Distributed Operating Systems (CS30009)

Consistency, Replication, Fault Tolerance

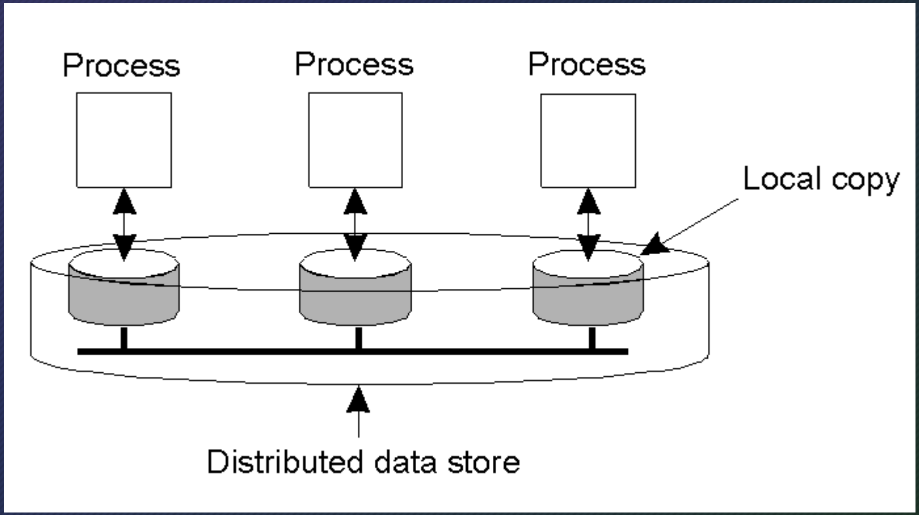
Dr. Jaydeep Das

Contents

Data-Centric Consistency Models

Introduction

- A data-store can be read from or written to by any process in a distributed system.
- A local copy of the data-store (replica) can support “fast reads”.
- However, a write to a local replica needs to be propagated to *all* remote replicas.



- Various consistency models help to understand the various mechanisms used to achieve and enable this.

What is a Consistency Model?

- A consistency model is a CONTRACT between a DS data-store and its processes.
- If the processes agree to the rules, the data-store will perform properly and as advertised.
- We start with *Strict Consistency*, which is defined as:
- Any read on a data item ‘x’ returns a value corresponding to the result of the most recent write on ‘x’ (regardless of where the write occurred).

Consistency Model Diagram Notation

- $W_i(x)a$ – A write by process ‘i’ to item ‘x’ with a value of ‘a’. That is, ‘x’ is set to ‘a’.

- (Note: The process is often shown as ‘P_i’).

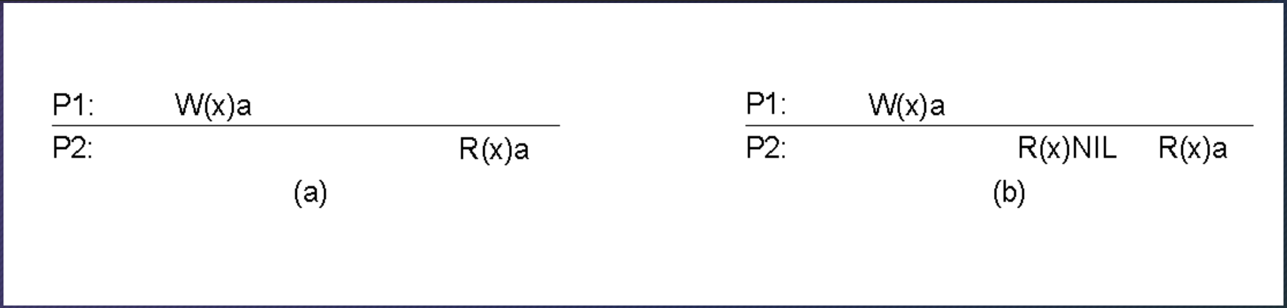
- $R_i(x)b$ – A read by process ‘i’ from item ‘x’ producing the value ‘b’. That is, reading ‘x’ returns ‘b’.

- Time moves from left to right in all diagrams.

Consistency Models

1. Strict Consistency
2. Linearizability Consistency
3. Sequential Consistency
4. Causal Consistency
5. FIFO Consistency
6. Weak Consistency
7. Release Consistency
8. Entry Consistency

1. Strict Consistency



- Behavior of two processes, operating on the same data item:

a) A strictly consistent data-store.
b) A data-store that is not strictly consistent.

- With Strict Consistency, all writes are instantaneously visible to all processes and absolute global time order is maintained throughout the Distributed System. This is the consistency model "Holy Grail" – not at all easy in the real world, and all but impossible within a Distributed System.

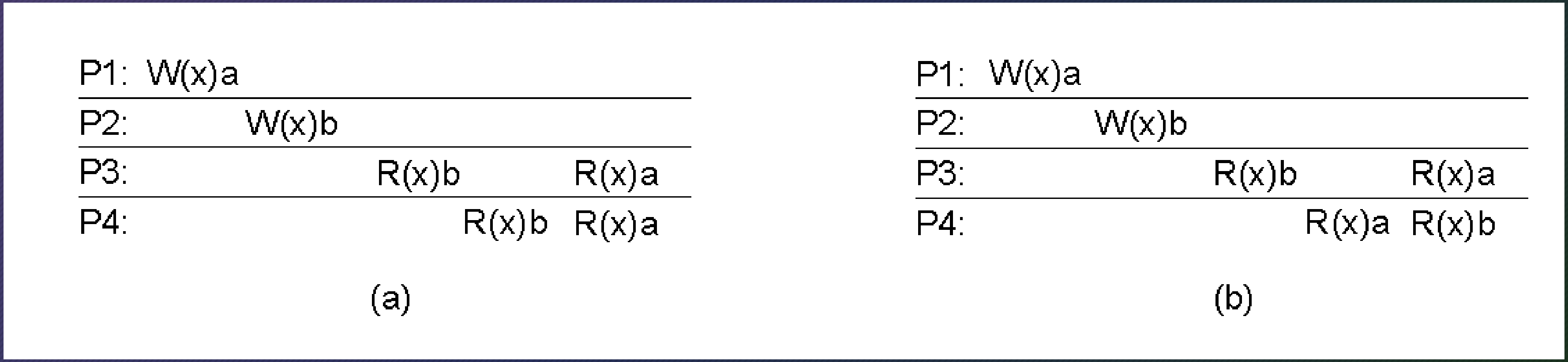
- So, other, less strict (or "weaker") models have been developed.

2. Sequential Consistency

- A weaker consistency model, which represents a relaxation of the rules.
- It is also must easier (possible) to implement.
- Definition of “Sequential Consistency”:
- The result of any execution is the same as if the (read and write) operations by all processes on the data-store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

Sequential Consistency Diagrams

In other words: all processes see the same interleaving set of operations, regardless of what that interleaving is.



a) A sequentially consistent data-store – the "first" write occurred *after* the "second" on all replicas.

b) A data-store that is not sequentially consistent – it appears the writes have occurred in a non-sequential order, and this is NOT allowed.

Problem with Sequential Consistency

- With this consistency model, adjusting the protocol to favour reads over writes (or vice-versa) can have a devastating impact on.
- For this reason, other weaker consistency models have been proposed and developed.
- Again, a relaxation of the rules allows for these weaker models to make sense.

3. Linearizability and Sequential Consistency

Process P1	Process P2	Process P3

x = 1;
print (y, z);
	y = 1;
	print (x, z);
		z = 1;
		print (x, y);

- Three concurrently executing processes.

Linearizability and Sequential Consistency (contd…)

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

```
x = 1;
print ((y, z);
y = 1;
print (x, z);
z = 1;
print (x, y);
```

Prints: 001011

Signature: 001011

(a)

```
x = 1;
y = 1;
print (x,z);
print(y, z);
z = 1;
print (x, y);
```

Prints: 101011

Signature: 101011

(b)

```
y = 1;
z = 1;
print (x, y);
print (x, z);
x = 1;
print (y, z);
```

Prints: 010111

Signature: 110101

(c)

```
y = 1;
x = 1;
z = 1;
print (x, z);
print (y, z);
print (x, y);
```

Prints: 111111

Signature: 111111

(d)

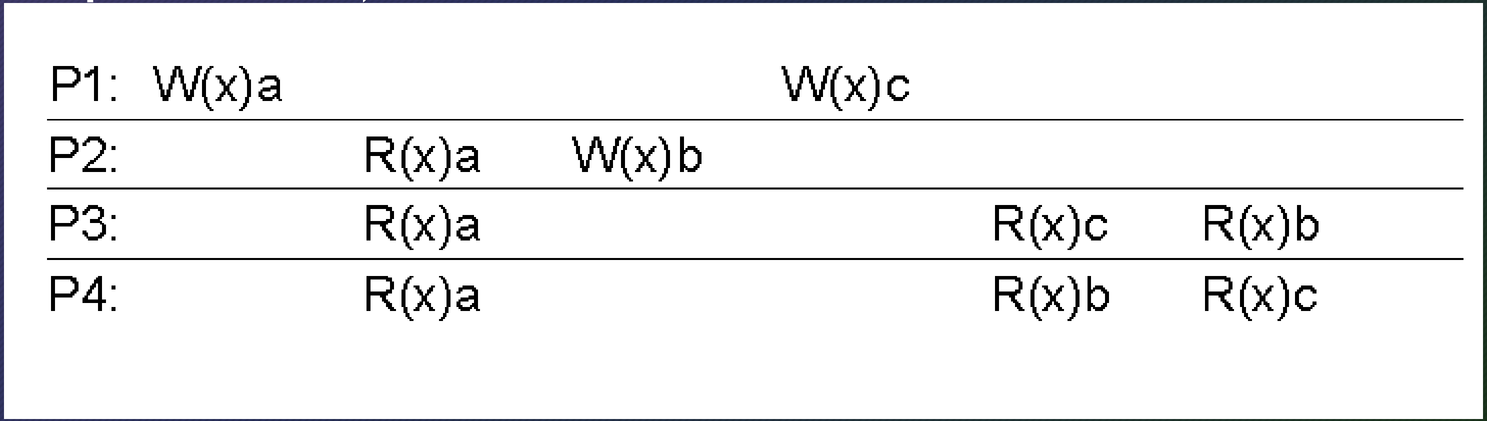
But, for instance, 001001 is not allowed.

4. Causal Consistency

- This model distinguishes between events that are “causally related” and those that are not.
- *If event B is caused or influenced by an earlier event A, then causal consistency requires that every other process see event A, then event B.*
- Operations that are not causally related are said to be *concurrent*.

Causal Consistency (contd…)

- A causally consistent data-store obeys this condition:
- Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines (i.e., by different processes).

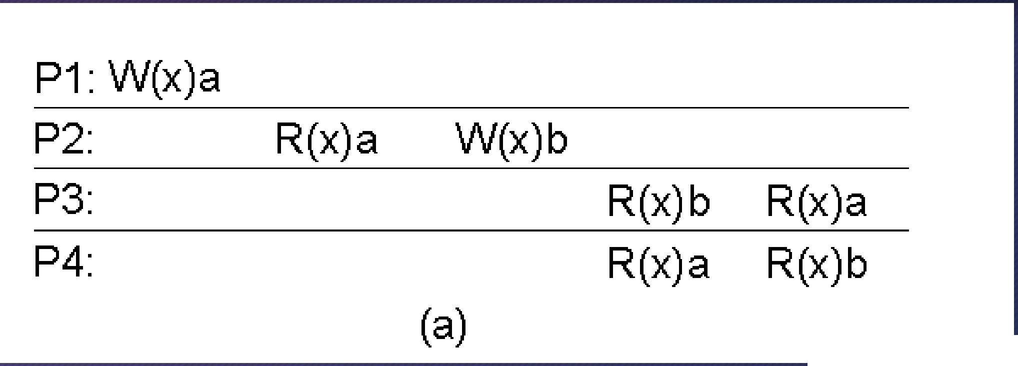


P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Note: it is assumed that W2(x)b and W1(x)c are concurrent.

Causal Consistency Example

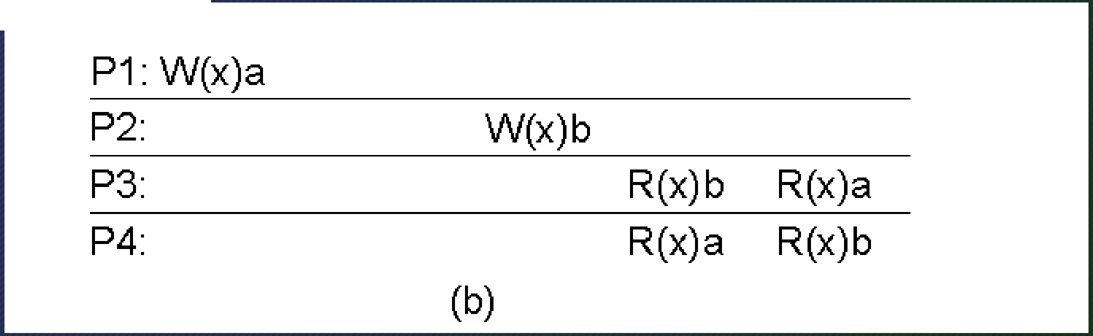


P1:	W(x)a			
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

incorrect

correct



P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

a) Violation of causal-consistency – P2's write is related to P1's write due to the read on 'x' giving 'a' (all processes must see them in the same order).

b) A causally-consistent data-store: the read has been removed, so the two writes are now *concurrent*. The reads by P3 and P4 are now OK.

5. FIFO Consistency

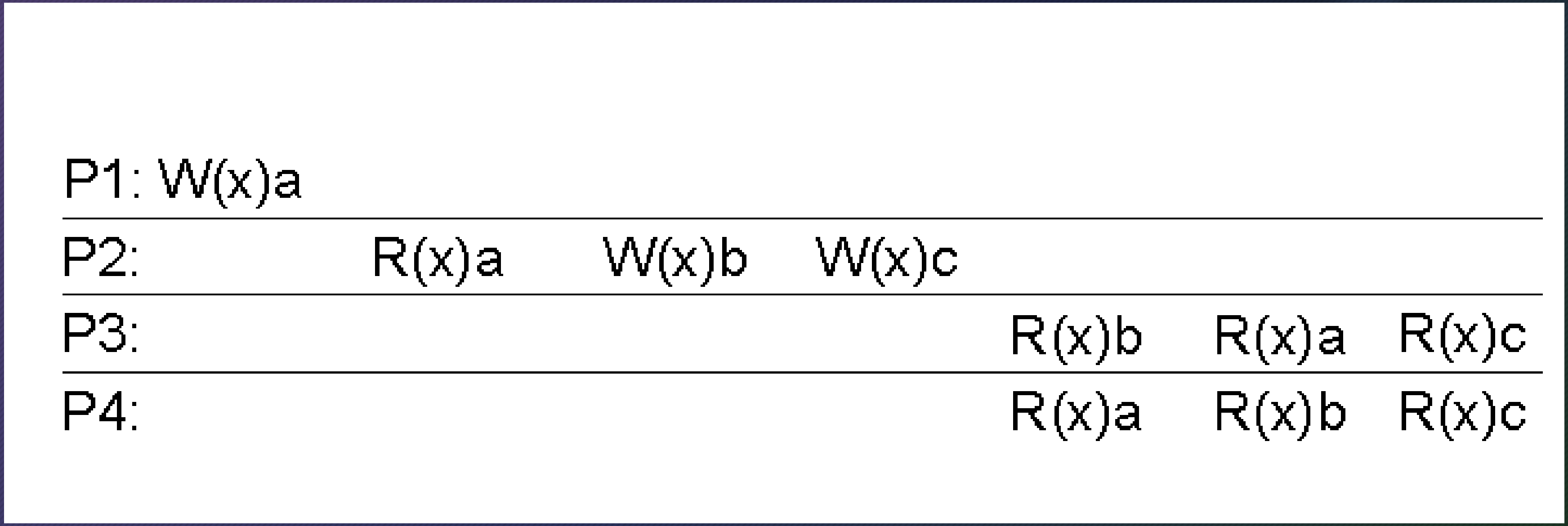
- Definition:

Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

- This is also called “PRAM Consistency” – Pipelined RAM.

