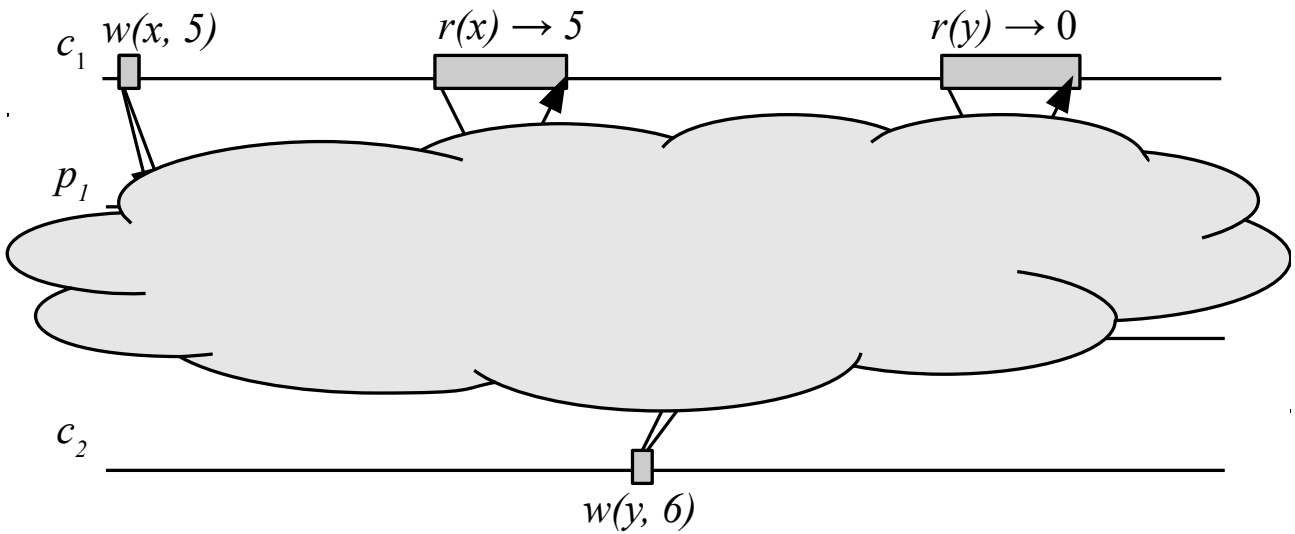
SC1 H contains exactly the same operations that occur in E, each paired with the return value received in E

SC2 H is a legal history of the sequential data type that is replicated

SC3 The total order of operations in H is compatible with the client partial order $<_{E,c}$

- $o_1 <_{E,c} o_2$ means that the o_1 and o_2 occur at the same client and that o_1 returns before o_2 is invoked
- The client order $<_{E,c}$ is a partial order

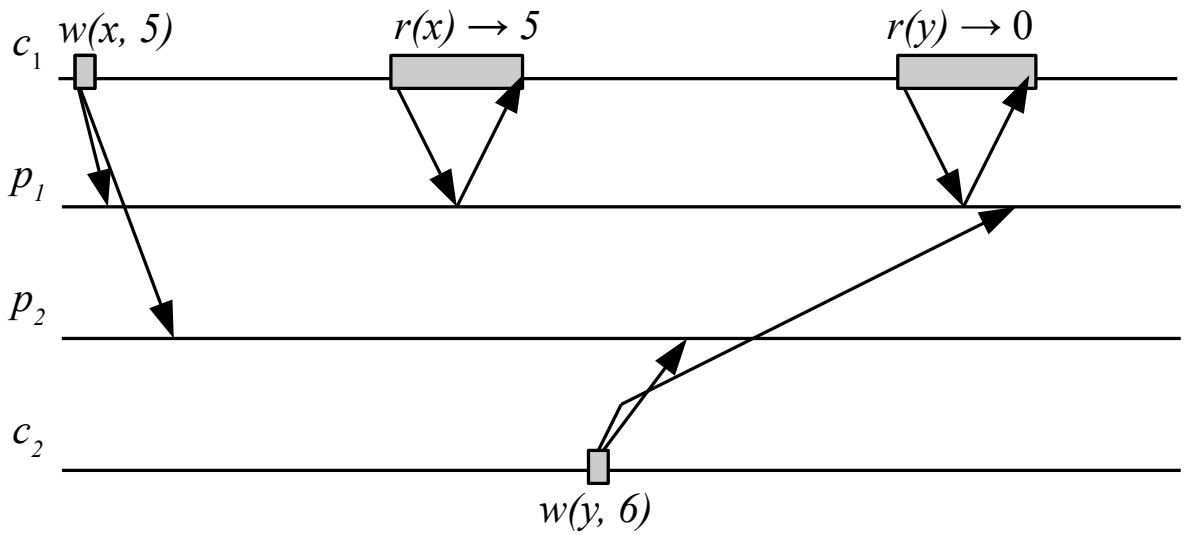
Sequential Consistency



Example

- Is the execution above sequentially consistent?

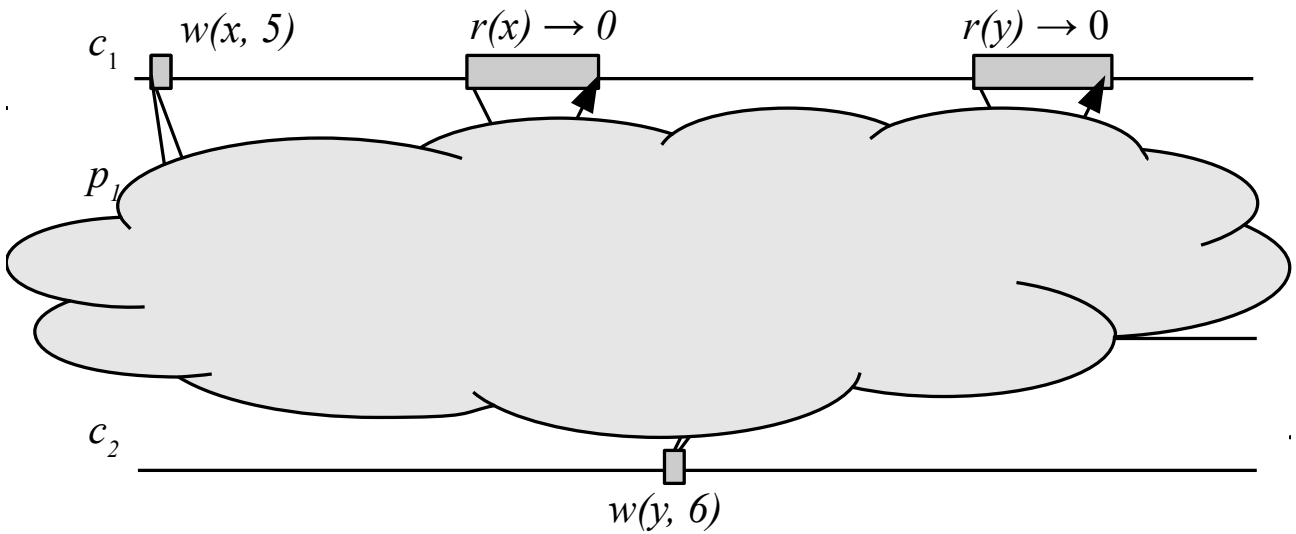
Sequential Consistency



Example

- Is the execution above sequentially consistent? YES

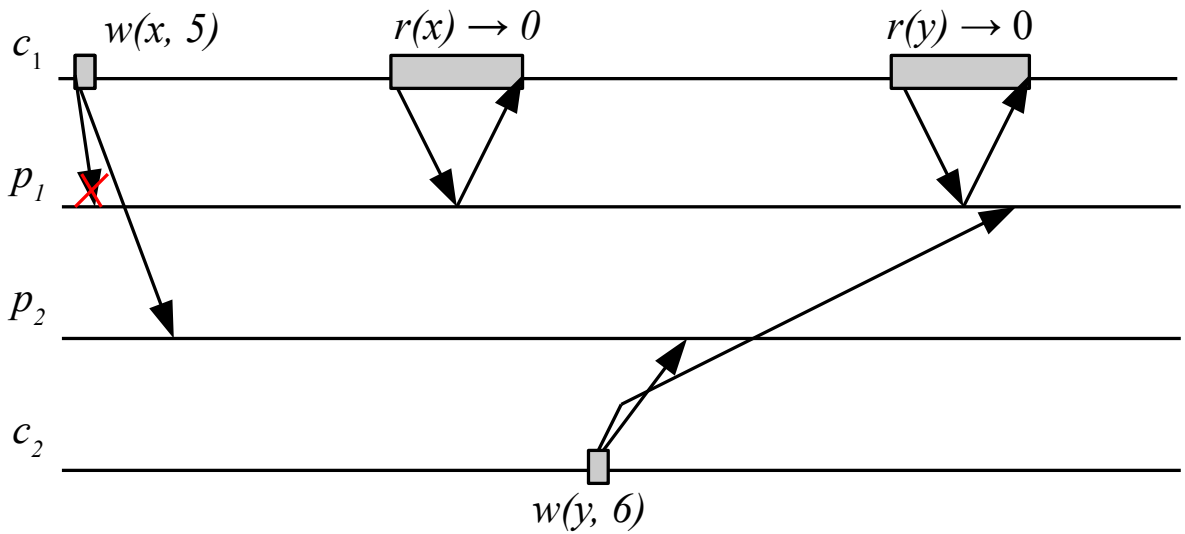
Sequential Consistency



Example

- Is the execution above sequentially consistent?