- The attractive characteristic of FIFO is that it is easy to implement. There are no guarantees about the order in which different processes see writes – except that two or more writes from a single process must be seen in order.

FIFO Consistency Example



P1:	W(x)a						
P2:		R(x)a	W(x)b	W(x)c			
P3:					R(x)b	R(x)a	R(x)c
P4:					R(x)a	R(x)b	R(x)c

- A valid sequence of FIFO consistency events.
- Note that none of the consistency models studied so far would allow this sequence of events.

6. Weak Consistency

- Not all applications need to see all writes, let alone seeing them in the same order.

- This leads to "Weak Consistency" (which is primarily designed to work with *distributed critical regions*).

- This model introduces the notion of a "synchronization variable", which is used update all copies of the data-store.

Weak Consistency Properties

Three properties of Weak Consistency:

1. Accesses to synchronization variables associated with a data-store are *sequentially consistent*.

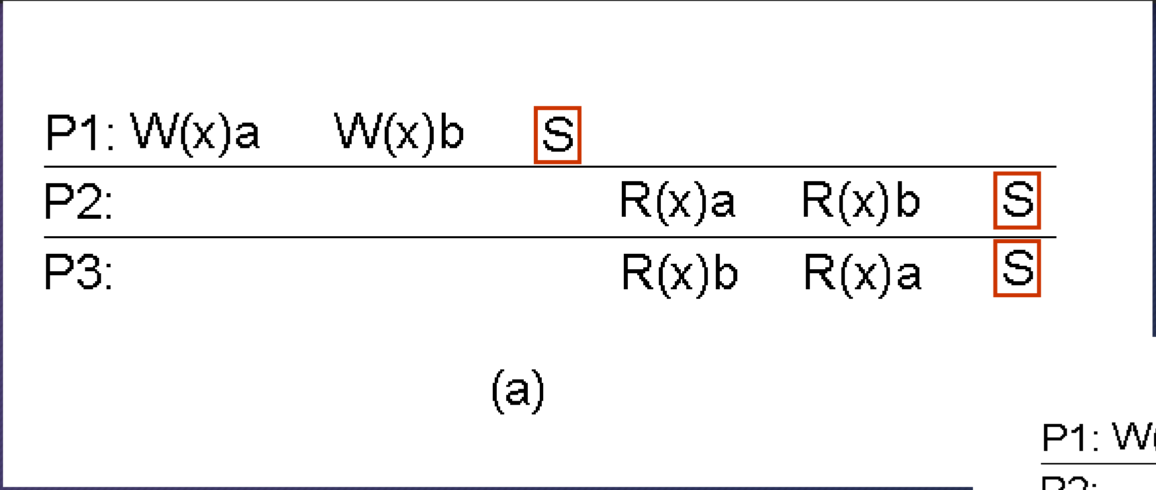
2. No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.

3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

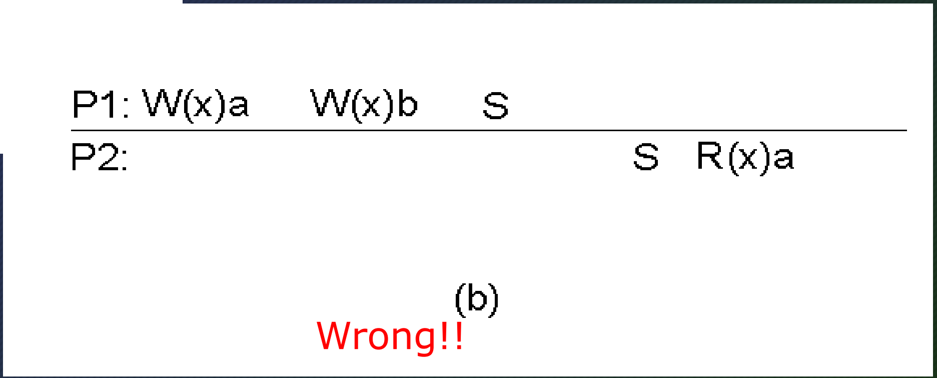
Weak Consistency: What It Means

- By doing a sync., a process can *force* the just written value out to all the other replicas.
- Also, by doing a sync., a process can be *sure* it's getting the most recently written value before it reads.
- In essence, the weak consistency models enforce consistency on a *group of operations*, as opposed to individual reads and writes (as is the case with strict, sequential, causal and FIFO consistency).

Weak Consistency Examples



before sync., any results are acceptable



Wrong!!

a) A valid sequence of events for weak consistency. This is because P2 and P3 have yet to synchronize, so there’s no guarantees about the value in ‘x’.

b) An invalid sequence for weak consistency. P2 has synchronized, so it cannot read ‘a’ from ‘x’ – it should be getting ‘b’.

7. Release Consistency

- Question: how does a weakly consistent data-store know that the sync is the result of a read or a write?
- Answer: It doesn’t!
- It is possible to implement efficiencies if the data-store is able to determine whether the sync is a read or write.
- Two sync variables can be used, “acquire” and “release”, and their use leads to the “Release Consistency” model.

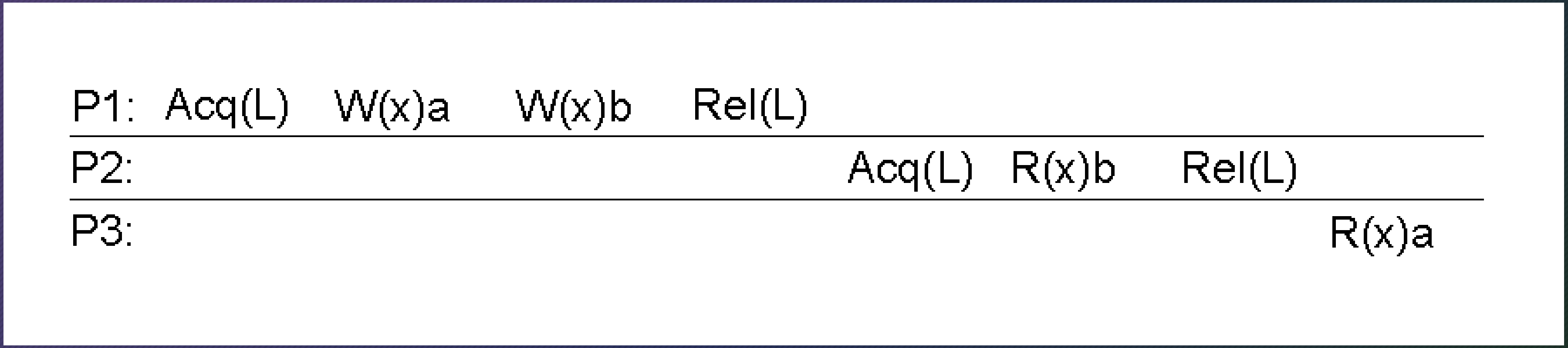
Release Consistency

- Defined as follows:

When a process does an “acquire”, the data-store will ensure that all the local copies of the protected data are brought up to date to be consistent with the remote ones if needs be.

When a “release” is done, protected data that have been changed are propagated out to the local copies of the data-store.

Release Consistency Example



- A valid event sequence for release consistency.
- Process P3 has not performed an *acquire*, so there are no guarantees that the read of 'x' is consistent. The data-store is simply not obligated to provide the correct answer.
- P2 does perform an *acquire*, so its read of 'x' is consistent.

Release Consistency Rules

- A distributed data-store is "Release Consistent" if it obeys the following rules:

1. Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
2. Before a release is allowed to be performed, all previous reads and writes by the process must have completed.
3. Accesses to synchronization variables are *FIFO consistent* (sequential consistency is not required).

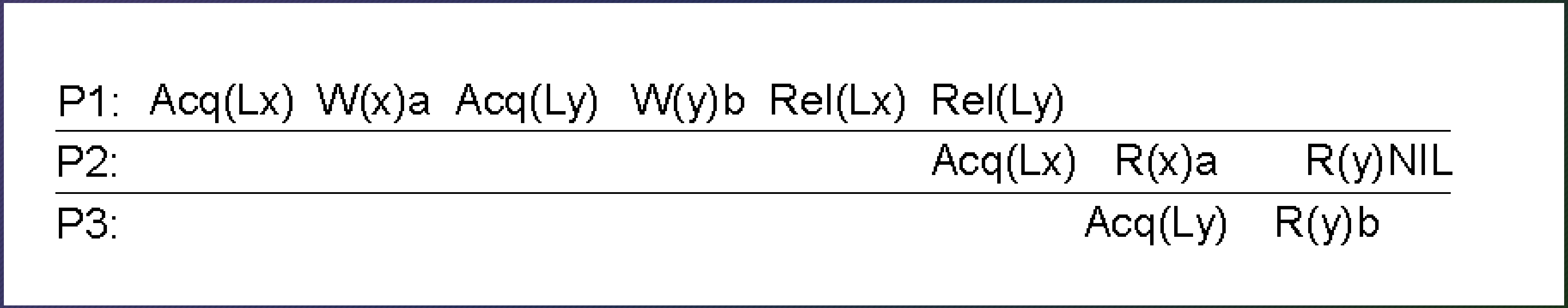
8. Entry Consistency

- A different twist on things is “Entry Consistency”. Acquire and release are still used, and the data-store meets the following conditions:

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Entry Consistency: What It Means

- So, at an *acquire*, all remote changes to guarded data must be brought up to date.
- Before a write to a data item, a process must ensure that no other process is trying to write *at the same time*.


P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y)NIL
P3: Acq(Ly) R(y)b

Locks associate with individual data items, as opposed to the entire data-store. Note: P2’s read on ‘y’ returns NIL as no locks have been requested.

Summary of Consistency Models

a) Consistency models that do not use synchronization operations.

b) Models that do use synchronization operations. (These require additional programming constructs, and allow programmers to treat the data-store *as if it is sequentially consistent*, when in fact it is not. They "should" also offer the best performance).

Consistency	**Description**
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

(a)

Consistency	**Description**
Weak	Shared data can be counted on to be consistent only after a synchronization is done.
Release	Shared data are made consistent when a critical region is exited.
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

Client-Centric Consistency Models

Introduction

Eventual Consistency

- There is a special class of distributed data stores that characterized by the lack of simultaneous updates, most of the operations involve reading data.
- These data stores offers a very weak consistency model, called eventual consistency.

Client-centric consistency models

- Eventual consistent data stores work fine as long as clients always access the same replica.
- However, problems arise when different replicas are accessed over a short period of time.
- Assume the user performs several update operations and then disconnects again.

Client-Centric Consistency Models

- Later, he accesses the database again, possibly after moving to a different location or by using a different access device.
- At that point, the user may be connected to a different replica than before.
- If the updates performed previously have not yet been propagated, the user will notice inconsistent behavior.
- In particular, he would expect to see all previously made changes, but instead, it appears as if nothing at all has happened.
- By introducing special client-centric consistency models, the previous problem can be alleviated in a relatively cheap way.
- In essence, client-centric consistency provides guarantees for a single client.

Client-Centric Consistency Models (Contd…)

- To explain these models, we consider a data store that is physically distributed across multiple machines.
- When a process accesses the data store, it generally connects to the nearest available copy, although any copy will be possible.
- All read and write operations are performed on that local copy.
- Updates are eventually propagated to the other copies.

Client-Centric Consistency models (Contd...)

- To simplify matters, we assume that data items have an associated owner, which is the only process that is permitted to modify that item.
- In this way, we avoid write-write conflicts.
- Let xi[t] denote the version of data item x at local copy Li at time t.
- xi[t] is the result of a series of write operations denoted as WS(xi[t]) at Li that took place since initialization until time t.

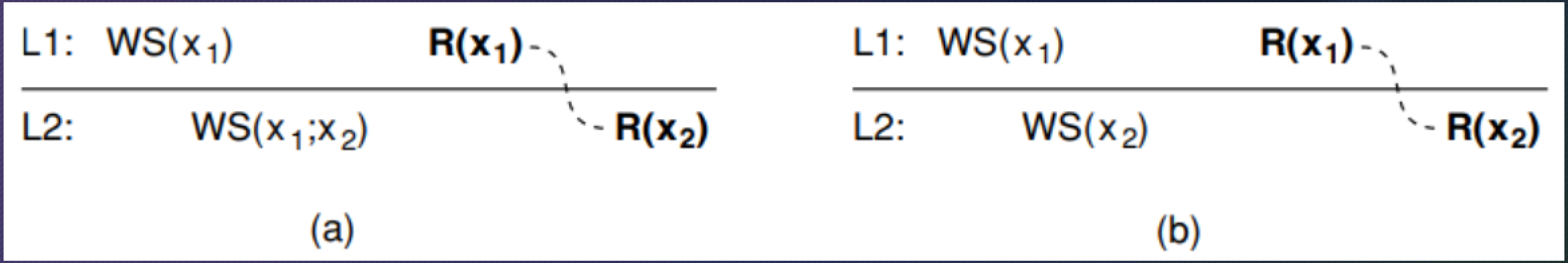
Monotonic Reads

Monotonic Reads

- In a monotonic-read consistent data store, if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

- **Example**: Consider a distributed email database, when a user open mailbox in San Francisco and notice a new email is present. When the user later flies to New York and opens his mailbox again, monotonic-read consistency guarantees that the messages that were in the mailbox in San Francisco will also be in the mailbox when it is opened in New York.

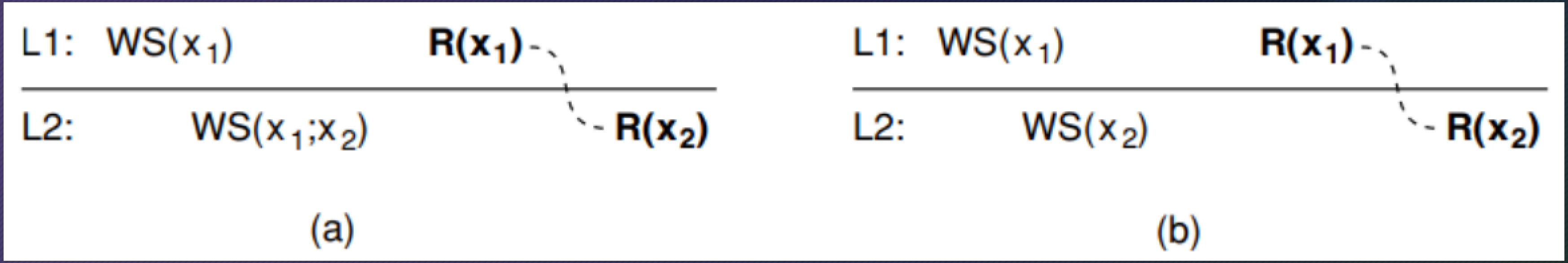
Monotonic Reads (Contd..)



- In the figure (a), process P first performs a read operation on x at L1 shown as R(x1) returning the value of x1. This value results from the write operations WS(x1) performed at L1
- Later, P performs a read operation on x at L2, shown as R(x2).

- To guarantee monotonic-read consistency, all operations in W(x1) should have been propagated to L2 before the second read operation takes place.
- In other words, we need to know for sure that WS(x1) is part of WS(x2) which is expressed as WS(x1;x2).

Monotonic Reads (Contd..)



- In the figure (b), shows a situation in which monotonic-read consistency is not guaranteed.
- After process P has read x1 at L1 it later performs the operation R(x2) at L2.
- However, only the write operations WS(x2) have been performed at L2.
- No guarantees are given that this set also contains all operations contained in WS(x1).

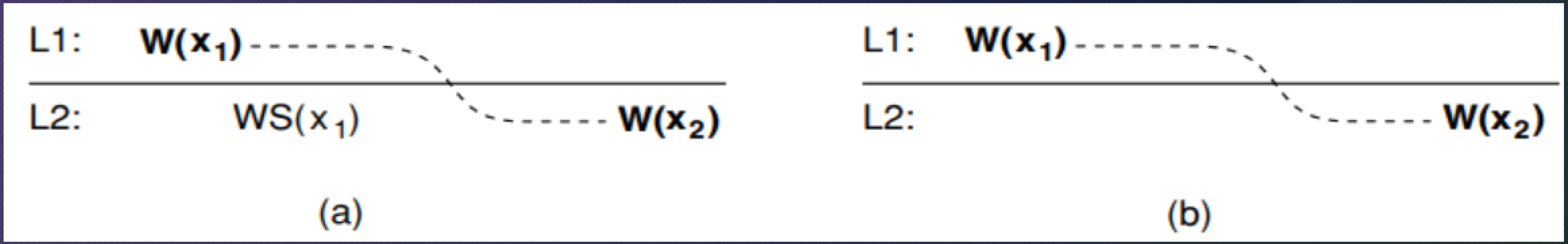
Monotonic Writes

Monotonic Writes:

- In a monotonic-write consistent data store, a write operation by a process on a data item x is completed before any successive write operation on x by the same process.

- In other words, no matter where that operation was initialized, for each node the new write of same process must wait for old write propagate to local copy.

Monotonic Writes (contd…)



- In the figure (a), a process P performs a write operation on x at local copy L1.
- Later, P performs another write operation on x, but this time at L2.
- To ensure monotonic-write consistency, it is necessary that the previous write operation at L1 has already been propagated to L1.
- This is why we have W(x1) at L2 and before W(x2).

- In the figure (b), in which monotonic-write consistency is not guaranteed.
- What is missing is the propagation of W(x1) to copy L2.
- By the definition of monotonic-write consistency, write operations by the same process are performed in the same order as they are initialized.

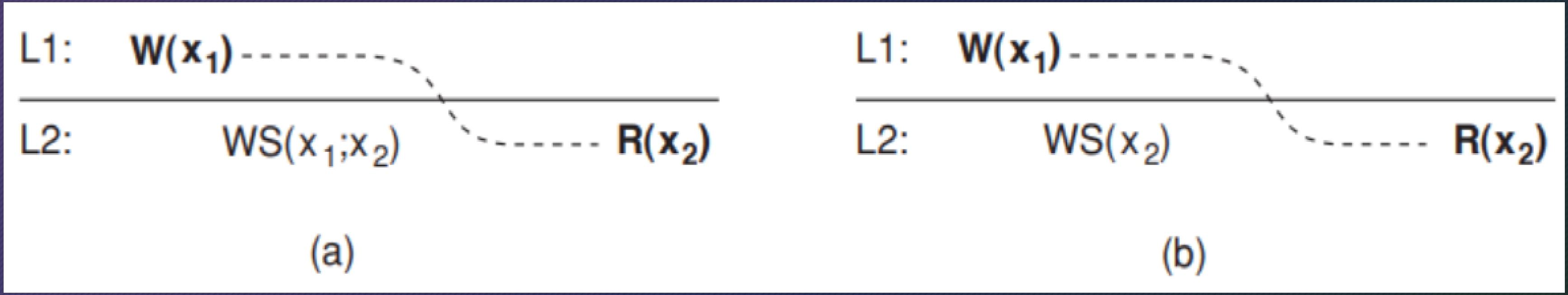
Read Your Writes

Read Your Writes:

- In a read-your-writes consistent data store, the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

- In other words, a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

Read Your Writes (contd...)



- In the figure (a), process P performed a write operation W(x1) and later a read operation at a different local copy.
- Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation.
- This is expressed by W(x1;x2), which states that W(x1) is part of WS(x2).
- In the figure (b) W(x1) has been left out of WS(x2), meaning that the effects of the previous write operation by process P have not been propagated to L2.

Writes Follow Reads

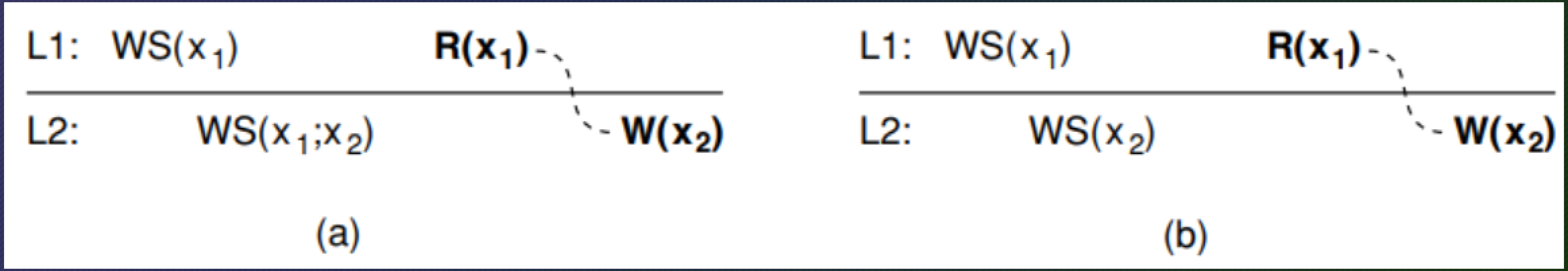
<u>**Writes Follow Reads**</u>

- In a writes-follow-reads consistent data store, a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.
- In other words, any successive write operation by a process on a data item will be performed on a copy of x that is up to date with the value most recently read by that process.

Writes Follow Reads (contd…)

- Writes-follow-reads consistency can be used to guarantee that user can only post a reaction to an article only after they have seen the original article.

- In the figure (a), a process reads x at local copy L1 and the value is also propagates to L2, where the same process later performs a write operation.
- In the figure (b), no guarantees are given that the operation performed at L2, because the write operation is performed on a copy that doesn't receive the update that is on the L1.



Replica Management, Consistency Protocols

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:

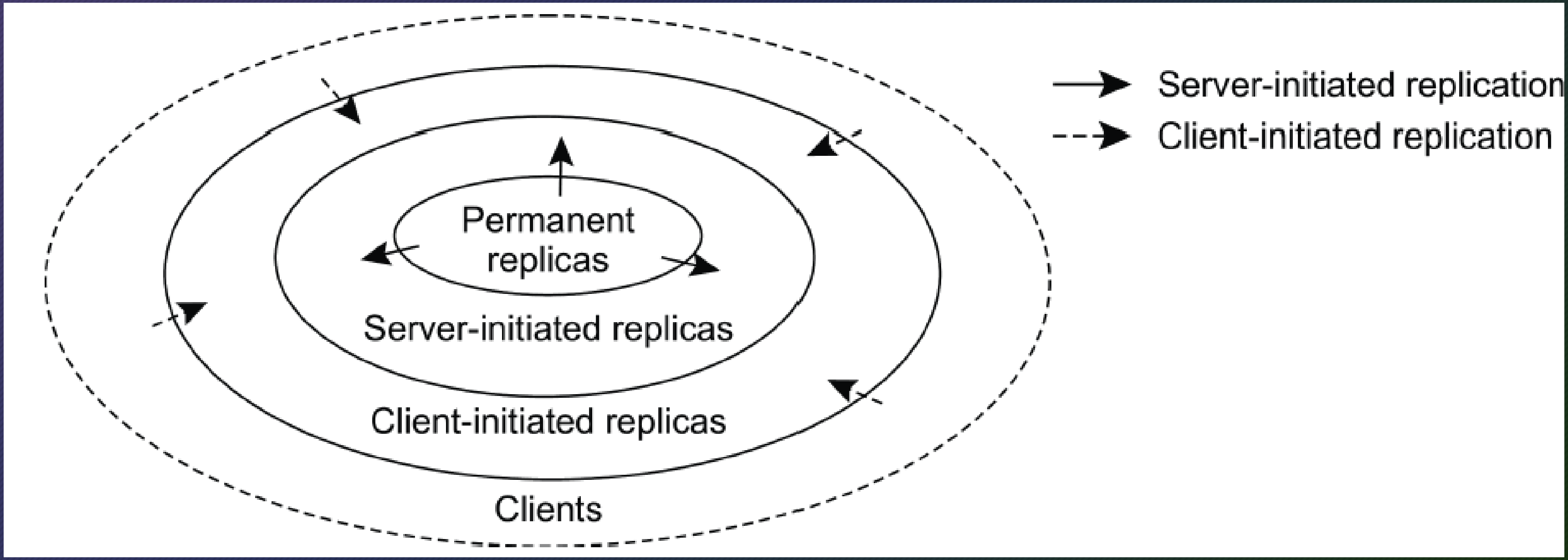
- Permanent replicas: Process/machine always having a replica

- 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)

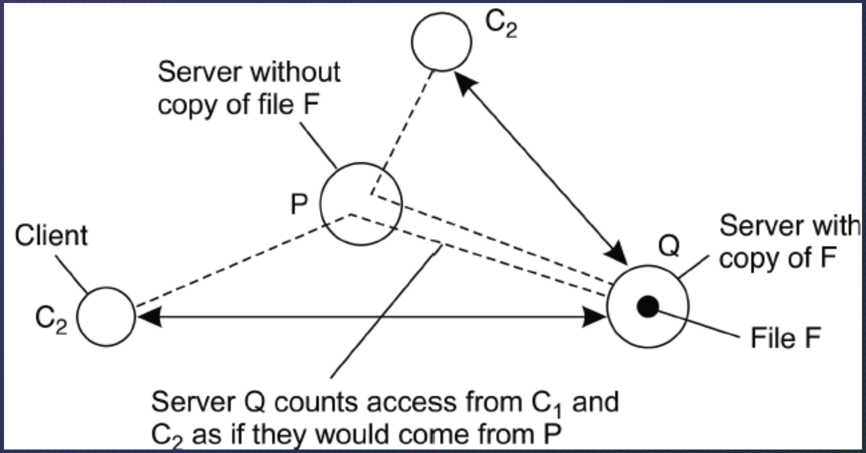
Content Replication

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


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

Server-Initiated Replicas

Counting access requests from different clients



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

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

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**: It is a Server-initiated approach, in which update is propagated regardless whether target asked for it.
- **Pulling updates:** It is a Client-initiated approach, in which client requests to be updated.

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

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.

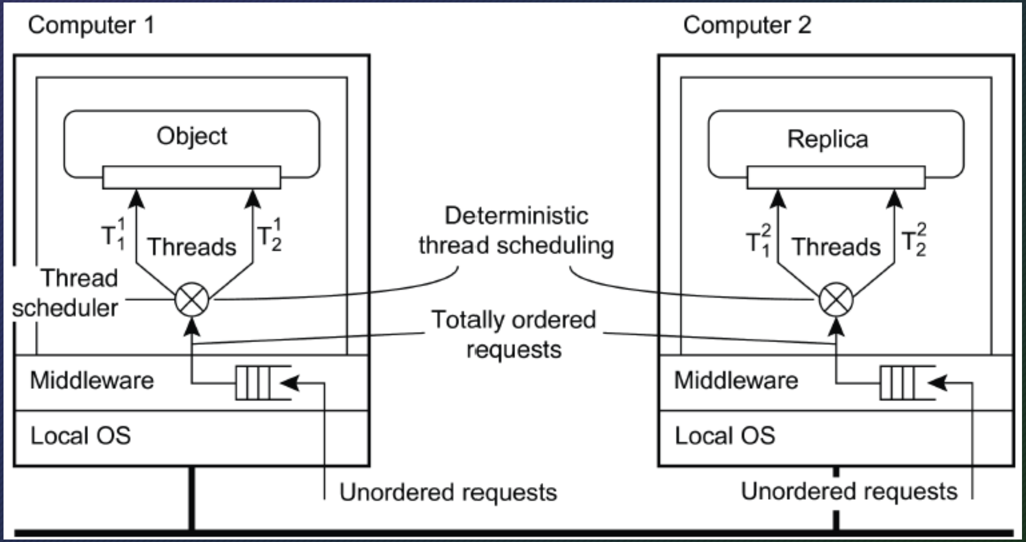
Make lease expiration time adaptive

- 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

Managing Replicated Objects

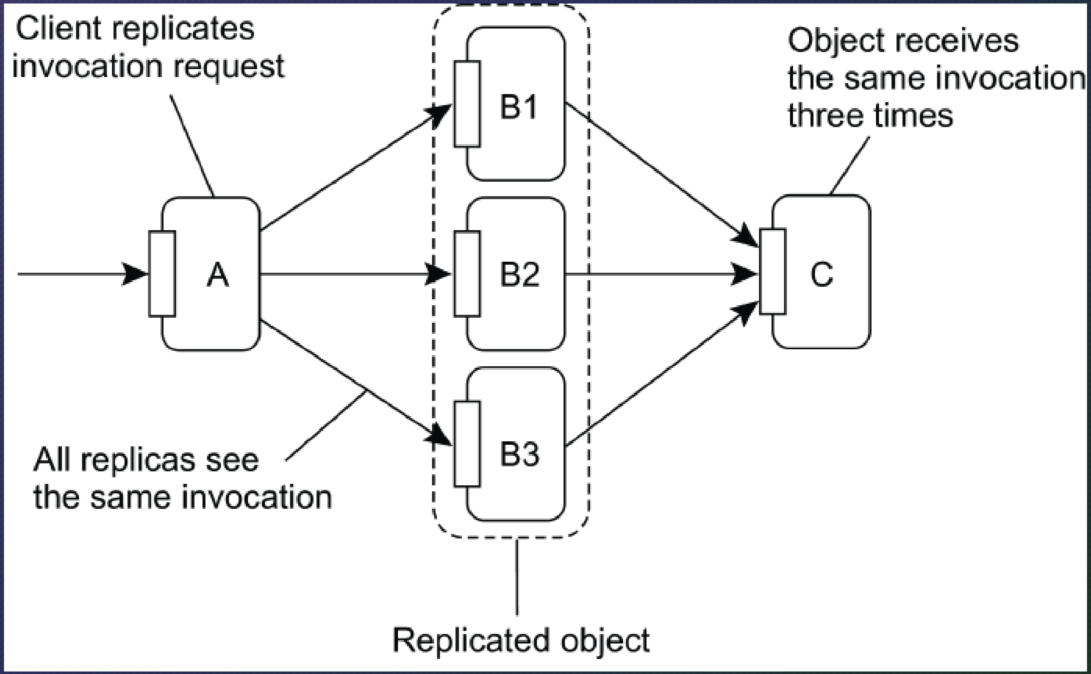
- Prevent concurrent execution of multiple invocations on the same object: access to the internal data of an object has to be serialized. Using local locking mechanisms are sufficient.

- 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.