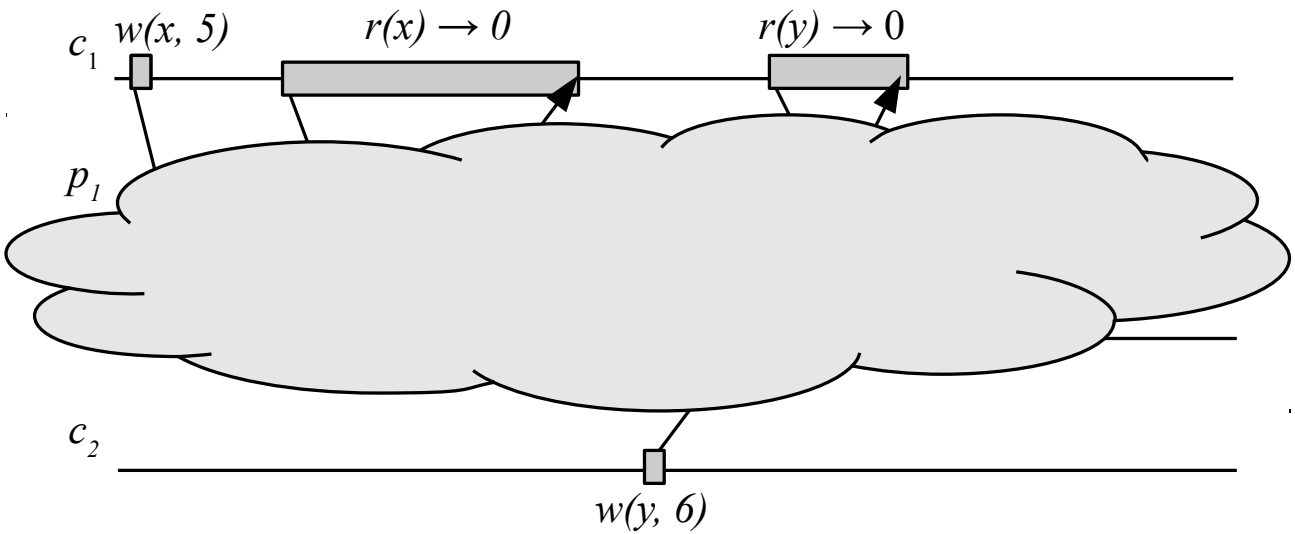
Sequential Consistency



Example

- Is the execution above sequentially consistent? NO

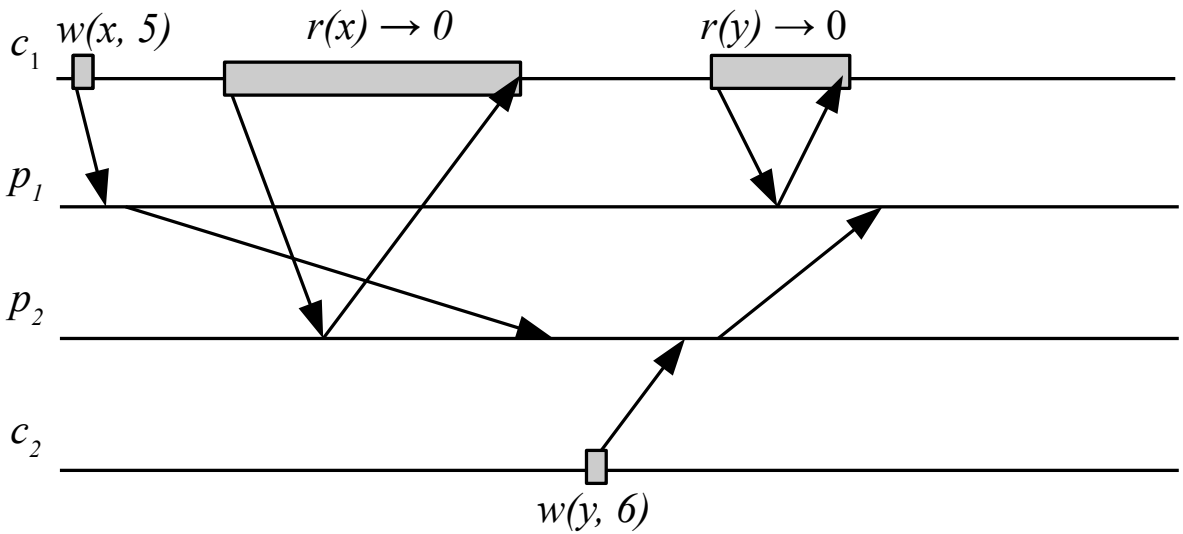
Sequential Consistency



Example

- Is the execution above sequentially consistent?

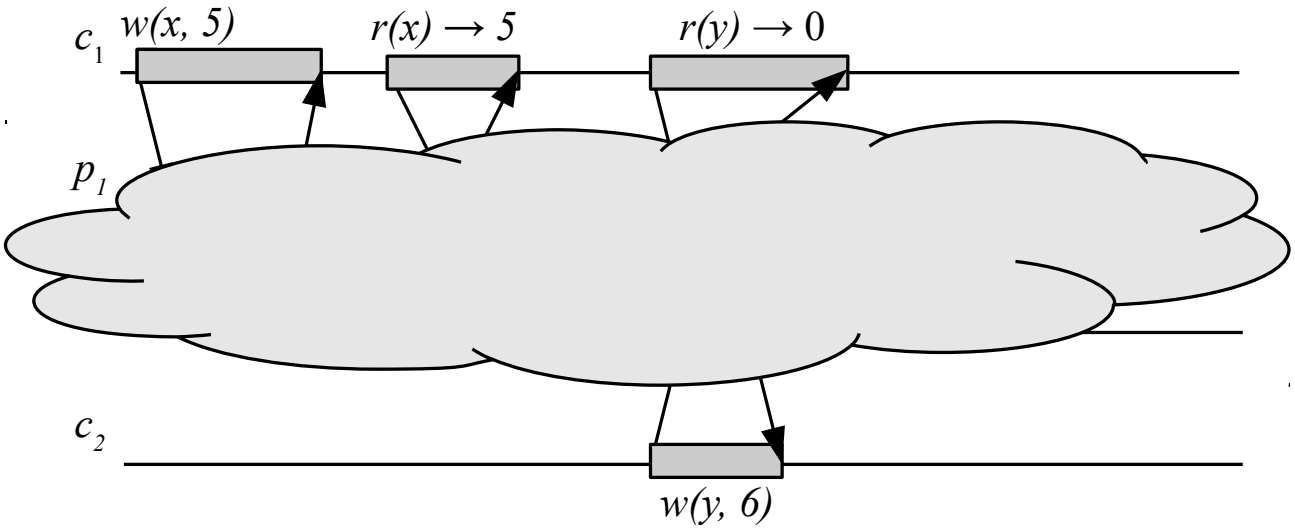
Sequential Consistency



Example

- Is the execution above sequentially consistent? NO

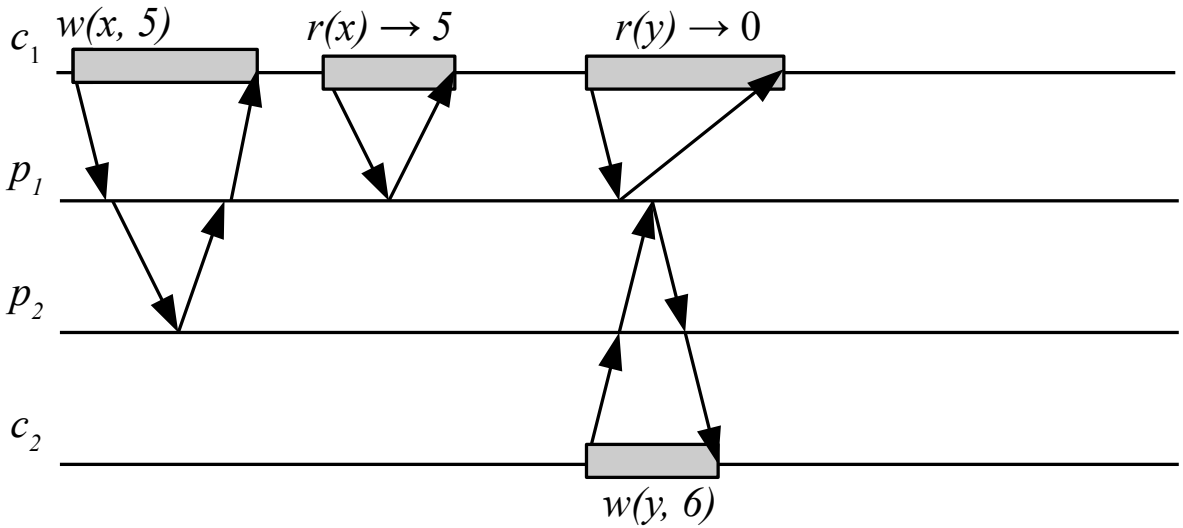
Sequential Consistency



Example

- Is the execution above sequentially consistent?

Sequential Consistency



Example

- Is the execution above sequentially consistent? YES

Sequential Consistency Example

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

- Initially, all variables have value 0
- How many "potential executions" (actions executed in any order)?
  $6! = 720$
- How many "valid executions" (with client partial order)?
  $(5!/4) \cdot 3 = 90$
- How many "potential outputs" (signatures ordered by p_1, p_2, p_3)?
  $2^6 = 64$

Sequential Consistency Example

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

- Initially, all variables have value 0
- How many "potential executions" (actions executed in any order)? $6! = 720$
- How many "valid executions" (with client partial order)? $(5!/4) \cdot 3 = 90$
- How many "potential outputs" (signatures ordered by p_1, p_2, p_3)? $2^6 = 64$

Three variables, replicated at the three processes. The $\rightarrow$ operation is a write, the print operation require two reads.

- Potential executions: corresponds top permutations, events can appear in any order
- Valid executions: consider the $5! = 120$ possible execution sequences that start with $x \leftarrow 1$
  - half of them have print x, z before $y \leftarrow 1$
  - half of this half have print x, y before $z \leftarrow 1$

  so only 30 out of 120 respect condition 2
- If we consider the sequences that start with $y \leftarrow 1$ and $z \leftarrow 1$, we have $(5!/4) \cdot 3$

Weak consistency

Problem

- It is easy to provide strong consistency through appropriate hardware and/or software mechanisms
- But these are typically found to incur considerable penalties, in latency, availability after faults, etc.

Example

- Strong consistency often implies that message should arrive in the same order
- Can be implemented through a sequencer replica
- Latency: the sequencer replica becomes a bottleneck
- Availability: a new sequencer must be elected after a failure

Weak consistency

Weak consistency models

Different weak models differs based by the precise details of which reorderings are allowed

- within the activity of a client
- by whether there are any constraints at all on the information provided to different clients

Eventual Consistency

Scenario: consider a system where

- updates are rare
- concurrent updates are absent, or can be easily resolved in an automatic way

Example: DNS

- Updates are rare w.r.t. to reads!
- Only a centralized authority can update the system; no concurrent updates.

Do we need sequential consistency in this case?

Eventual Consistency

Definition (Eventual consistency)

If no updates take place for a long time, all replicas will gradually become consistent (i.e., the same)

Comment:

- The consistency policy of epidemic protocols
- This is not a safety property, is a liveness one
- What happens in our three- process example with prints?

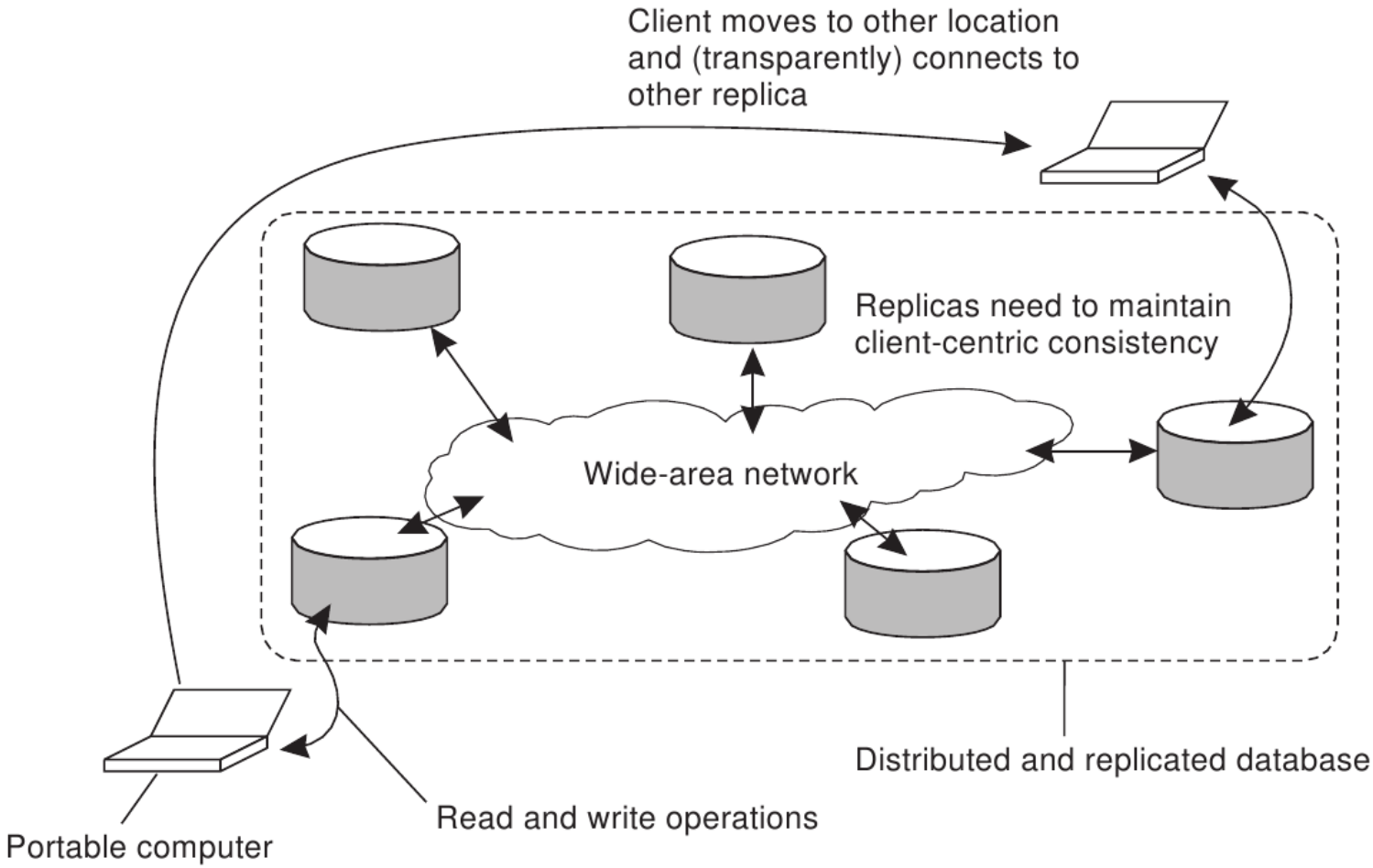
Eventual Consistency

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: Is 000000 eventual consistent? Yes
- In general, all the potential 64 outputs are possible

Consistency for mobile users

Consider a replicated database that you access through your notebook. The notebook acts as a front-end to the database



Consistency for mobile users

Problem: Eventual Consistency is not sufficient

- You move from location A to location B
- Unless you use the same server, 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

What we can do?

The only thing you really care 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

Idea

In some cases, we can avoid system-wide consistency, by concentrating on what specific clients want, instead of what should be provided by servers

Models:

- Read-after-read / Monotonic reads
- Write-after-write / Monotonic writes
- Read-after-write / Read-your-writes
- Write-after-read / Write-follows-reads

Justifications

Definition (Justification)

Each operation o performed in an execution E has a justification J_o, which is a sequence of other operations that occurred in E such that the return value o received in E is the one which the sequential data type would give to operation o when performed in the state which is produced by starting in the initial state and then applying each operation in J_o in turn.

Monotonic reads – Read-after-read

Definition (Monotonic reads)

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

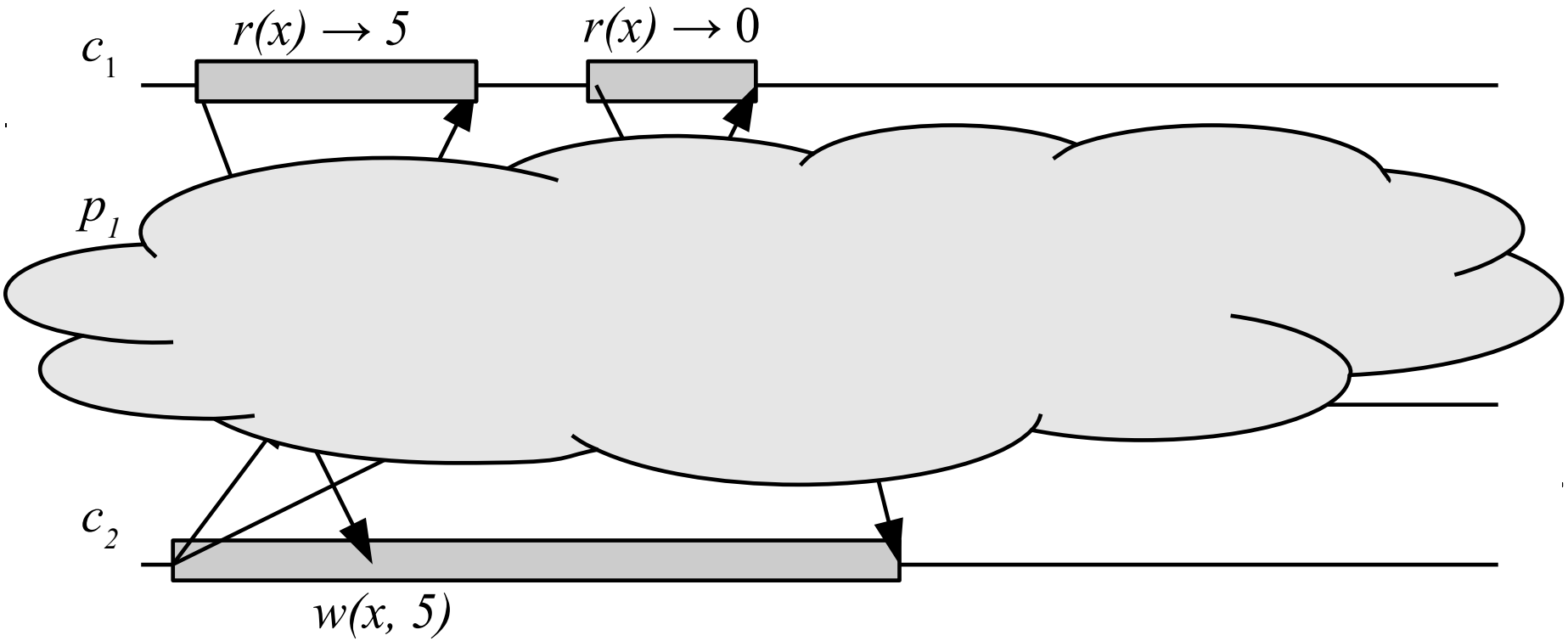
Formally

Given two read operations o_1 and o_2 submitted by a client c, the justification J_{o_1} is a prefix of justification of J_{o_2}

Example

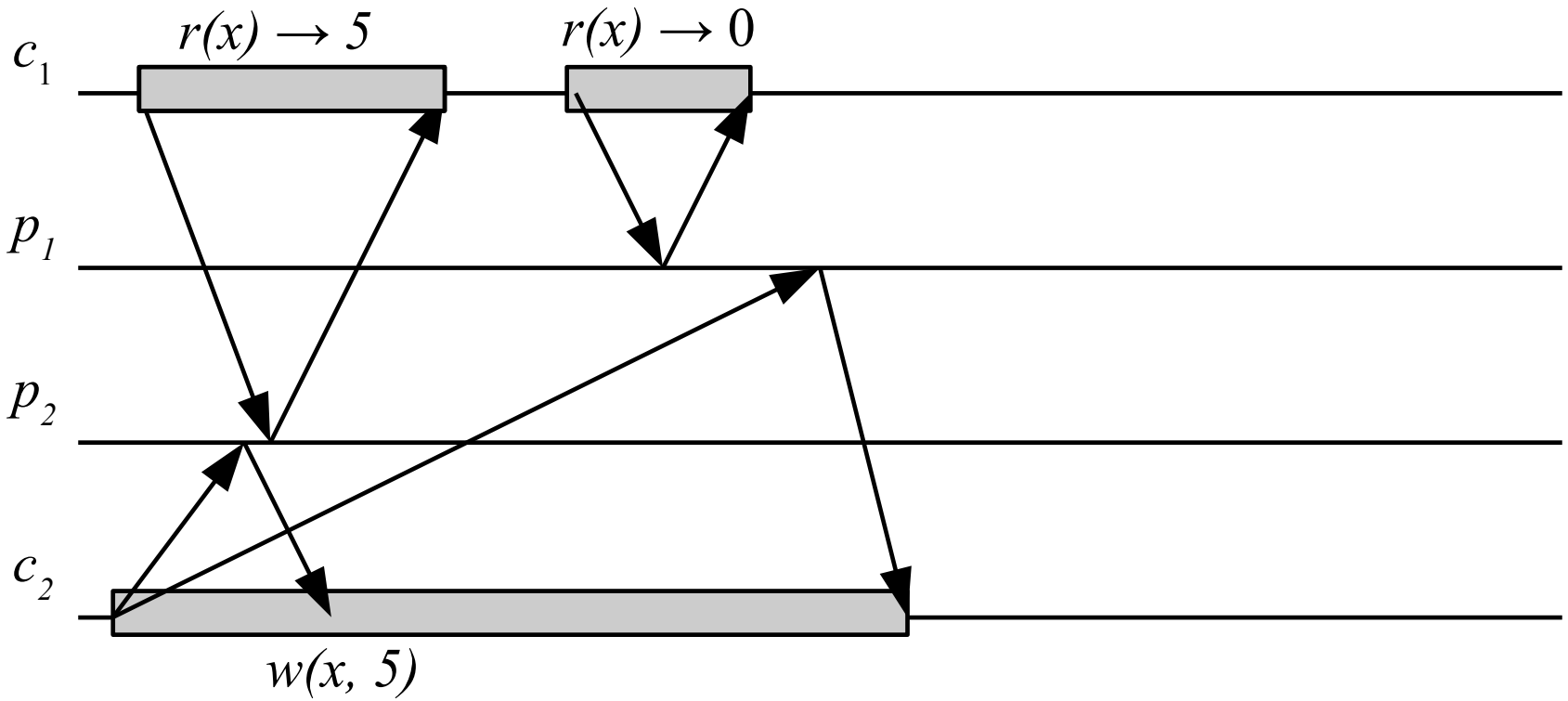
Reading incoming mail on a web-server. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited