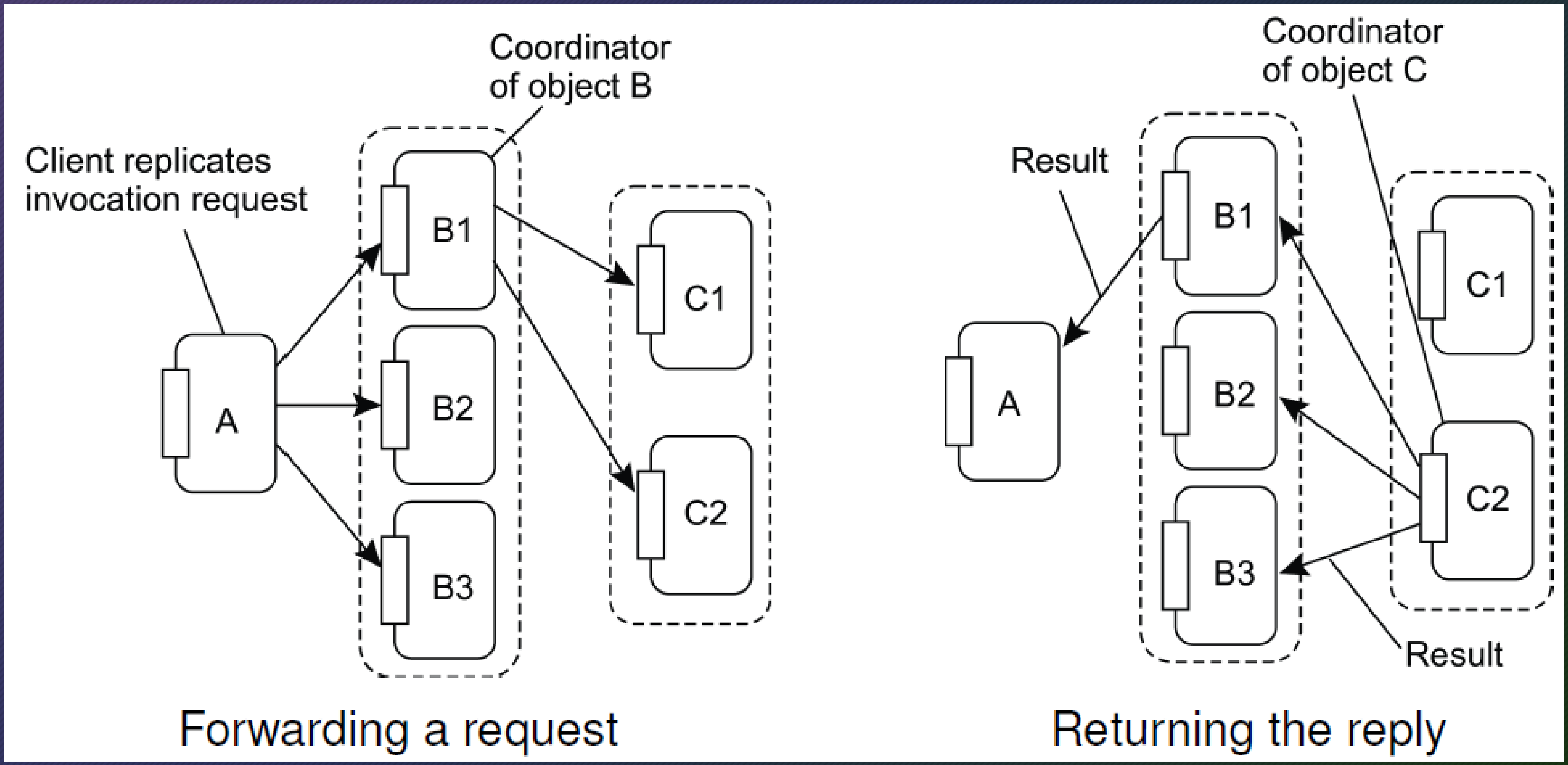


Replicated-Object Invocations

- Problem when invocating a replicated object

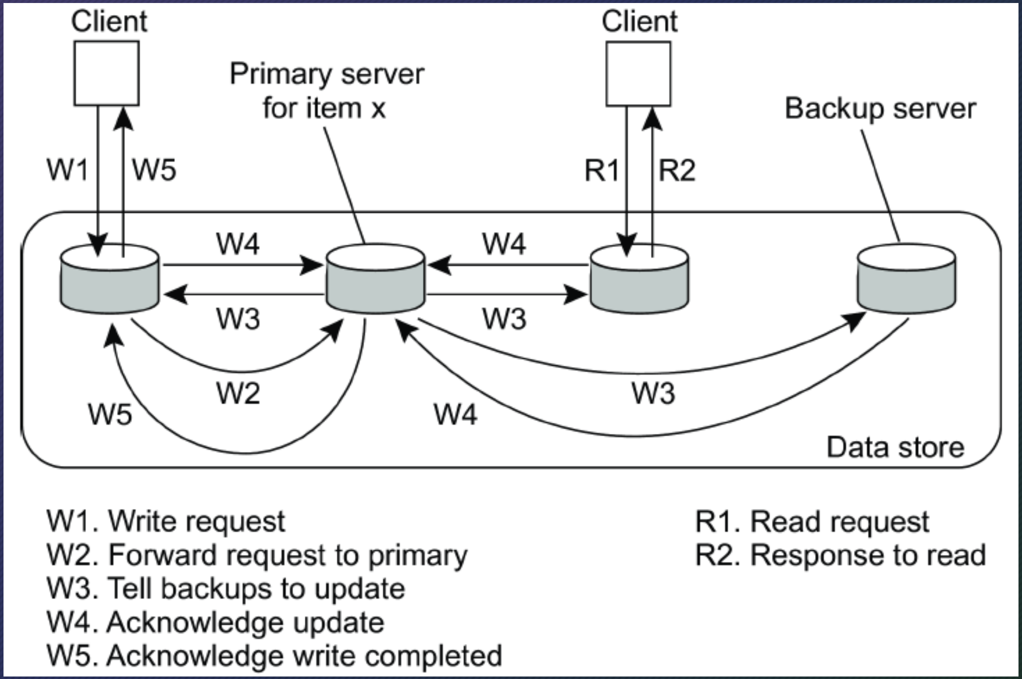

Client replicates invocation request
A
B1
B2
B3
C
Object receives the same invocation three times
All replicas see the same invocation
Replicated object

Replicated-Object Invocations



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

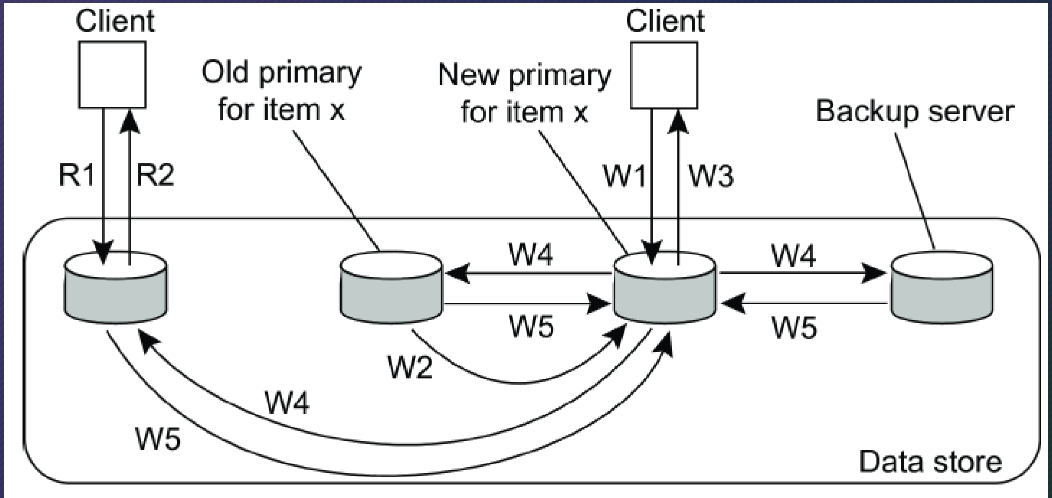
R1. Read request
R2. Response to read

- Example of 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 with Local writes Protocol



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

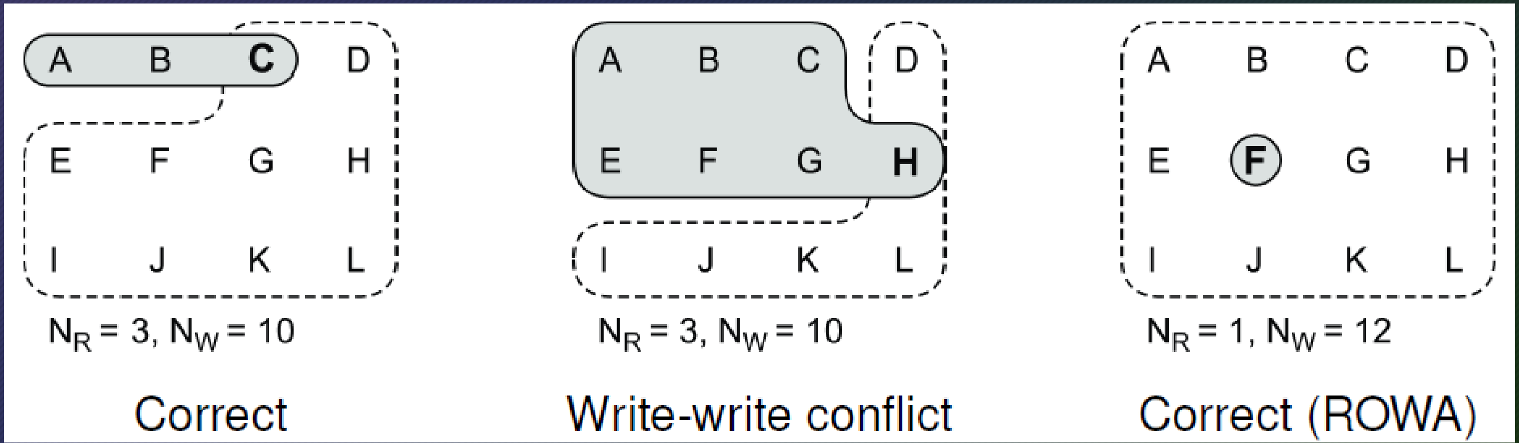
R1. Read request
R2. Response to read

- Example of primary-backup with local writes protocol

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)

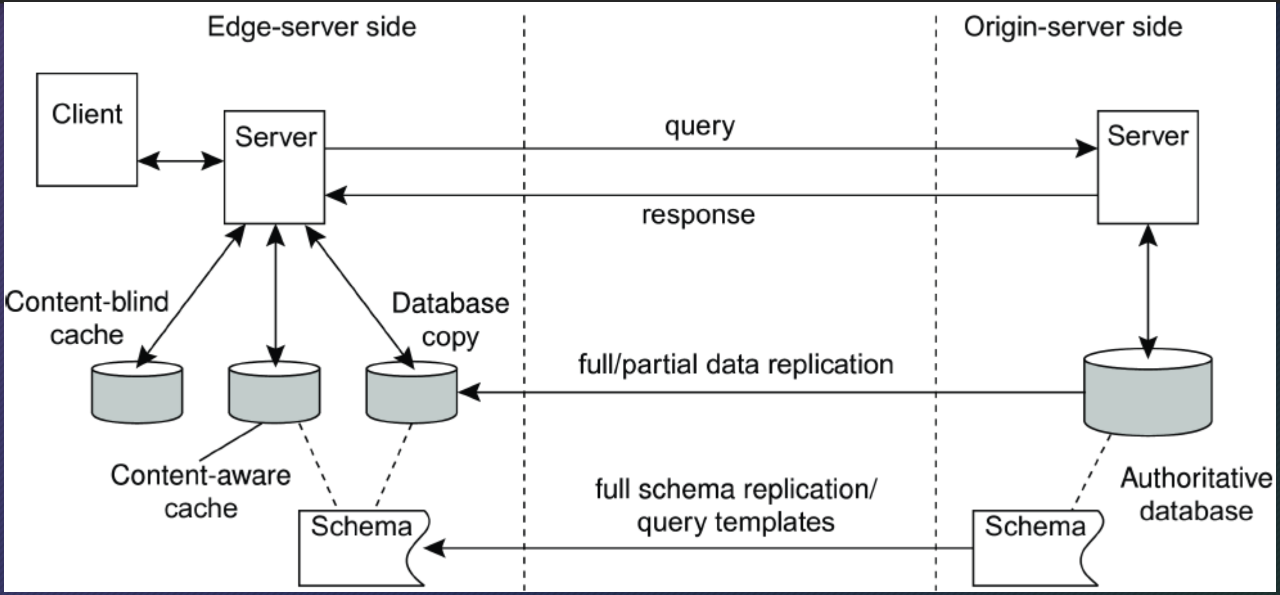


Web-cache Consistency

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.

Fault Tolerance

Fault Tolerance

- Component Faults
- System Failure
- Synchronous Vs Asynchronous Systems
- Use of Redundancy
- Fault Tolerance using Active Replication
- Fault Tolerance using Primary Backup

1. Component Faults

1. Design Error
2. Manufacturing Error
3. Programming Error
4. Harsh Environmental Condition
5. Physical Damage
6. Operator Error
7. Unexpected Inputs

Types of Faults

1. Transient Faults
   - It occurs once and then disappear
   - If the operation is repeated, the fault goes away

   Example: Lost bits on some network because of bird flying through the beam of microwave transmitter

2. Intermittent Faults
   - It occurs, then vanishes of its own accord, then reappears

   Example: Loose contact on a connector

3. Permanent Faults
   - It continues to exist until the faulty component is repaired

   Example: Burnt out chips, software bugs, disk crashes

Faults

1. If some component has a probability p of malfunctioning in a given second of time

2. The probability of it not failing for k consecutive seconds and then failing is $p(1-p)^k$

3. The expected time to failure then given by the formula

$$\text{MTTF} = \sum_{k=1}^{infinite} kp \ (1-p)^{k-1}$$

4. If k= 1, $\sum \alpha^k = \alpha / (1- \alpha)$

Substituting $\alpha = 1-p$, MTTF = 1/p

2. System Failures

System be able to survive component (processor) faults.

Two Types:

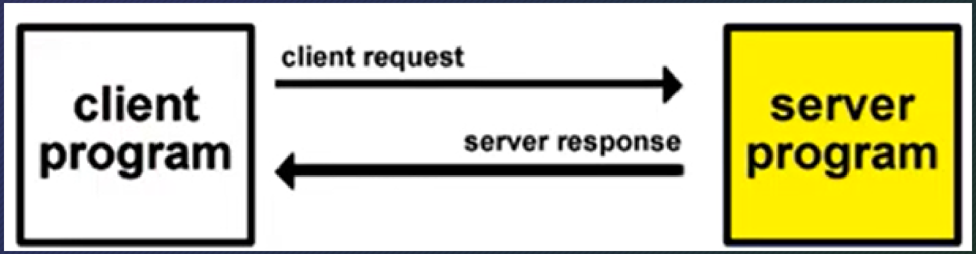
1. Fail-silent Faults/ Fail stop Faults
   - A faulty processor just stops and does not respond to subsequesnt input or produce further output, except perhaps to announce that it is no longer functioning.

2. Byzantine Faults
   - A faulty processor continues to run, issuing wrong answers to questions, and possibly working together maliciously with other faulty processors to give the impression that they are all working correctly when they are not.

3. Synchronous Vs Asynchronous System

- Synchronous System: A system that has the property of always responding to a message within finite bound if it is working

- System not having above property is said to be Asynchronous



4. Use of Redundancy

- Types:

1. Information Redundancy

   Example: Hamming Code

2. Time Redundancy

   Example: Atomic Transaction

3. Physical Redundancy

   Example: Addition of extra processors

Physical Redundancy

- With physical redundancy, extra equipment is added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components

- Two ways to organize these extra processors:

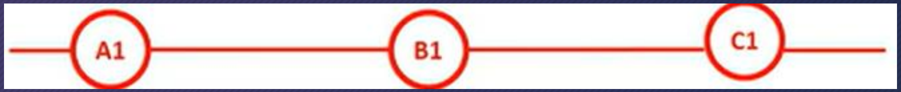
1. Active Replication: When active replication is used, all the processors are used all the time as servers (in parallel) in order to hide faults completely.

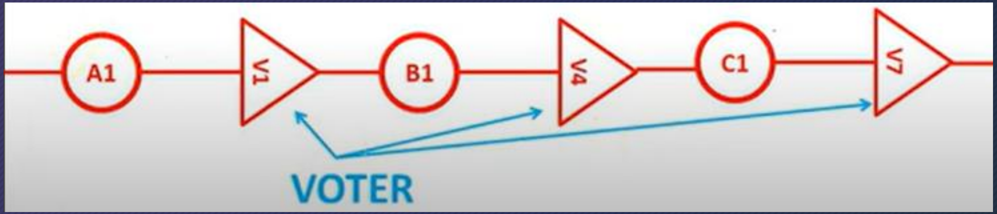
2. Primary Backup: The primary backup scheme just uses one processor as a server, replacing it with a backup if it fails.