Monotonic reads – Read-after-read



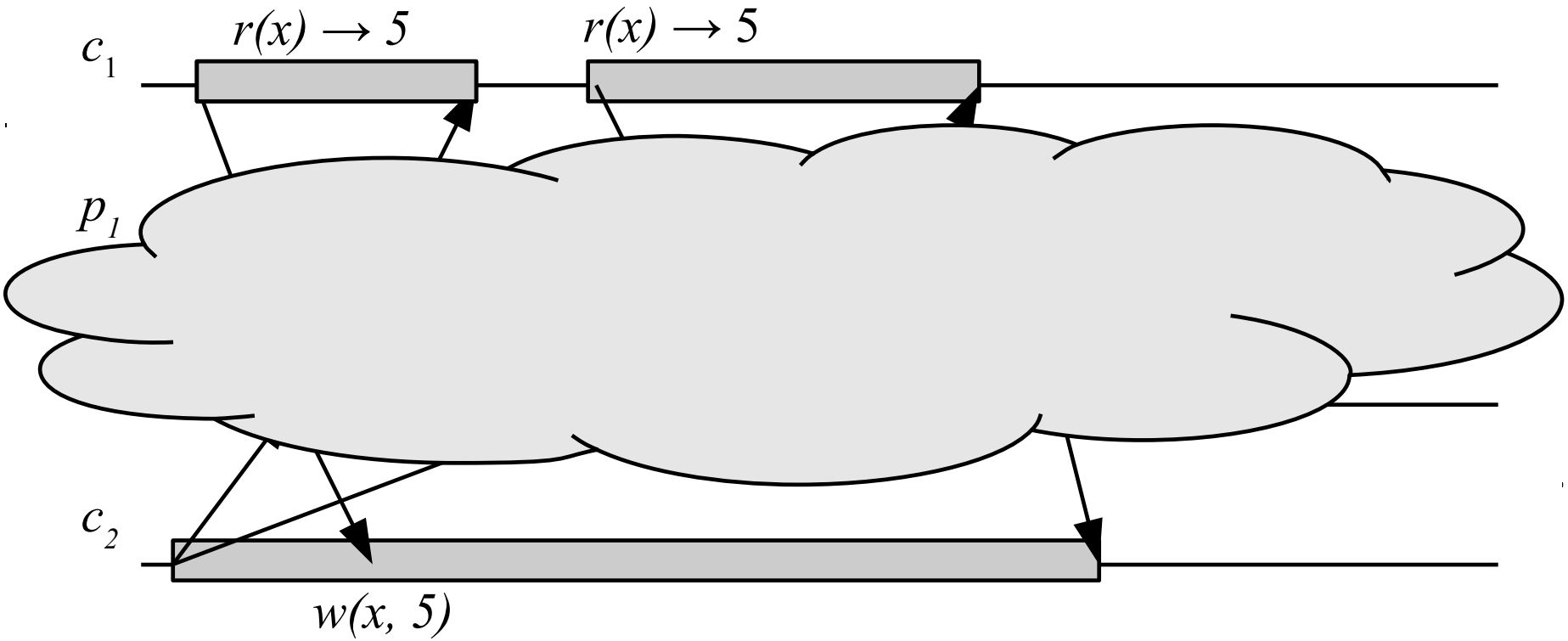
- Is the above execution satisfying read-after-read?

Monotonic reads – Read-after-read



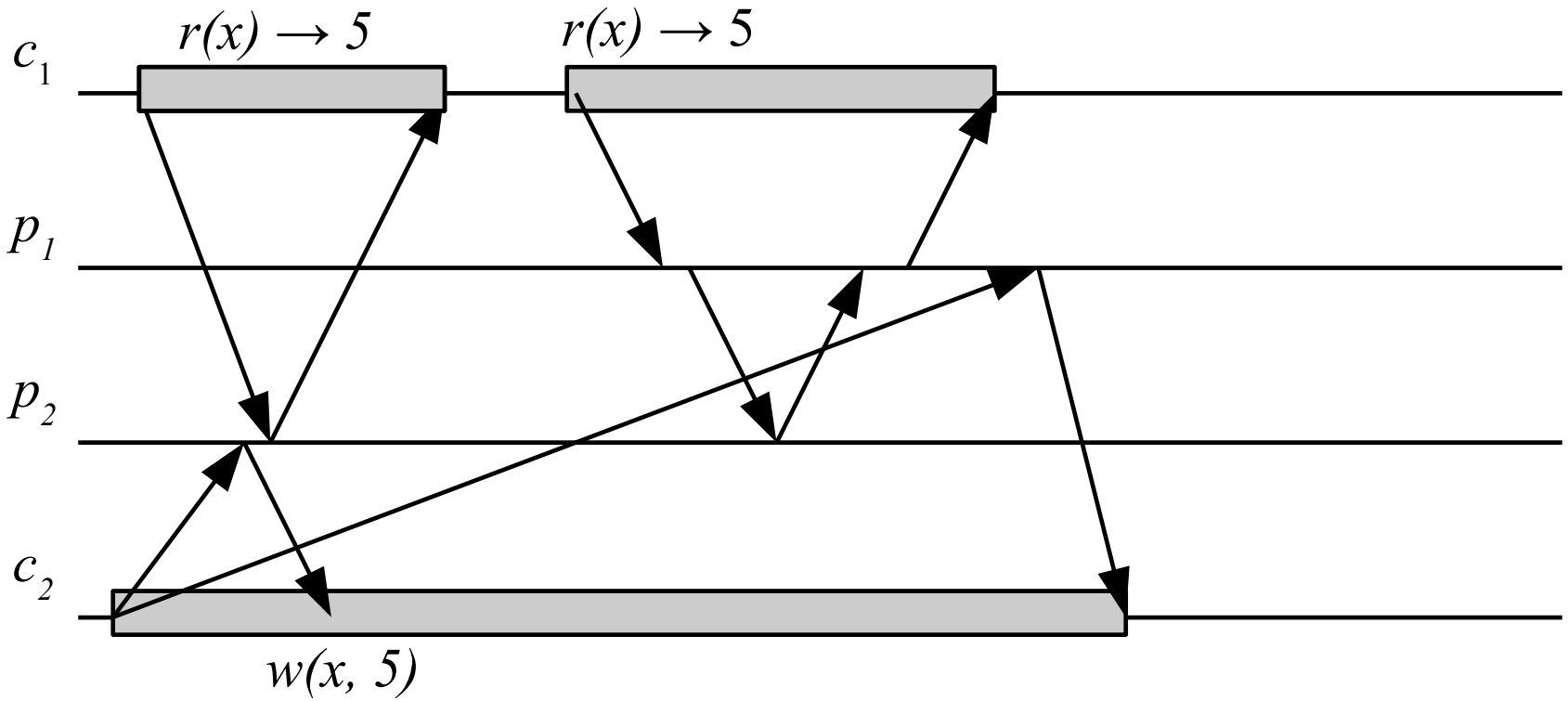
- Is the above execution satisfying read-after-read? NO!

Monotonic reads – Read-after-read



- Is the above execution satisfying read-after-read?

Monotonic reads – Read-after-read



- Is the above execution satisfying read-after-read? YES!

Read your writes – Read-after-write

Definition (Read your writes)

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

Formally

When submitting a read operation o, J_o is equal to the sequence of write operation performed by client c before submitting o.

Example

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

Other client-centric models

Definition (Monotonic writes)

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

Definition (Writes follow read)

A write operation by a process P on a data item x following a previous read operation on x by P, is guaranteed to take place on the same or a more recent value of x that was read

Definition (Monotonic writes)
A write operation by a process on a data item x is completed before any successive write operation on x by the same process

Definition (Writes follow read)
A write operation by a process P on a data item x following a previous read operation on x by P, is guaranteed to take place on the same or a more recent value of x that was read

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

Causal consistency wouldn't be better in this case?

Session consistency

Definition (Session consistency)

- A practical version of read-your-writes, where processes access a data storage in the context of a session
- As long as the session exists, the system guarantees read-your-writes
- If the session terminates because of a failure, a new session must be created
- Guarantees are limited to sessions

Causal Consistency – (Hutto and Ahamad, 1990)

Definition (Causal Consistency)

All writes that are (potentially) causally related must be seen by every process in causal order

Define "causally related":

- a read followed by a write, on the same process:
  - the write is (potentially) causally related by the read
- a write followed by a read of the same value, on diff. process:
  - the read is (potentially) causally related by the write

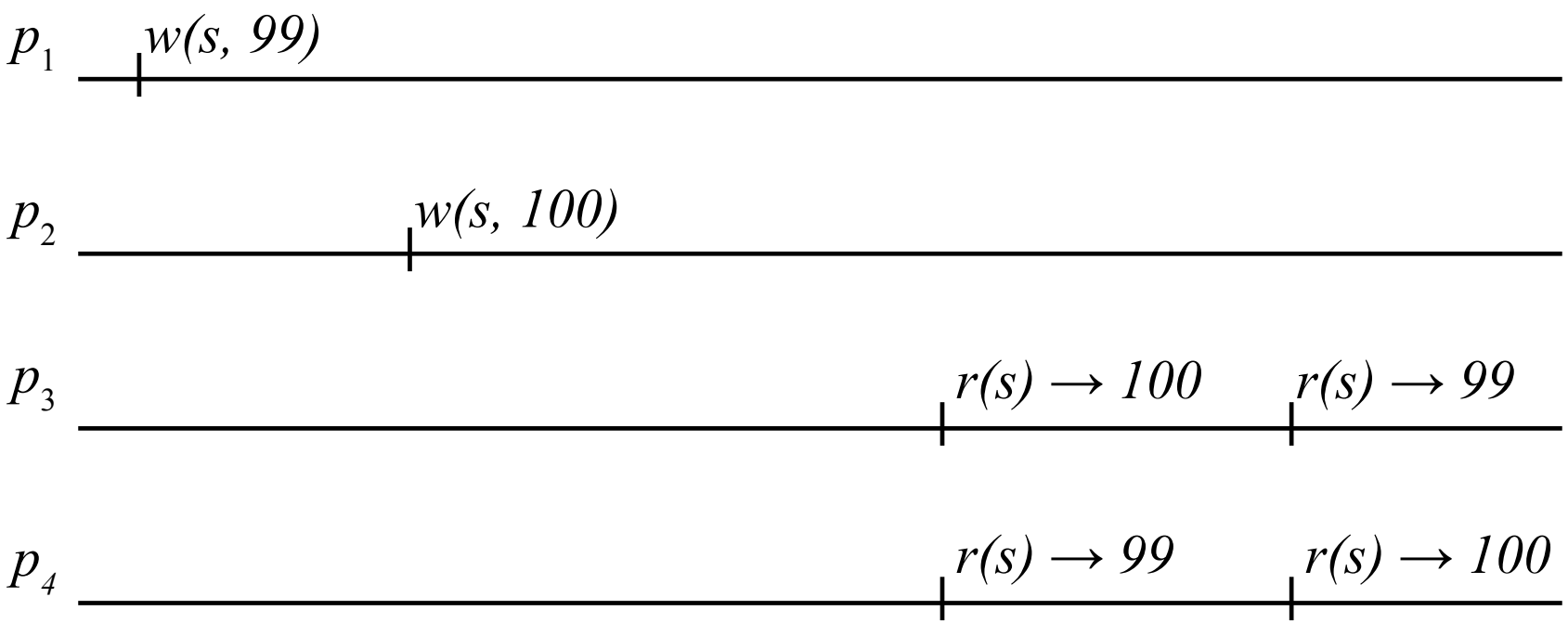
Example of use:

- Bulletin board

Causal Consistency

Example

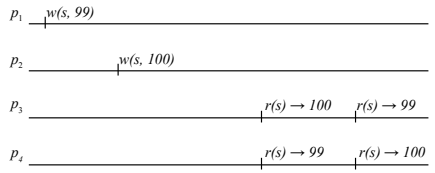
- Is the following example causally consistent?
- Is the following example sequentially consistent?



Causal Consistency

Example

- Is the following example causally consistent?
- Is the following example sequentially consistent?

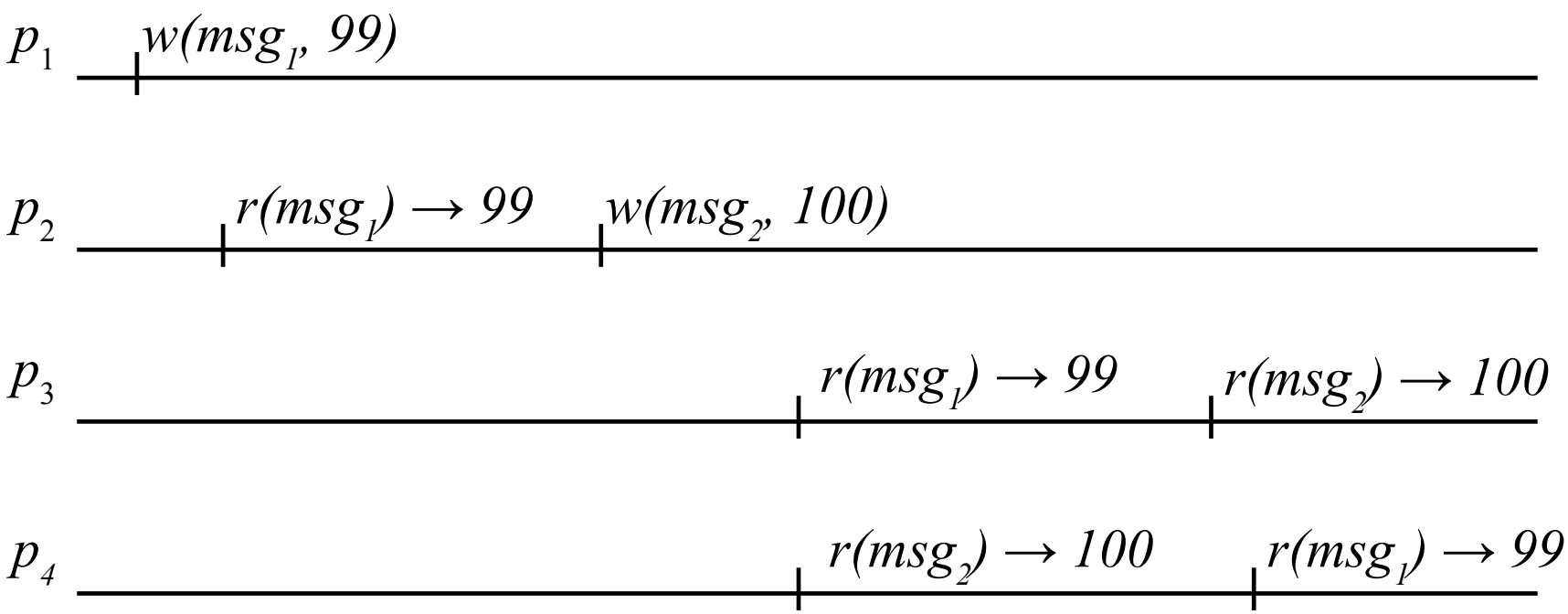


Interpretation: message 99 and message 100 are written in a newsgroup. Given that the two messages are not related, they could appear in any order.

Causal Consistency

Example

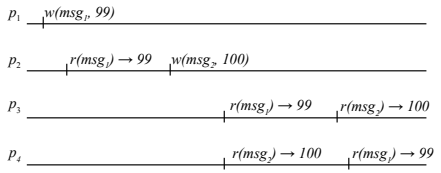
- Is the following example causally consistent?
- Is the following example sequentially consistent?



Causal Consistency

Example

- Is the following example causally consistent?
- Is the following example sequentially consistent?



Interpretation: message 99 and message 100 are written in a newsgroup. Message 100 could be potentially caused by message 99

Reality Check

Amazon S3

Amazon S3 (Simple Storage Service) is an online storage web service offered by Amazon Web Services. S3 is designed to provide 99.99% availability and 99.999999999% durability of objects over a given year.

From Amazon S3's FAQ (2014)

***Q**: What data consistency model does Amazon S3 employ?*
Amazon S3 buckets in the US West (Northern California), EU (Ireland), Asia Pacific (Singapore), and Asia Pacific (Tokyo) Regions provide read-after-write consistency for PUTS of new objects and eventual consistency for overwrite PUTS and DELETES. Amazon S3 buckets in the US Standard Region provide eventual consistency.

Reality Check

Amazon S3

Amazon S3 (Simple Storage Service) is an online storage web service offered by Amazon Web Services. S3 is designed to provide 99.99% availability and 99.999999999% durability of objects over a given year.

From Amazon S3's FAQ (2016)

***Q**: What data consistency model does Amazon S3 employ?*
Amazon S3 buckets in all Regions provide read-after-write consistency for PUTS of new objects and eventual consistency for overwrite PUTS and DELETES.

Reality Check

Berkeley DB

Oracle's Berkeley DB is a computer software library that provides a high-performance embedded database for key/value data. Used in Postfix, Subversion, SpamAssassin, BitCoin.

From the Berkeley DB manual

In a distributed system, the changes made at the master are not always instantaneously available at every replica, although they eventually will be. In general, replicas not directly involved in contributing to a transaction commit will lag behind other replicas because they do not synchronize their commits with the master. For this reason, you might want to make use of the read-your-writes consistency feature.

Reality Check

Apache ZooKeeper

Apache ZooKeeper is a software project of the Apache Software Foundation, providing an open source centralized configuration service and naming registry for large distributed systems. ZooKeeper is a sub project of Hadoop.

From ZooKeeper

> *Sequential Consistency: Updates from a client will be applied in the order that they were sent.*

What?

Reading material

Bibliography

- A. D. Fekete and K. Ramamritham. Consistency models for replicated data. In B. Charron-Bost, F. Pedone, and A. Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010. http://www.disi.unitn.it/~montreso/ds/papers/replication.pdf

Additional readings

Bibliography

- H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective.

  In *Proc. of 5th Biennial Conference on Innovative Data Systems Research (CIDR'11)*, pages 134–143, Asilomar, CA, USA, Jan. 2011.

  http://www.disi.unitn.it/~montreso/ds/papers/ConsistencyCloud.pdf

- D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser.

  Managing update conflicts in Bayou, a weakly connected replicated storage system.

  In *Proc. of the 15th ACM symposium on Operating systems principles*, SOSP'95, pages 172–182. ACM, 1995.

  http://www.disi.unitn.it/~montreso/ds/papers/bayou.pdf

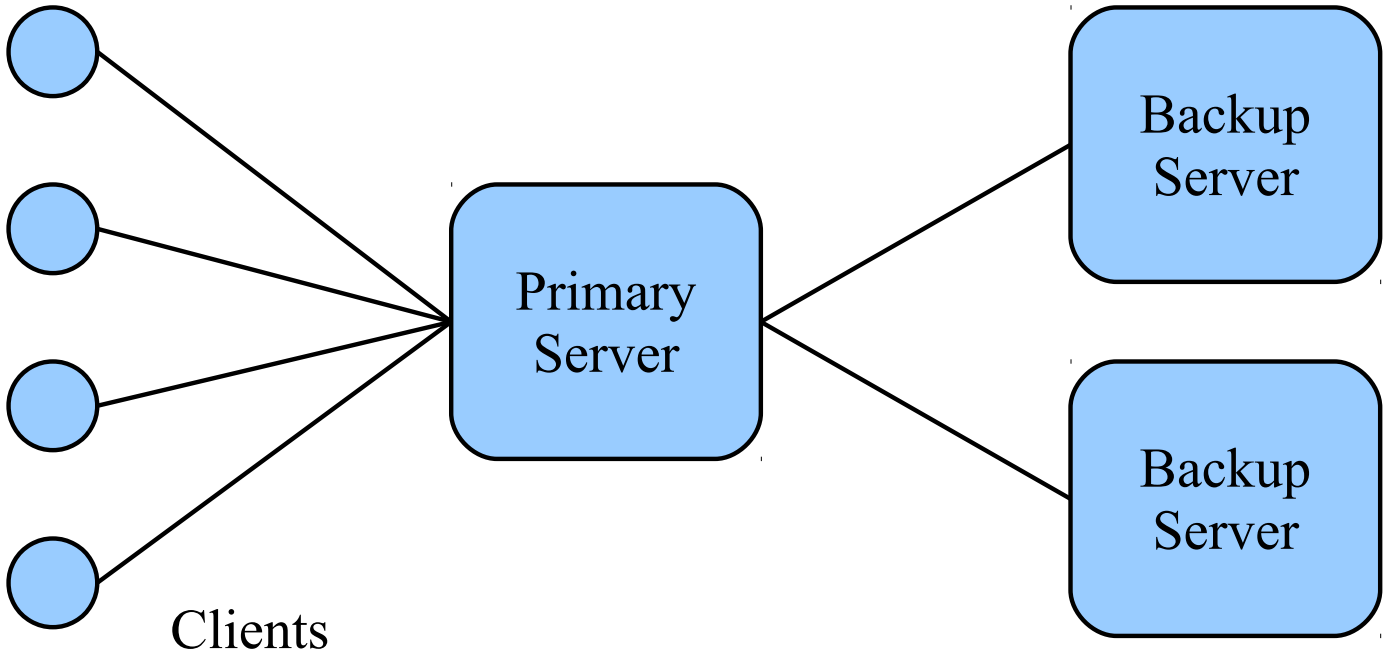
- A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.*

  Prentice-Hall, 2nd edition, 2007. [Chapter 7]