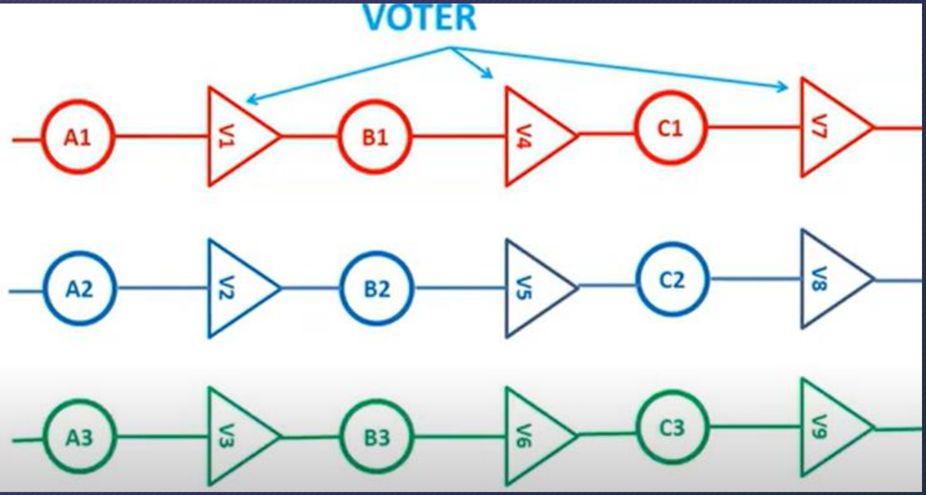
Issues in Active & Primary Replication

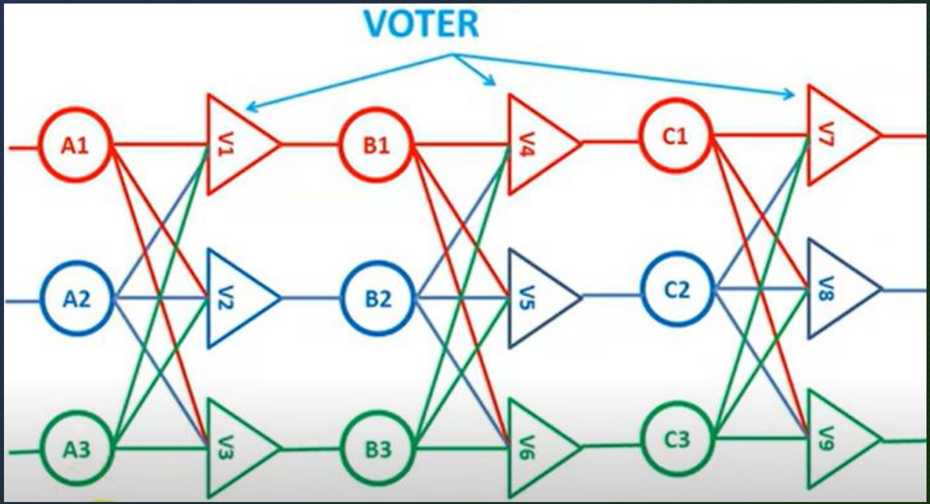
1. The degree of replication required
2. The average and worst-case performance in absence of faults
3. The average and worst-case performance when a fault occurs

Fault Tolerance using Active Replication



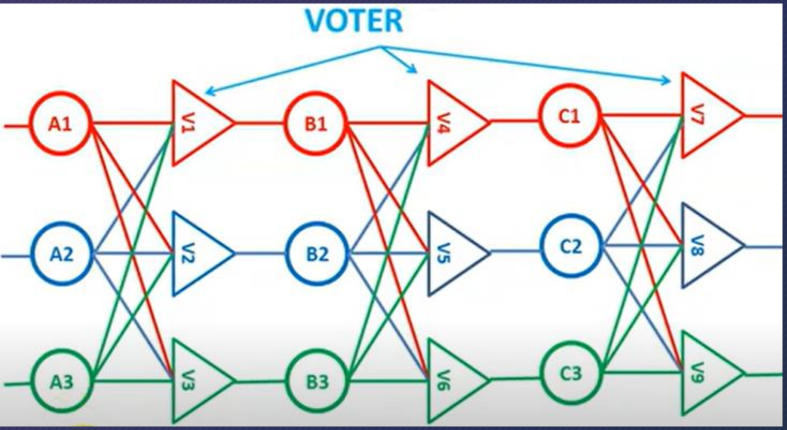


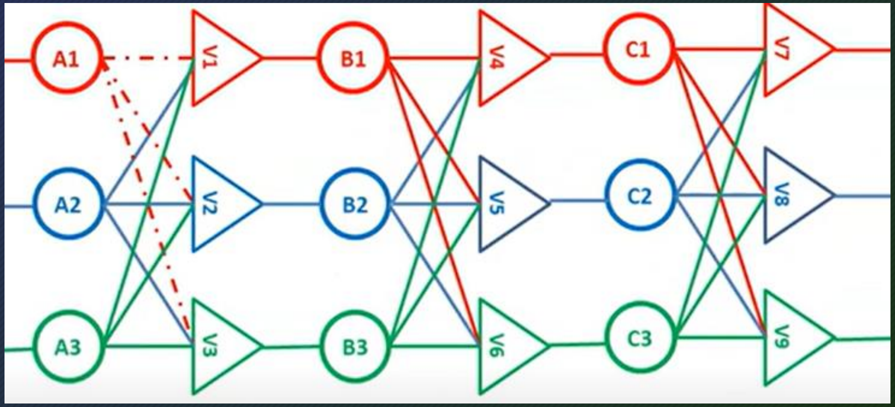




Fault Tolerance using Active Replication

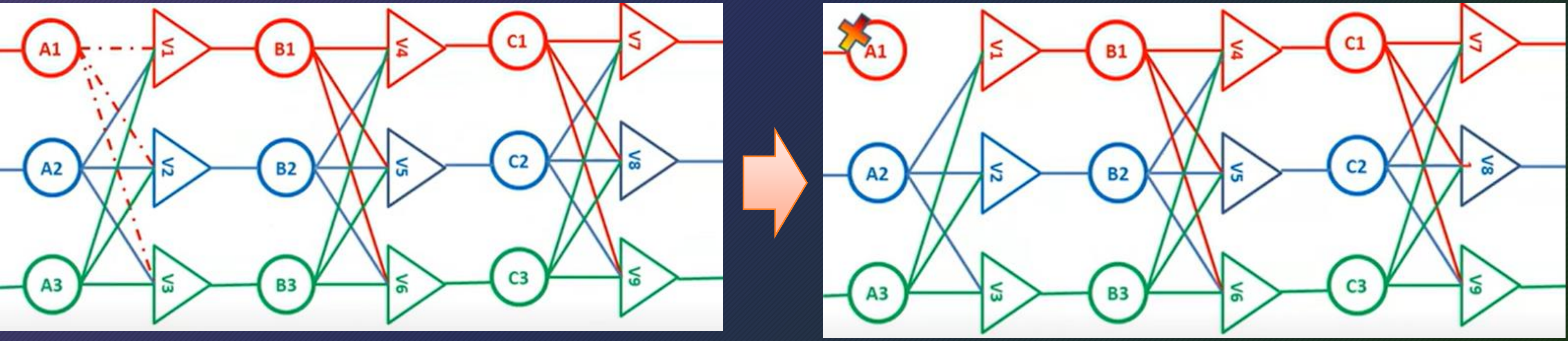
- Suppose element A1 fails
- Each of the voters, V1, V2, V3 gets two good (identical) inputs and one wrong input, and each of them outputs the correct value to the second stage.





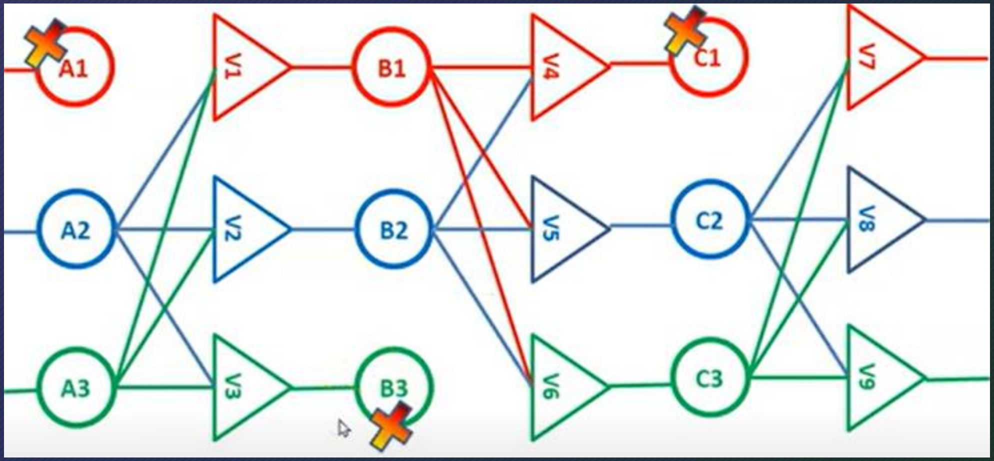
Fault Tolerance using Active Replication

- The effect of A1 failing is completed masked, so that the inputs to B1, B2, and B3 are exactly the same as they would have been had no fault occurred



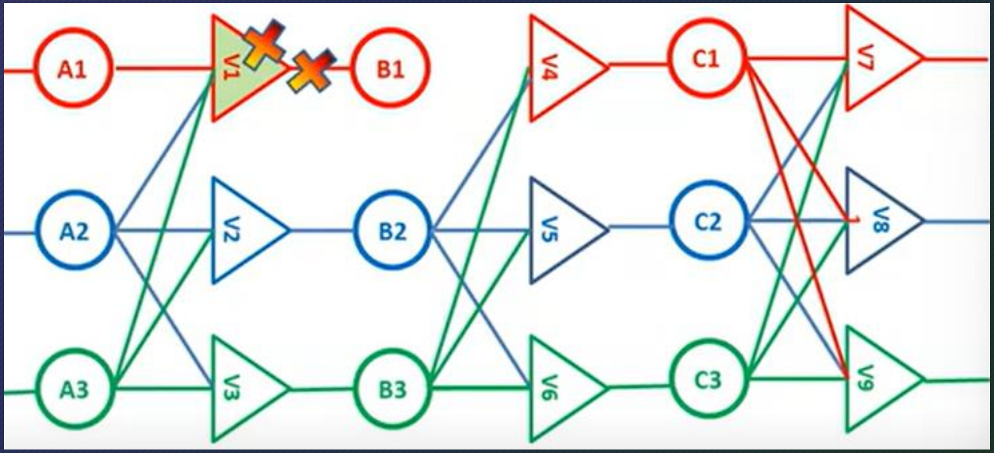
Fault Tolerance using Active Replication

- Now consider what happens if B3 and C1 are also faulty, in addition to A1. These effects are also masked, so the three final outputs are still correct.



Fault Tolerance using Active Replication

- Suppose that V1 malfunctions
- The input to B1 will then be wrong, but as long as everything else works, B2 and B3 will produce the same output and V4, V5, and V6 will all produce the correct result into same stage three.
- A fault in V1 is effectively no different than a fault in B1.



Triple Modular Redundancy

- TMR can be applied recursively, for example, to make a chip highly reliable by using TMR inside it, unknown to the designers who use the chip.

- **How much replication required?**
- It depends on the amount of fault tolerance desired.
- A system is said to be k fault tolerant if it can survive faults in k components and still meet its specifications.
- If the components (processors) fail silently, then having k+1 of them is enough to provide k fault tolerance.
- If k of them simply stop, then the answer from the other one can be used.

Fault Tolerance using Primary Backup

- The essential idea of the primary backup method is that at any one instance, one server is the primary and does all the work. If the primary fails, the backup takes over

Advantages

- It is simpler during normal operation since messages go to just one server (primary), and not to a whole group. The problems associated with ordering these messages also disappear
- In practice, it requires fewer machines because at any instance one primary and one backup is needed

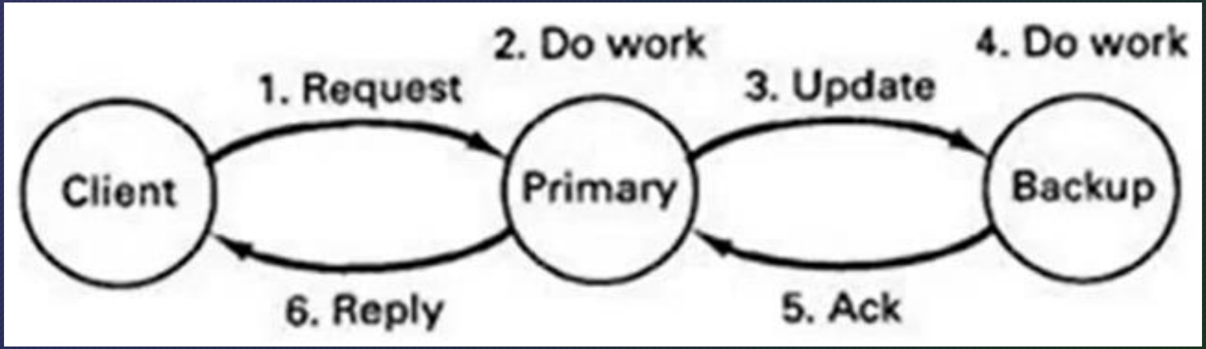
Fault Tolerance using Primary Backup

Disadvantages

- It works poorly in the presence of Bizantine failures in which the primary erroneously claims to be working perfectly.
- Recovery from a primary failure can be complex and time consuming.

Primary Backup Example

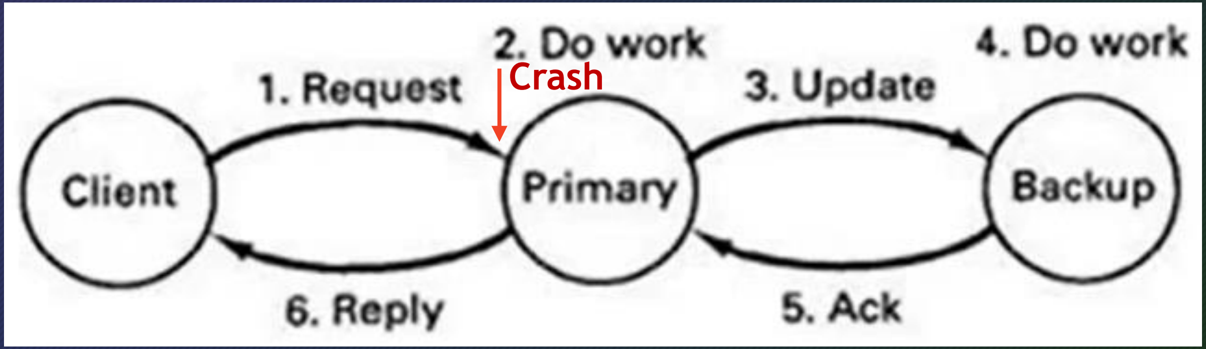
1. The client sends a message to the primary
2. Primary does the work and then sends an update message to the backup
3. When the backup gets the message, it does the work and then sends an acknowledgement back to the primary
4. When the acknowledgement arrives, the primary sends the reply to the client



Effect of Primary Crash during RPC

1. Primary crashes before doing the work (step 2)

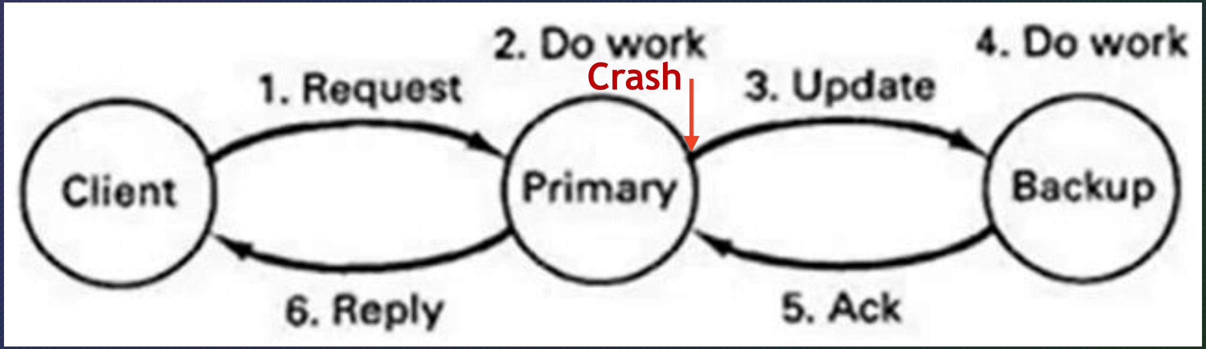
- No harm is done
- The client will time out and retry
- If it tries often enough, it will eventually get the backup and the work will be done exactly once



Effect of Primary Crash during RPC

1. Primary crashes after doing the work but before sending update (after step 2)

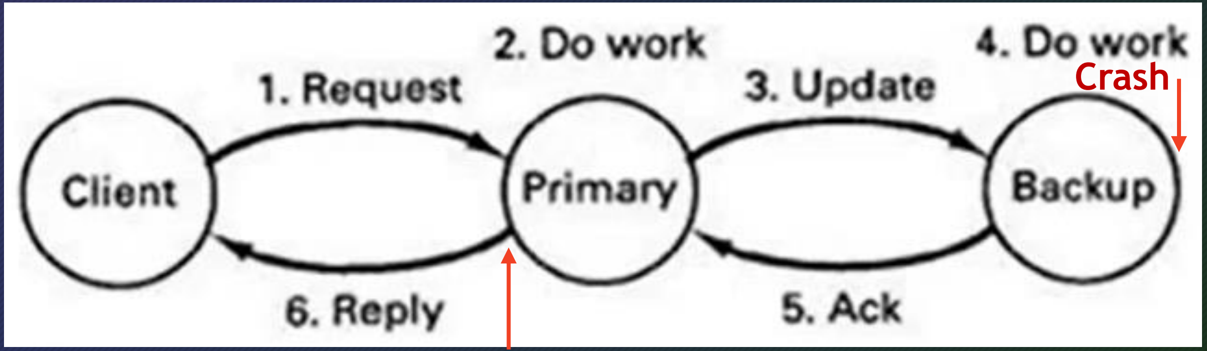
- When the backup takes over and the request comes in again
- The work will be done a second time



Effect of Primary Crash during RPC

1. **Primary crashes after step 4 but before step 6**

- The work may end up being done three times, once by the primary, once by the backup as a result of step 3, and once after the backup becomes the primary.
- If requests carry identifiers, it may be possible to ensure that the work is done only twice, but getting it done exactly once is difficult to impossible



Practical Problem with Primary Backup Approach

Scenario 1:

When to cut over from primary to the backup?