Contents

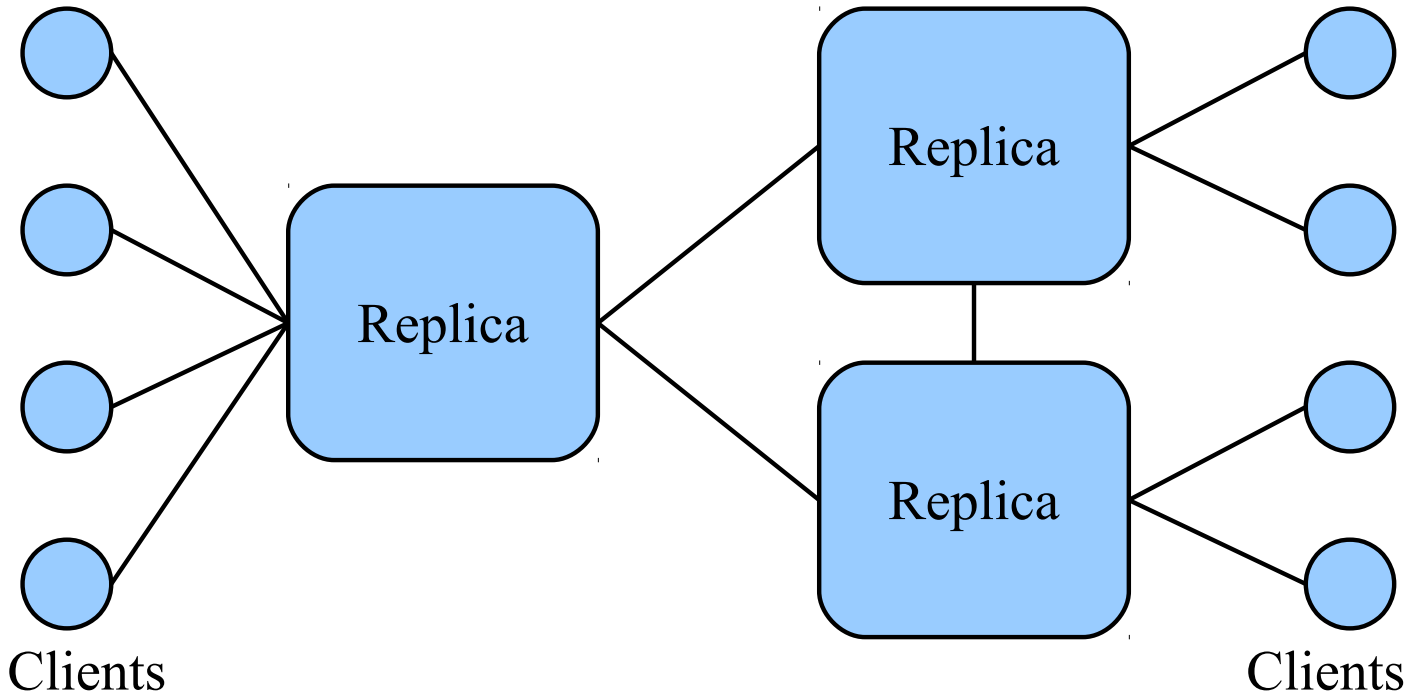
Passive replication

- Clients communicate with primary server
- Updates are forwarded from primary to backups
- Queries are replied by the primary



Active replication

- Several (all) replicas handle the invocation and send the response
- Updates must be applied in the same order – total order broadcast



Passive vs Active

Passive replication

- Computation is performed only at primary
- If state updates are large, can waste network bandwidth
- Can handle non-determinism

Active replication

- Small recovery delay after failures
- If operations are compute intensive, can waste computational resources
- Only deterministic

Consistency protocols

- Primary-based protocols
  - Definition
  - Lower bounds
- Replicated-write protocols
  - Majority, quorum-based
  - State machine approach
- Client-centric protocols
  - Monotonic reads
  - Read-your-writes

Primary-Backup

The idea

- Clients communicate with a single replica (the primary)
- The primary updates the other replicas (backup)
- Backups detect the failure of the primary using a timeout mechanism
- Clients learn from the service when the primary fails and the service "fail over" to a backup
- Note: non-deterministic events are executed only at the primary

How to evaluate a primary-backup protocol

Definition (Degree of replication)

Number of servers used to implement the service; the smaller, the better

Definition (Blocking time)

The worst-case period between a request and its response in any failure-free execution

Definition (Failover time)

The worst-case period during which request can be lost because there is no primary

Definitions

Definition (Service outage)

The service has a server outage at t if some correct client sends a request at time t to the service, but does not receive a response

Definition ((k, Δ)-bofo service - "bounded outage, finitely often")

A service in which all server outages can be grouped into at most k intervals of time, each of them at most length Δ

Specification

PB1 At any time, there is at most one server p_i that acts as a primary

PB2 If a client request arrives at a server that is not the current primary, then the request is ignored

PB3 There exist fixed values k and Δ such that the service behaves like a single (k, Δ)-bofo service

Primary-backup – Simple protocol

System model:

- point-to-point communication
- no communication failures
- upper bound δ on message delivery time
- FIFO channels
- at most one server crashes

Two servers:

- The primary p_1
- The backup p_2

Variables:

- At server p_i, $primary = \textbf{true}$ if p_i acts as the current primary
- At clients, $primary$ is equal to the identifier of the current primary

Primary-backup – Simple protocol

System model:

- point-to-point communication
- no communication failures
- upper bound δ on message delivery time
- FIFO channels
- at most one server crashes

Two servers:

- The primary p_1
- The backup p_2

Variables:

- At server p_i, *primary* = **true** if p_i acts as the current primary
- At clients, *primary* is equal to the identifier of the current primary

Can we realize such system? From the point of view of communication, we can use a dedicate network interface to connect primary and backup. This guarantees bounded delay and (potentially) accurate failure detection.

Maximum one failure means “before an administrator takes action”.

Primary-backup – Simple protocol

Protocol executed by the primary p_1

upon initialization **do**
  $primary \leftarrow$ **true**

upon receive $\langle \text{REQ}, r \rangle$ **from** c **do**
  $state \leftarrow \mathsf{update}(state, r)$ % Update local state
  send $\langle \text{STATE}, state \rangle$ **to** p_2 % Send update to backup
  send $\langle \text{REP}, \mathsf{reply}(r) \rangle$ **to** c % Reply to client

repeat every τ seconds
  send $\langle \text{HB} \rangle$ **to** p_2 % Heartbeat message

upon recovery after a failure **do**
  { start behaving like a backup }

Primary-backup – Simple protocol

Protocol executed by the backup p_2

upon initialization **do**
  $primary \leftarrow$ **false**

upon receive $\langle \text{STATE}, s \rangle$ **do**
  $state \leftarrow s$ % Update local state

upon not receiving a heartbeat for $\tau + \delta$ seconds **do**
  $primary \leftarrow$ **true** % Becomes new primary
  send $\langle \text{NEWP} \rangle$ **to** c % Inform the client of new primary
  { start behaving like a primary }

Primary-backup – Client code

Protocol executed by client c

upon initialization **do**
    $primary \leftarrow p_1$ % Initial primary

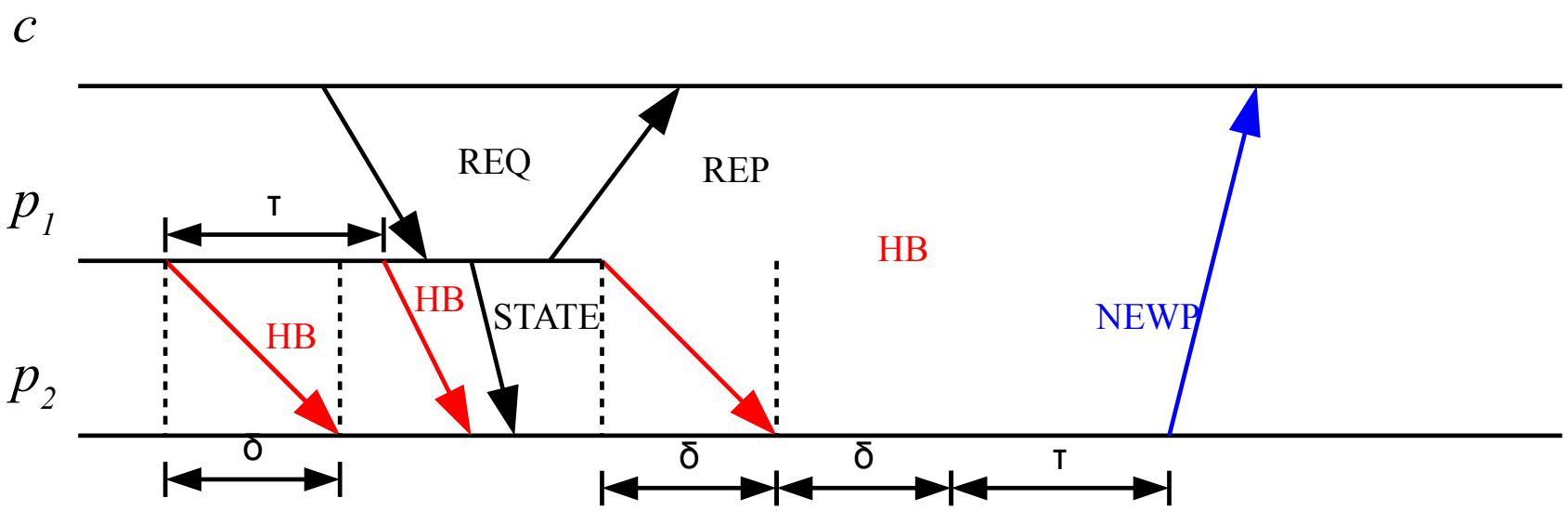
upon receive $\langle \text{NEWP} \rangle$ **from** p_2 **do**
    $primary \leftarrow p_2$ % Backup

upon operation(r) **do**
    while not received a reply **do**
        send $\langle \text{REQ}, r \rangle$ **to** $primary$
        wait receive $\langle \text{REP}, v \rangle$ **or receive** $\langle \text{NEWP} \rangle$
    return v

Simple protocol – Proof of correctness

PB1 At any time, there is at most one server p_i that acts as a primary

Proof
- $primary_1 = \textbf{true} \wedge primary_2 = \textbf{false}$ until the failure of p_1
- $primary_2 = \textbf{false}$ until the expiration of the timeout
- $primary_2 = \textbf{true}$ after the expiration of the timeout
- Failover time: $\tau + 2\delta$



Simple protocol – Proof of correctness

PB2 If a client request arrives at a server that is not the current primary, then the request is ignored

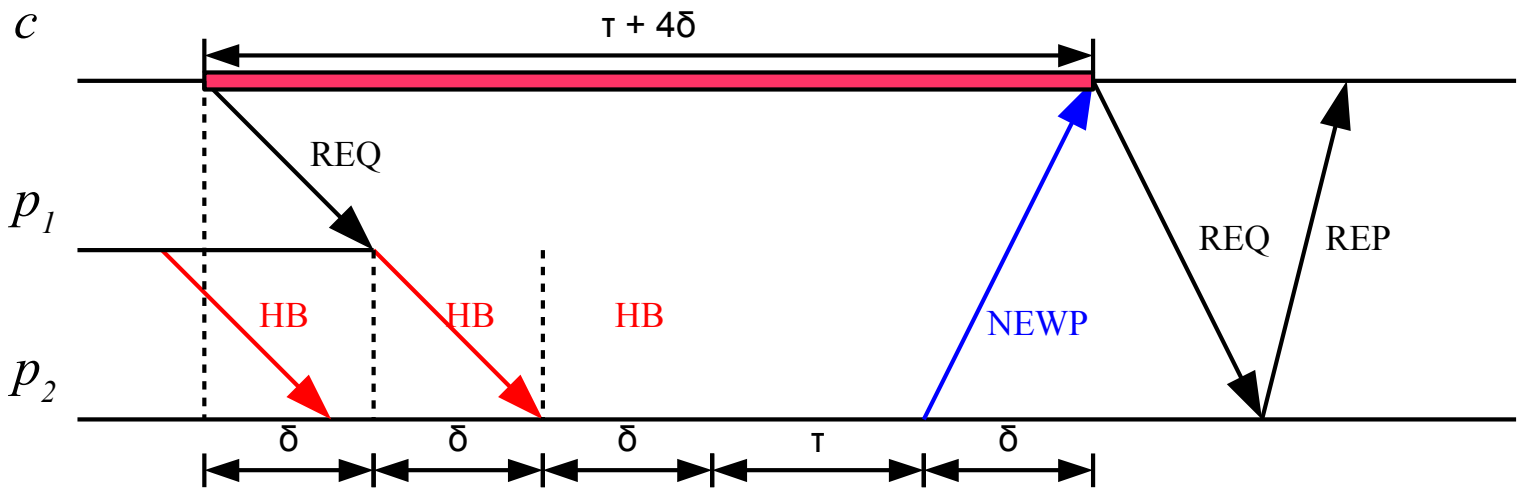
Proof Trivially follows from the protocol

Simple protocol – Proof of correctness

PB3 There exist fixed values k and Δ such that the service behaves like a single (k, Δ)-bofo service

Proof Find k, Δ

- At most one process can fail: $k = 1$
- $\Delta = \tau + 4\delta$:
  - ★ assume p_1 crashes at t_c
  - ★ any client request sent to p_1 at time $t_c - \delta$ or later may be lost
  - ★ p_2 may not become the new primary until $t_c + \tau + 2\delta$
  - ★ client may not learn that p_2 is new primary for another δ



Simple protocol – Questions

Question

What kind of consistency model is provided by this simple protocol?

Answer: Linearizability

An execution E is linearizable provided that there exists a sequence (linearization) H such that

- L1 H contains exactly the same operations that occur in E, each paired with the return value received in E
- L2 H is a legal history of the sequential data type that is replicated
- L3 the total order of operations in H is compatible with the real-time partial order $<_{E,rt}$

Simple protocol – Questions

Question

What kind of consistency model is provided by this simple protocol?

Answer: Linearizability

An execution E is linearizable provided that there exists a sequence (linearization) H such that

- L1 H contains exactly the same operations that occur in E, each paired with the return value received in E
- L2 H is a legal history of the sequential data type that is replicated
- L3 the total order of operations in H is compatible with the real-time partial order $<_{E,rt}$

Given that we are assuming no communication failures, the simple sending of a message before the crash means that the message will be received, before the backup actually declare itself as new primary.

Primary-backup – Multiple backups

System model:

- point-to-point communication
- Perfect Channels
- perfect failure detector P
- FIFO channels
- at most $f < n$ servers crash

n servers:

- $p_1, \ldots, p_n$

Primary-backup – Multiple backups

Protocol executed by process p_i

upon receive $\langle \text{REQ}, id, r \rangle$ **from** c **do**
- $servers \leftarrow servers - \{p_j : p_j \in servers \wedge j < i\}$
- **if** $id \notin state$ **then**
  - $state \leftarrow \mathsf{update}(state, r)$
  - **send** $\langle \text{STATE}, state, id \rangle$ **to** $servers$
  - **wait receive** $\langle \text{STATE}, id \rangle$ **from** $servers$
- **send** $\langle \text{REP}, id, \mathsf{reply}(r) \rangle$ **to** c

upon $\mathsf{suspect}(p_j)$ **do**
- $servers \leftarrow servers - \{p_j\}$

upon receive $\langle \text{STATE}, id, s \rangle$ **from** p_k **do**
- $servers \leftarrow servers - \{p_j : p_j \in servers \wedge j < k\}$
- **if** $p_k \in servers$ **then**
  - $state \leftarrow s$
  - **send** $\langle \text{STATE}, id \rangle$ **to** p_k

Primary-Backup – Client code

Protocol executed by client c

upon initialization **do**
  MAP *response* ← **new** MAP();

upon receive ⟨REP, *id*, *v*⟩ **do**
  response[*id*] ← *v*

upon suspect(p_j) **do**
  servers ← *servers* − $\{p_j\}$

upon operation(r) **do**
  id ← newId()
  while *servers* ≠ ∅ **or** *response*[*id*] = **nil do**
    p_k ← min(*servers*)
    send ⟨REQ, *id*, *r*⟩ **to** p_k
    wait *response*[*id*] ≠ **nil or** $p_k \notin$ *servers*
  return *response*[*id*]

Primary-backup – Multiple backups

- How large is the failover time?
  $\tau + 2\delta$, as before (hidden in the Failure Detector)
- How large is the outage period Δ?
  $(\tau + 2\delta)f$
- What kind of consistency model we obtain if all operations are handled by the primary?
  Linearizability
- What kind of consistency model we obtain if only write operations are handled by the primary?
  Sequential consistency

Lower bounds

- Assuming that no more than f components can fail, what are the smallest possible values (lower bounds) of
  - the degree of replication
  - the failover time?
  - the blocking time
- Knowing the lower bounds for a problem enables to evaluate the quality of a protocol
- Tight lower bounds $\rightarrow$ optimal protocols
- Components:
  - Processes
  - Point-to-point links
  - Up to f crash+link failures $\rightarrow$ at most f processes may crash or f links may crash or f_1 links + f_2 processes = f components

Lower bounds

Failure Model	Degree of Replication	Blocking Time	Failover Time
crash	$n > f$	0	$f\delta$
crash+link	$n > f + 1$	0	$2f\delta$
rec-omission	$\left\lfloor \frac{3f}{2} \right\rfloor$	2δ	$2f\delta$
send-omission	$n > f$	2δ	$2f\delta$
omission	$n > 2f$	2δ	$2f\delta$

Lower bounds

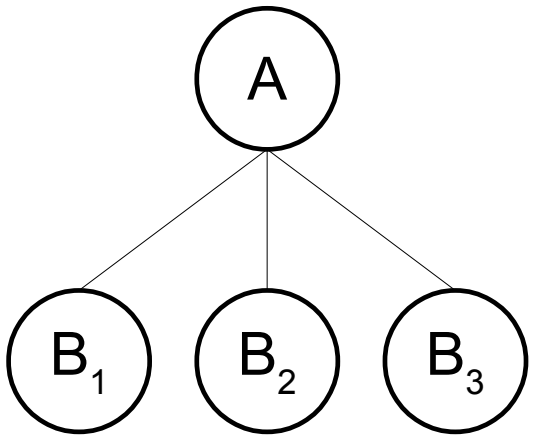
Crash+link

To tolerate up to f crash+link failures, more than $f + 1$ servers are needed

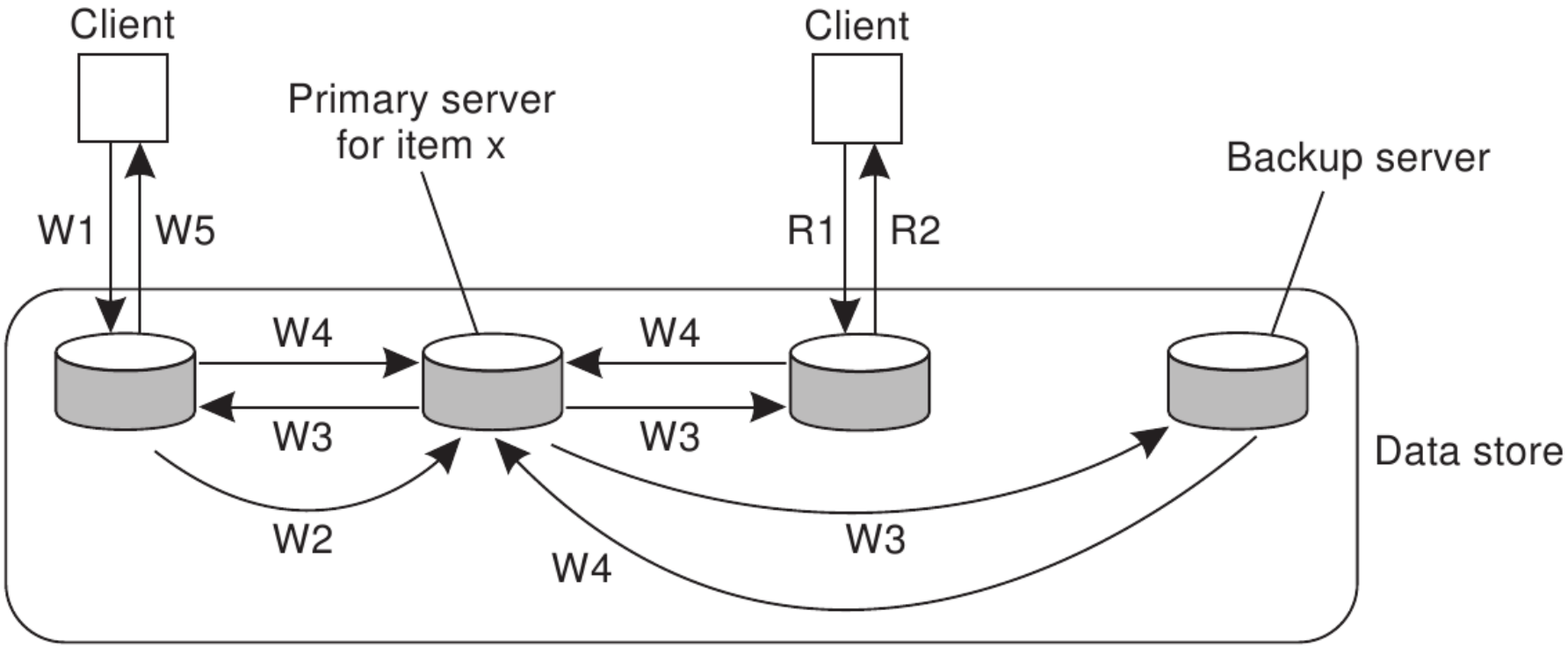
Proof – by contradiction

Suppose $n = f + 1$ servers is sufficient

- divide the n servers in two subsets A and $B_1 \ldots B_f$
- if all server in B crash, A must become primary
- if A crashes, one of servers B_i must become primary
- what if all f links between A and B_i fails?