- The backup could send “Are You Alive?” message periodically to the primary.
- If the primary fails to respond within a certain time, the backup would take over

Practical Problem with Primary Backup Approach

Scenario 2:

What if the primary has not crashed, but is merely slow?

- There is no way to distinguish between a slow primary and one that has gone down
- When the backup takes over, the primary really stops trying to act like the primary
- Ideally the backup and primary should have a protocol to discuss this, but it is hard to negotiate with the dead
- The best solution is a hardware mechanism in which the backup can forcibly stop or reboot the primary

Variant of Primary Backup Approach

Dual-Ported Disk

- A dual-ported disk shared is between the primary and secondary.
- In this configuration, when the primary gets a request, it writes the request to disk before doing any work and also writes the results to disk
- No messages to or from the backup are needed.
- If the primary crashes, the backup can see the state of the world by reading the disk.

Disadvantage:

There is only one disk, so if that fails, everything is lost. At the cost of extra equipment and performance, the disk could also be replicated and all writes could be done to both disks

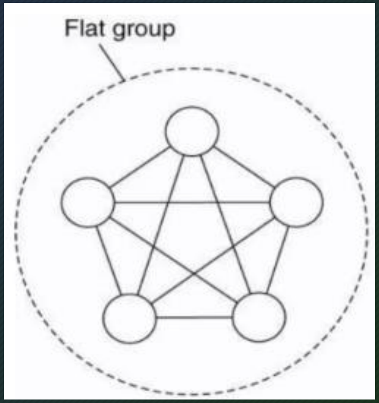
Process Resilience & Distributed Commit

Process Resilience

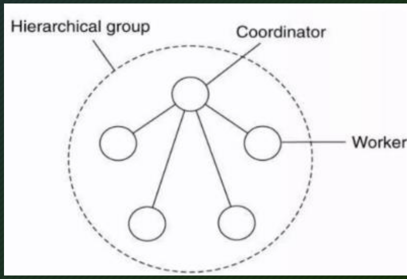
- Process resilience in a distributed system is the ability of the system to automatically adapt and continue functioning when adverse situations occur.
- Faults: It is an incorrect internal state in your system
- Example: Database slowdown, memory leaks, blocked thread, bad data etc.
- Failure: Inability of the system to do it’s intended job
- Resilience: It is about preventing fault turning into failure
- Process can be made fault tolerant by arranging to have a group of processes with each member of the group being identical
- A message sent to the group is delivered to all the copies of the processes (the group members), and then only one of them perform the required service

Process Resilience (contd…)

- If any one process fails, it is assumed that another/other will still be able to functional & provide service
- Flat Group
  - All the processes are equal, decisions are made collectively
  - No single point of failure
  - Decision making is complicated as consensus is required

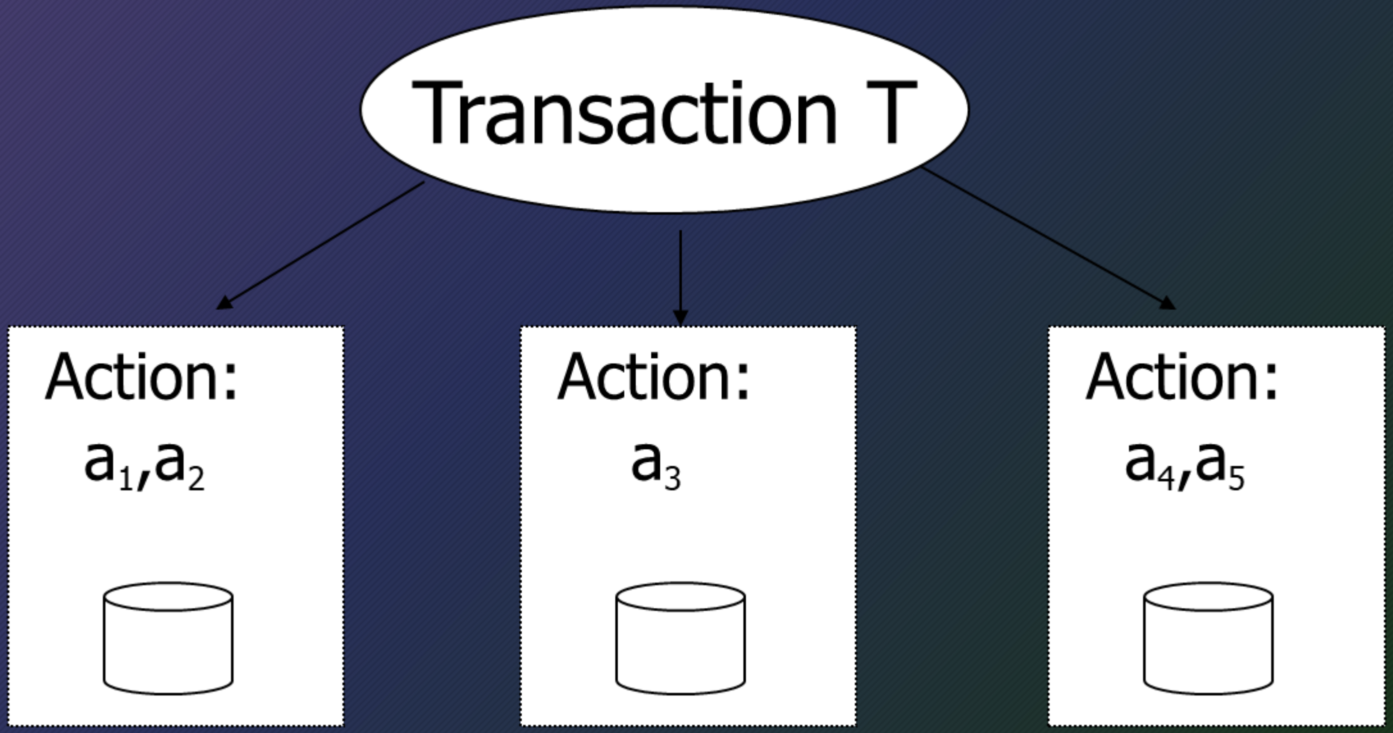


- Hierarchical Group
  - One of the processes is elected to be the coordinator, which selects another process (a worker) to perform the operations
  - There is a chance of single point of failure
  - Decisions are made easily and quickly by coordinator without any consensus



Distributed Commit

Commit must be Atomic



1-Phase Commit (1PC)

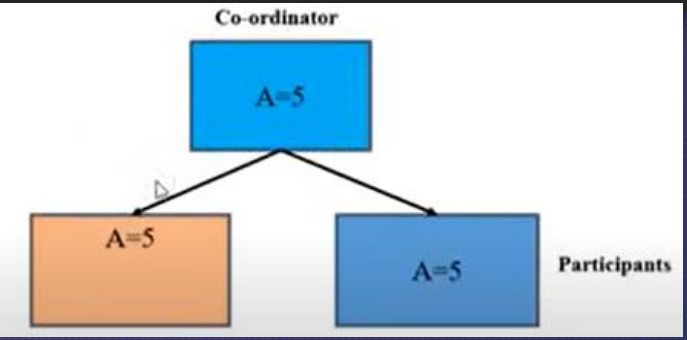
Distributed commit is often established by means of a coordinator.

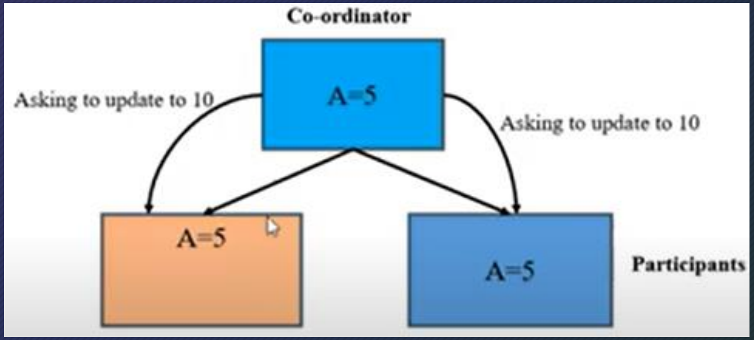
One-phase commit protocol: It is a simple scheme. The coordinator tells all other processes that are also involved, called *participants*, whether or not to (locally) perform the operation in question.

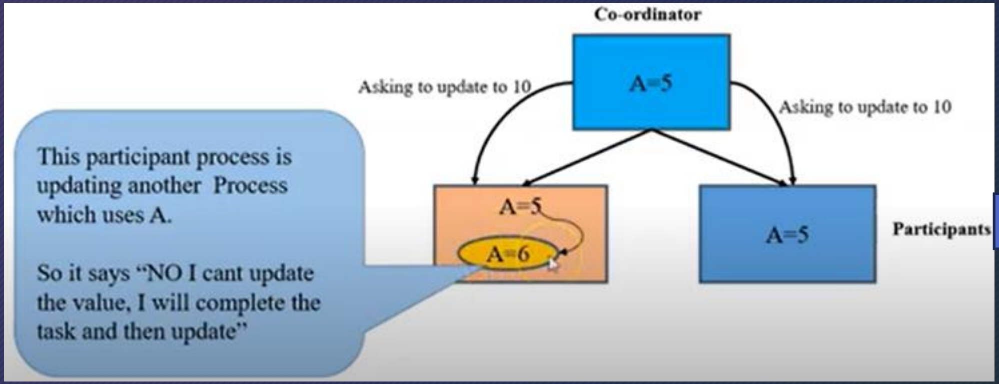
Drawback: If one of the participants cannot actually perform the operation, there is no way to tell the coordinator.

Example: In case of distributed transactions, a local commit may not be possible because this would violate concurrency control constraints.

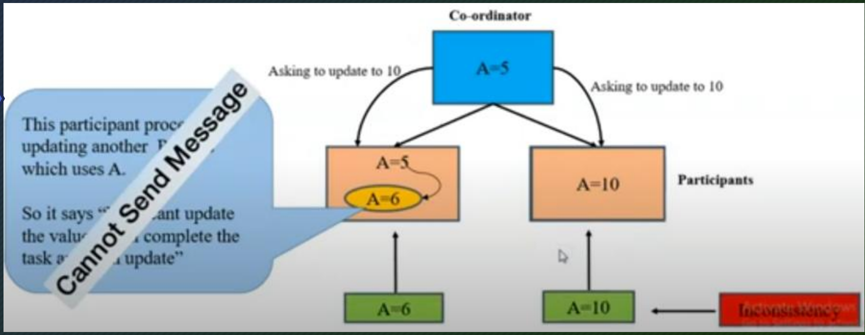
1-Phase Commit (1PC): Example






Co-ordinator
Asking to update to 10
A=5
Asking to update to 10
This participant process is updating another Process which uses A.

So it says "NO I cant update the value, I will complete the task and then update"
A=5
A=6
A=5
Participants



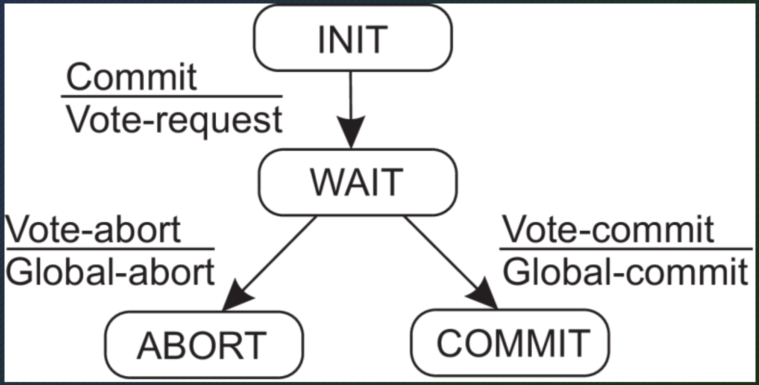
2-Phase Commit (2PC)

The protocol consists of the following two phases, each consisting of two steps

Phase 1:

1. The coordinator *sends a vote-request message* to all participants.

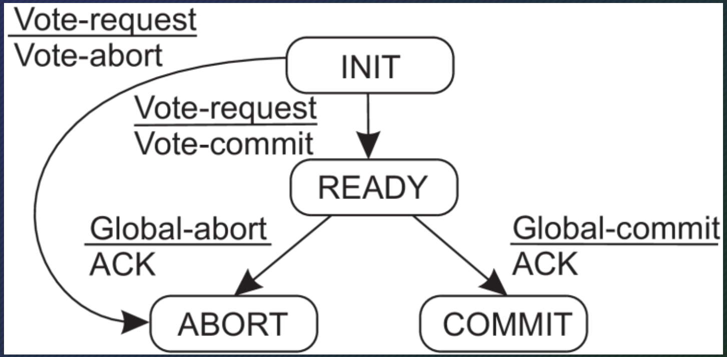
2. When a participant receives a vote-request message, it *returns either a vote-commit message* to the coordinator telling the coordinator that it is prepared to locally commit its part of the transaction, or otherwise a *vote-abort message*.


$$\begin{tikzcd}
 & \boxed{\text{INIT}} \arrow[d, "\frac{\text{Commit}}{\text{Vote-request}}"'] & \\
 & \boxed{\text{WAIT}} \arrow[dl, "\frac{\text{Vote-abort}}{\text{Global-abort}}"'] \arrow[dr, "\frac{\text{Vote-commit}}{\text{Global-commit}}"] & \\
\boxed{\text{ABORT}} & & \boxed{\text{COMMIT}}
\end{tikzcd}$$

2-Phase Commit (2PC)

Phase 2:

1. The coordinator collects all votes from the participants. If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a *global-commit message* to all participants.



However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a *global-abort message*.

2. Each participant that voted for a commit waits for the final reaction by the coordinator. If a participant receives a *global-commit message*, it locally commits the transaction.

Otherwise, when receiving a *global-abort message*, the transaction is locally aborted as well.