Multiple primaries



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

Quorum protocols (Gifford, 1979)

Definition

Quorum-based protocols guarantee that each operation is carried out in such a way that a majority vote (a quorum) is established.

- Write quorum n_W: the number of replicas that need to acknowledge the receipt of the update to complete the update
- Read quorum n_R: the number of replicas that are contacted when a data object is accessed through a read operation

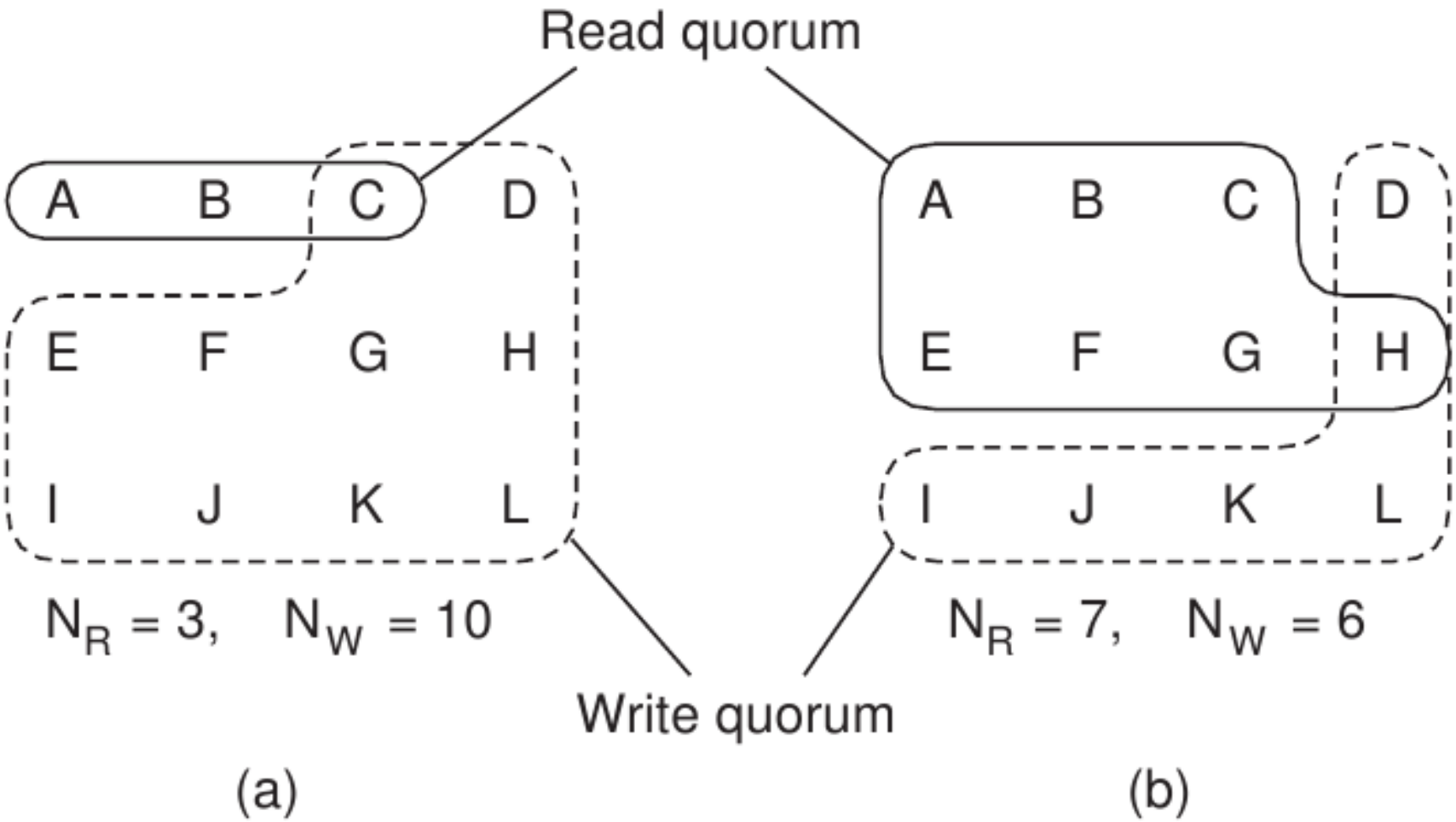
Constraints

- $n_R + n_W > n$ (prevent R-W conflicts)
- $n_W > n/2$ (prevent W-W conflicts)

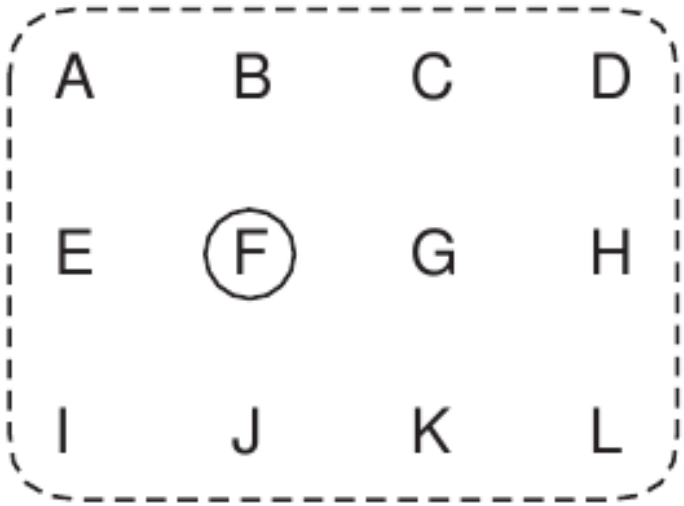
The algorithm

- To read, the most up-to-date entry is taken
- Quorums guarantee that the last written entry will be present

Quorum protocols (Gifford, 1979)



Quorum protocols (Gifford, 1979)



State machine

Definition (State machine)

A state machine consists of:

- State variables
- Commands which transforms its state
  - Implemented by deterministic programs
  - Atomic with respect to other commands

Specification

- Agreement: every correct replica receives the same set of commands
- Order: every non-faulty state machine processes the commands it receives in the same order

Implementing linearizability – General scheme

Implementation

- The initiator A-broadcasts all read, write requests to all servers
- When the message is A-delivered at the initiator, it replies to the client

Correctness

- All replicas execute read, write in the same order

Assumptions

- Synchronous system
- Asynchronous system with $\diamond S$ failure detector

Implementing sequential consistency – General scheme

Implementation

- The initiator A-broadcasts write requests to all servers
- When the message is A-delivered, the replica updates its local copy
- Read request are replied immediately by the initiator

Correctness

- Writes are executed in the same order everywhere
- Reads are consistent with local order

Assumptions

- Synchronous system
- Asynchronous system with $\diamond S$ failure detector

Implementing causal consistency – General scheme

Implementation

- The initiator C-broadcasts write requests to all servers
- When the message is C-delivered, the replica updates its local copy
- Read request are replied immediately by the initiator

Correctness

- Writes are executed in a causal order
- Reads are consistent with local (and causal) order

Assumptions

- Asynchronous system

Hypervisor-based fault tolerance

- Implement state machine on virtual machines running on the same instruction-set as underlying hardware
- Undetectable by higher layers of software
- One of the great come-backs in systems research!
  - CP-67 for IBM 369 [1970]
  - Xen [SOSP 2003], VMware
- State transition should be deterministic
- ...but some VM instructions are not (e.g. time-of-day)!
- Two types of commands
  - Virtual-machine instructions
  - Virtual-machine interrupts (with DMA input)
    Interrupts must be delivered at the same point in cmd sequence

Hypervisor-based fault tolerance

- Thomas C. Bressoud, Fred B. Schneider. Hypervisor-based Fault Tolerance. ACM TOCS, 14(1):80-107
- John R. Douceur an Jon Howell. Replicated Virtual Machines. Microsoft Research TR-2005-119
  - Technical paper associated to a patent
- Brendan Cully et al. Remus: High Availability via Asynchronous Virtual Machine Replication. NSDI'08.
  - Best paper award
  - Real implementation for XEN

Client-centric consistency - Naive implementations

- Each write operation is assigned a unique identifier
  - Done by the server where the operation is requested
- For each client c, we keep track of:
  - Read set WS_r: contains write operations relevant to the read operations performed by c
  - Write set WS_w: contains write operations relevant to the write operations performed by c
- For each server, we keep track of:
  - Write set WS: contains the write operations executed so far

Monotonic reads - Naive implementation

- To perform a read operation o_r, a client:
  - send o_r and its read set WS_r to the server
- The server
  - Checks whether all the writes in WS_r have been executed locally ($WS_r \subseteq WS$?)
  - If not, asks the appropriate servers the missing operations O
  - Applies the operations O and add them to WS
  - Returns the requested value and the WS set to the client
- The client
  - Adds WS to its local read set: $WS_r = WS_r \cup WS$

Read your writes - Naive implementation

- To perform a read operation o_r, a client:
  - send o_r and its write set WS_w to the server
- The server
  - Checks whether all the writes in WS_w have been executed locally ($WS_w \subseteq WS$?)
  - If not, asks the appropriate servers the missing operations O
  - Applies the operations O and add them to WS
  - Returns the requested value to the client
- To perform a write operation o_w, a client c
  - send o_w to the server
  - add o_w to the write set WS_w

Contents

CAP theorem

Theorem (Impossibility of CAP)

It is impossible for a web service to provide more than two of the following three guarantees:

- ***C**onsistency*
- ***A**vailability*
- ***P**artition-tolerance*

- This is the reason why Amazon Web Services only provide eventual consistency
  - W. Vogels. Eventual consistent.
    Comm. of the ACM, 52(1):40–44, 2009
- Similar stands have been taken for example by HP
  - HP. There is no free lunch with distributed systems.
    `http://www.disi.unitn.it/~montreso/ds/papers/NoFreeLunchDS.pdf`

CAP theorem

History:

- First introduced by Eric Brewer in a keynote at PODC'00
  - E. A. Brewer. Towards robust distributed systems (abstract). In *Proc. of the 19th ACM symposium on Principles of distributed computing*, PODC'00, page 7. ACM, 2000
- Formally proved by Gilbert and Lynch two years later
  - S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002. http://www.disi.unitn.it/~montreso/ds/papers/CapProof.pdf

Reading Material

- N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach.
  In S. Mullender, editor, *Distributed Systems (2^{nd} ed.)*. Addison-Wesley, 1993.
  http://www.disi.unitn.it/~montreso/ds/papers/PrimaryBackup.pdf

- F. Schneider. Replication management using the state machine approach.
  In S. Mullender, editor, *Distributed Systems (2^{nd} ed.)*. Addison-Wesley, 1993.
  http://www.disi.unitn.it/~montreso/ds/papers/StateMachine.pdf