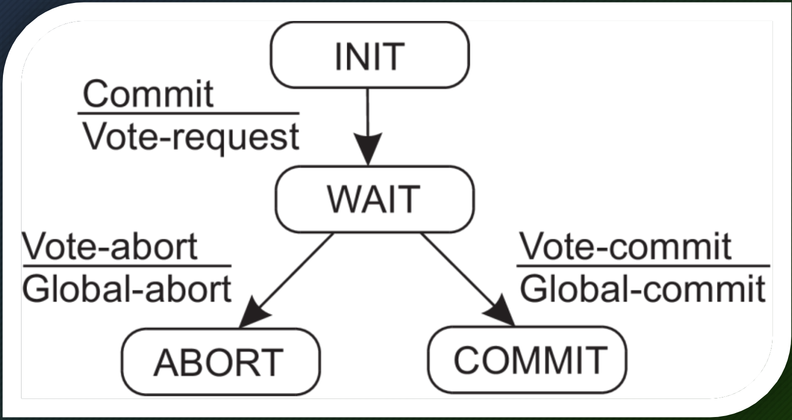
Problems with 2PC

- Several problems arise when this basic 2PC protocol is used in a distributed system where failures occur
- There are a total of *three states* in which either a coordinator or participant is blocked waiting for an incoming message

1. A participant may be waiting in its INIT state for a *vote-request message* from the coordinator.

2. The coordinator can be blocked in state WAIT, waiting for the votes of each participant.

3. A participant can be blocked in state READY, waiting for the global vote as sent by the coordinator.

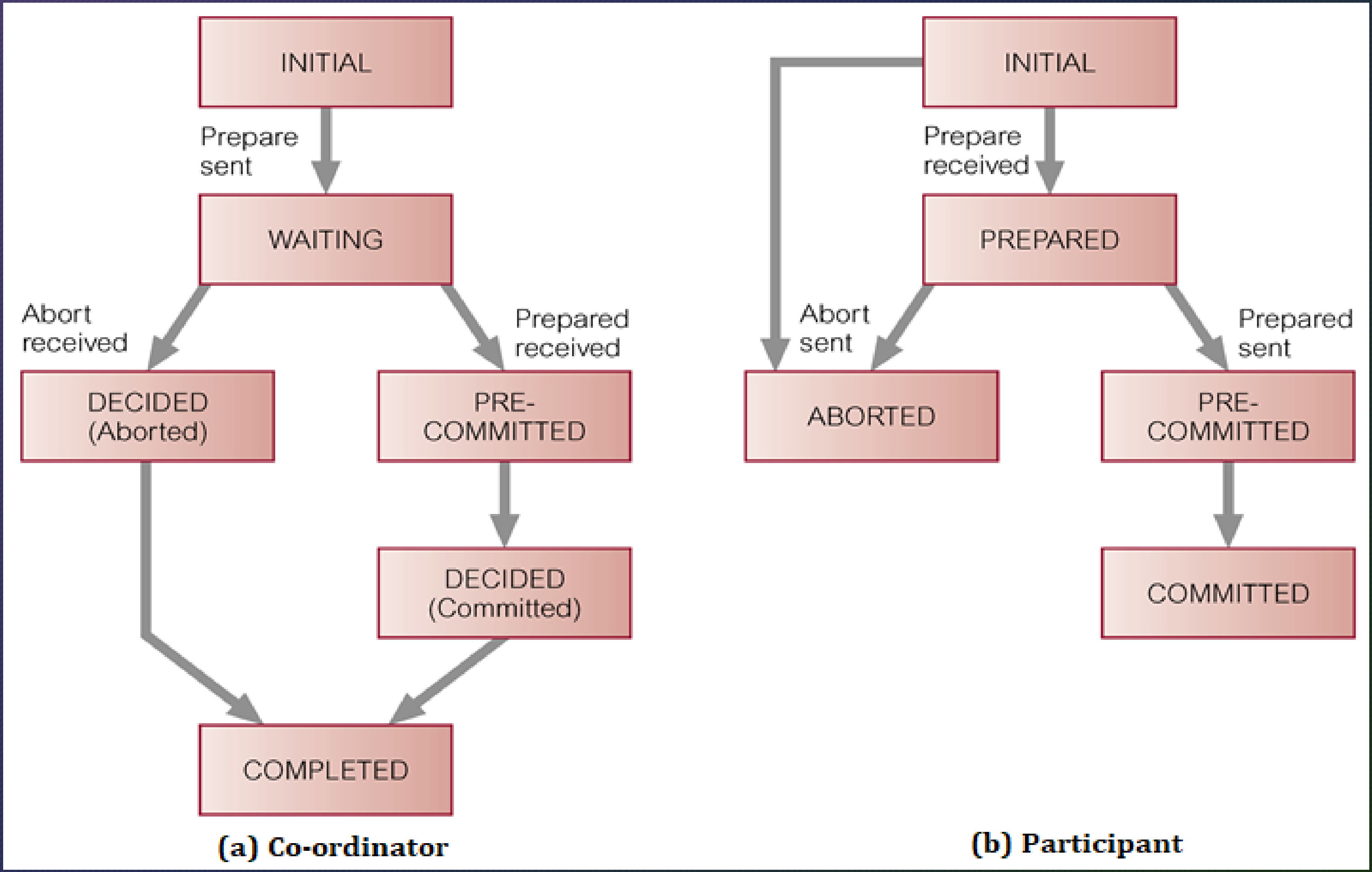


3-Phase Commit (3PC)

- 3PC is non-blocking.
- Non-blocking for site failures, except in event of failure of all sites.
- 3PC removes uncertainty period for participants who have voted commit and await global decision

- 3PC introduces third phase, called *pre-commit*, between voting and global decision.
- On receiving all votes from participants, coordinator sends global pre-commit message.
- Participant who receives global pre-commit, knows all other participants have voted commit and that, in time, participant itself will definitely commit.

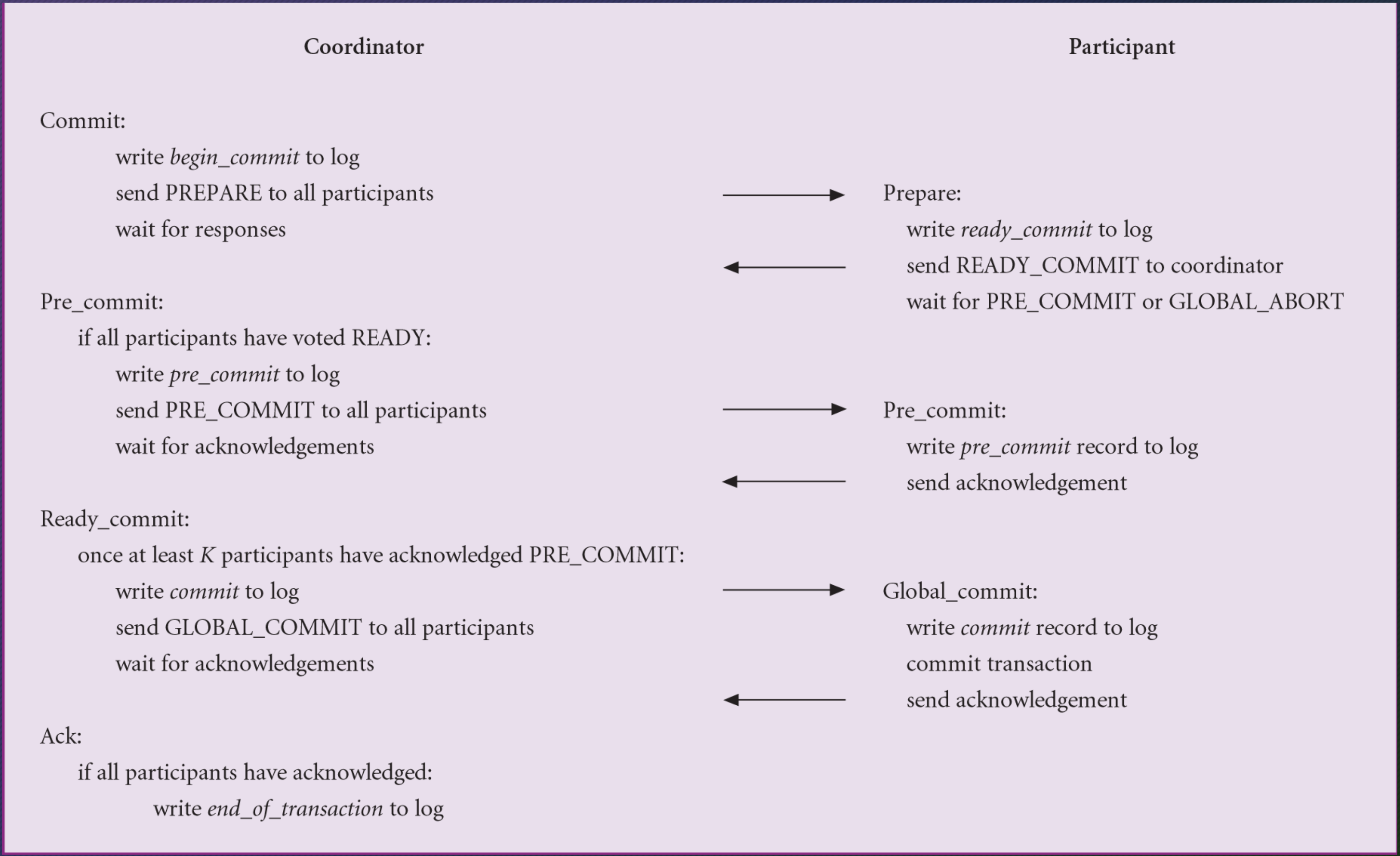
3-Phase Commit (3PC)



(a) Co-ordinator

(b) Participant

3-Phase Commit (3PC)



```
                 Coordinator                                                        Participant

Commit:
        write begin_commit to log
        send PREPARE to all participants                         →     Prepare:
        wait for responses                                                 write ready_commit to log
                                                                  ←        send READY_COMMIT to coordinator
Pre_commit:                                                                wait for PRE_COMMIT or GLOBAL_ABORT
    if all participants have voted READY:
        write pre_commit to log
        send PRE_COMMIT to all participants                      →     Pre_commit:
        wait for acknowledgements                                          write pre_commit record to log
                                                                  ←        send acknowledgement
Ready_commit:
    once at least K participants have acknowledged PRE_COMMIT:
        write commit to log                                      →     Global_commit:
        send GLOBAL_COMMIT to all participants                             write commit record to log
        wait for acknowledgements                                          commit transaction
                                                                  ←        send acknowledgement
Ack:
    if all participants have acknowledged:
            write end_of_transaction to log
```

Problems with 3PC

- Several problems arise when 3PC protocol is used in a distributed system where failures occur

- <u>**Coordinator or Participants**</u>

✓ Timeout in WAITING state
- Same as 2PC. Globally abort transaction.

✓ Timeout in PRE-COMMITTED state
- Write commit record to log and send GLOBAL-COMMIT message.

✓ Timeout in DECIDED state
- Same as 2PC. Send global decision again to sites that have not acknowledged.

3PC Recovery Protocols

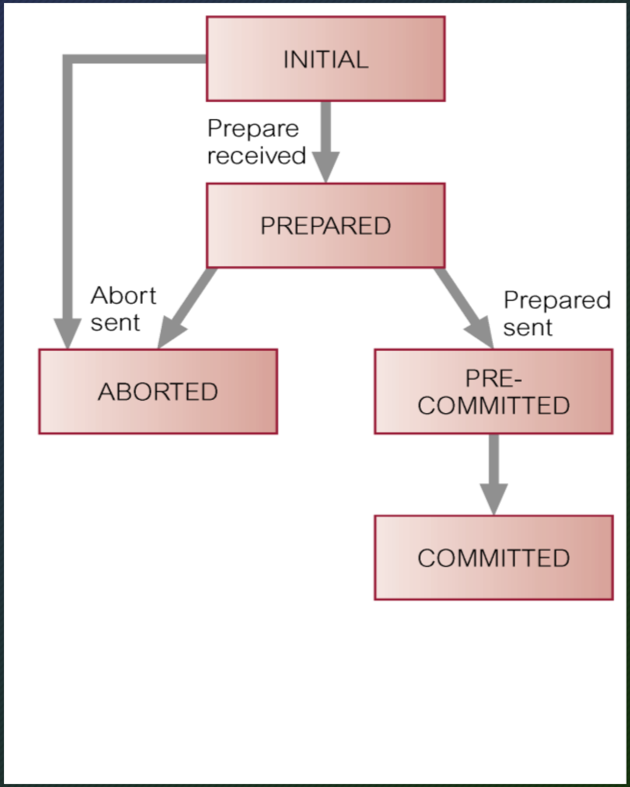
- <u>Coordinator Fails</u>

✓ Failure in INITIAL state
  - Recovery starts commit procedure.

✓ Failure in WAITING state
  - Contact other sites to determine fate of transaction.

✓ Failure in PRE-COMMITTED state
  - Contact other sites to determine fate of transaction.

✓ Failure in DECIDED state
  - If all acknowledgements in, complete transaction; otherwise initiate termination protocol above.



3PC Recovery Protocols (contd…)

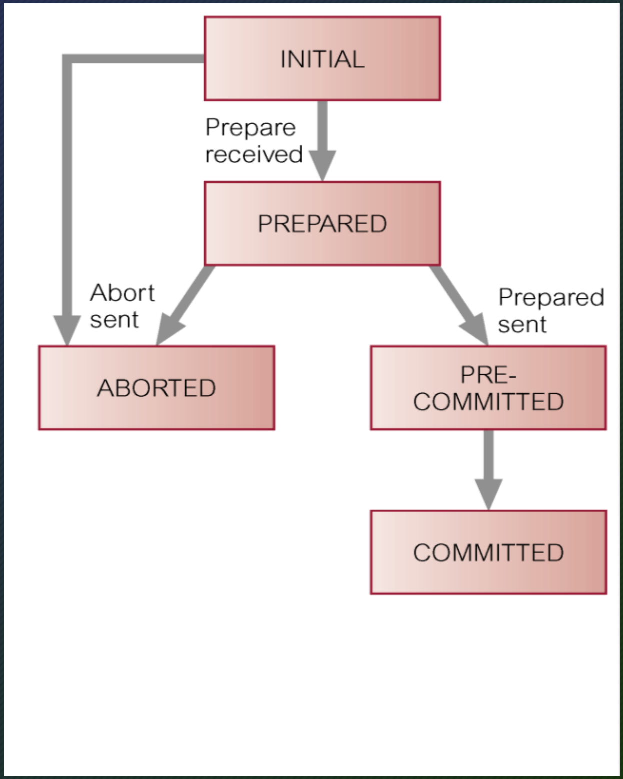
- **Participant Fails**

✓ Failure in INITIAL state
- Unilaterally abort transaction.

✓ Failure in PREPARED state
- Contact other sites to determine fate of transaction.

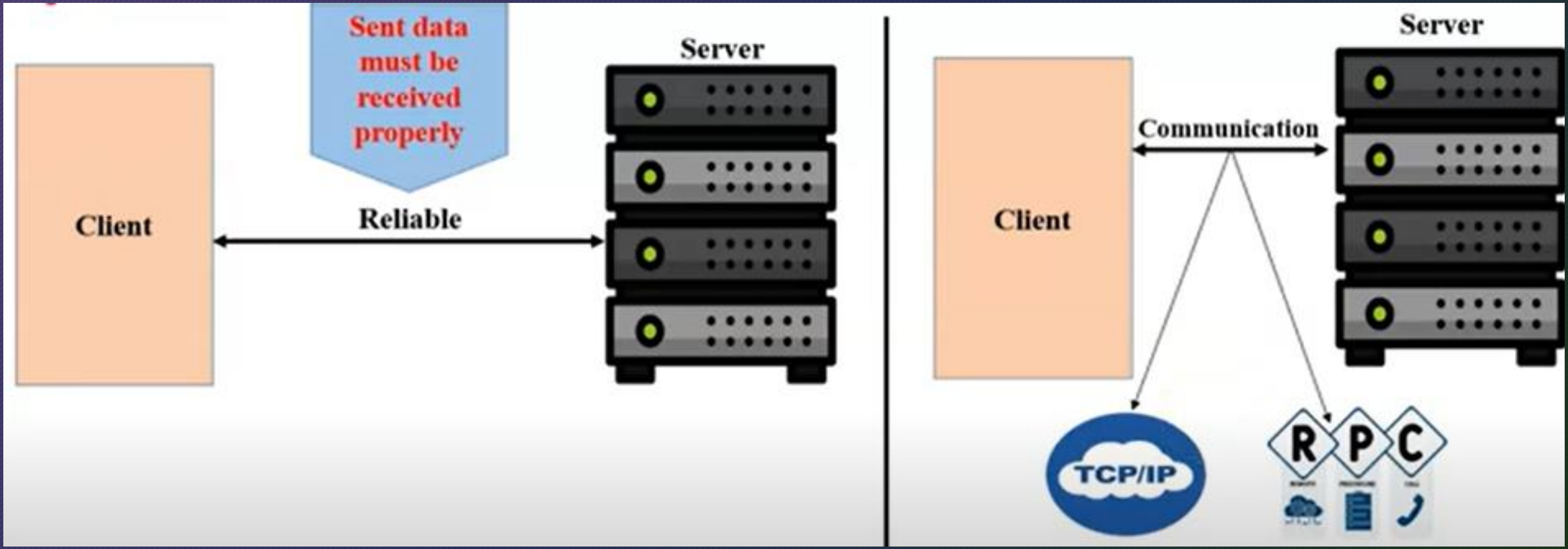
✓ Failure in PRE-COMMITTED state
- Contact other sites to determine fate of transaction.

✓ Failure in ABORTED or COMMITTED states
- On restart, no further action is necessary.



Reliable Client-Server Communication

Introduction



It can be performed two ways:
1. Client-Server Communication using TCP
2. Client-Server Communication using RPC

Point-to-Point Communication

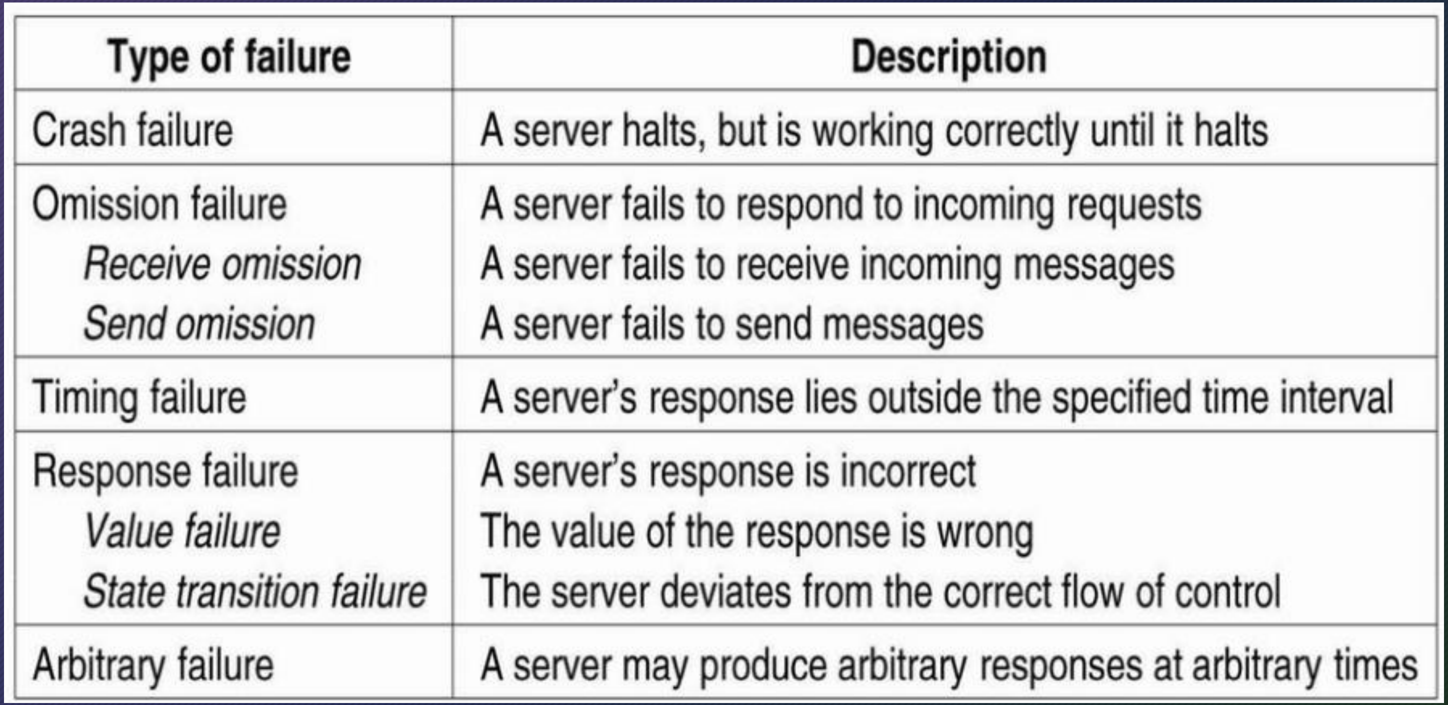
Using TCP

- A reliable point-to-point communication can be established by using TCP protocol
- TCP masks omission failures
- TCP does not mask crash failures

Using RPC

- The goal of RPC is to hide communication by making remote procedure calls that look just like local ones.
- The RPC mechanism works well as long as both the client and server function perfectly

Failure Types



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

Reasons for Failure

- Process can fail
- Communication channel can also fail

RPC Failure Classes

- The RPC mechanism works well as long as both the client and server function perfectly

- 5 classes of RPC failure can be identified:

1. The client cannot locate the server [No request can be sent]
2. The client’s request to the server is lost

   [No response is returned by the server to the waiting client]
3. The server crashes after receiving the request

   [The service request is left acknowledged, but undone]
4. The server’s reply is lost on its way to the client

   [Service has completed, but the results never arrive to the client]
5. The client crashes after sending its request [Server sends a reply to a newly restarted client that may not be expecting it]

1. Client Unable to Locate Server

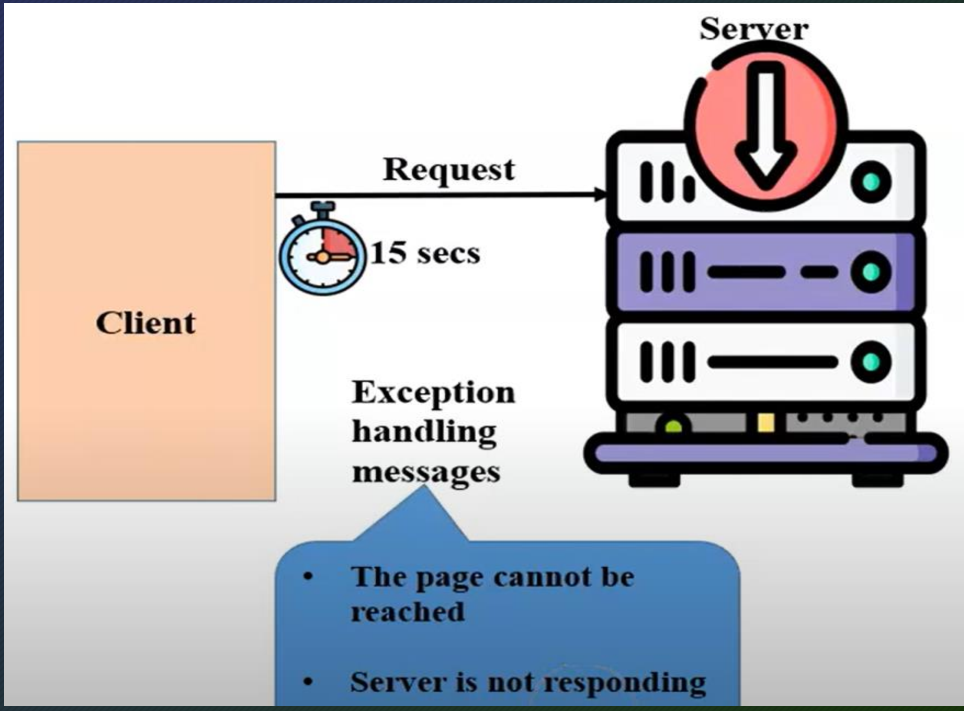
- Problem- When server goes down

- The RPC system informs the client of the failure

- Solution: Exception handling

Java- Division by Zero

C- Signal handlers

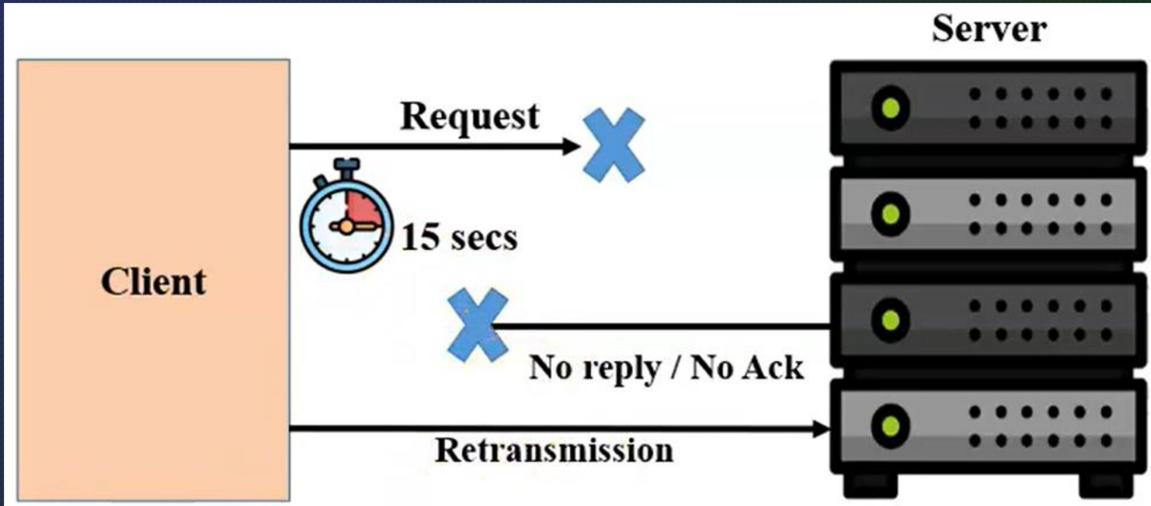

Server
Request
15 secs
Client
Exception handling messages
• The page cannot be reached
• Server is not responding

2. Lost Request Message (Client -> Server)

- OS starts a timer when the stub is generated and sends a request. If response is not received before timer expires, then a new request is sent.

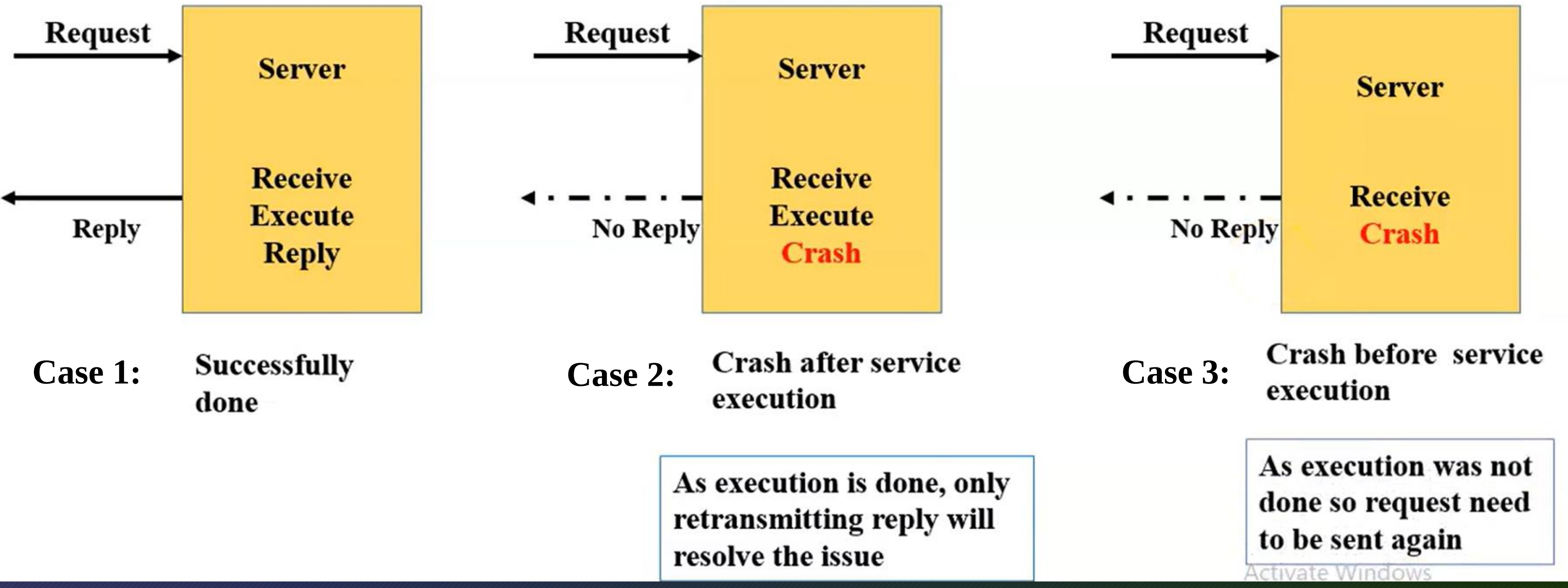
- Solution: Using Acknowledgement and timer concept

- Advantage: Server will never know that packets are retransmitted



3. Server Crashes After Receiving Request

- Solution: Using Acknowledgement and timer concept
- Depending on the case of crash, the operation that server should perform will vary



As execution is done, only retransmitting reply will resolve the issue

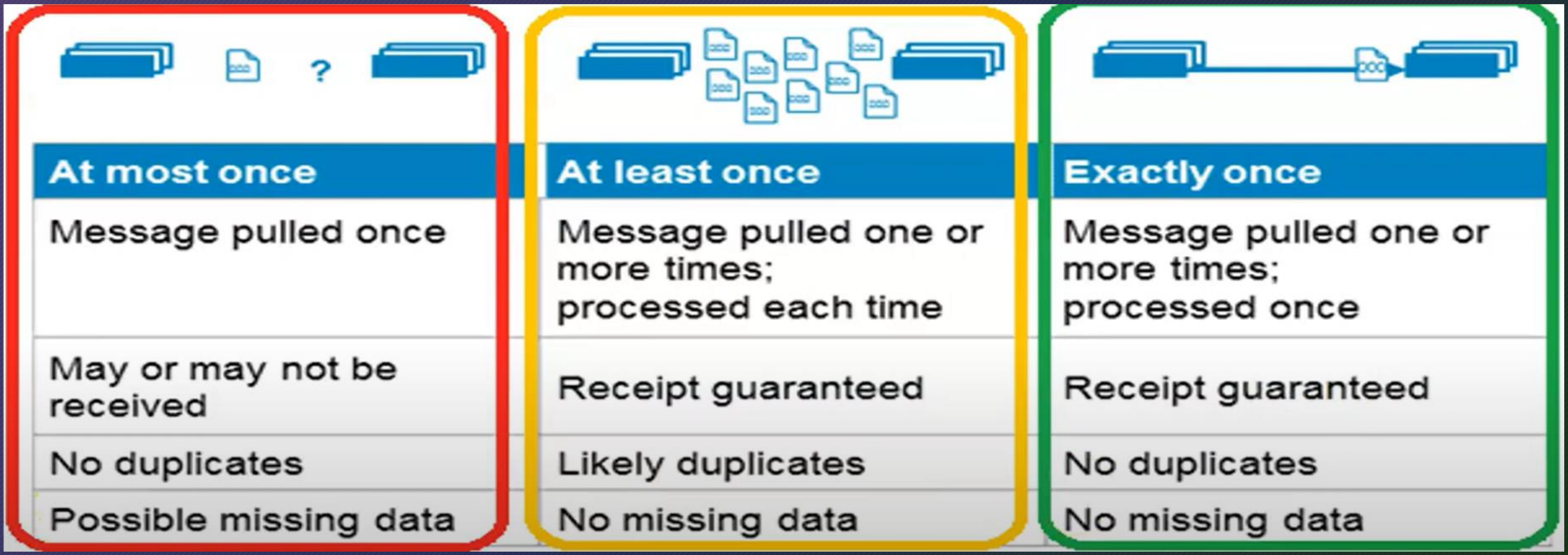
As execution was not done so request need to be sent again

3. Server Crashes After Receiving Request

- Remote operation: print some text and (when done) send a completion message
- Three events that can happen at the server:
  - Send the completion Message (M)
  - Print the text (P)
  - Crash (C)
- Three events can occur in six different orderings
  1. M -> P -> C [A crash occurs after sending the completion message and printing the text]
  2. M -> C (->P) [A crash happens after sending the completion message, but before the text could be printed.]
  3. P -> M -> C [A crash occurs after sending the completion message and printing the text]
  4. P -> C (-> M) [The text printed, after which a crash occurs before the completion message could be sent]
  5. C (-> P -> M) [A crash happens before the server could do anything]
  6. C (-> M -> P) [A crash happens before the server could do anything]

Four Possible Implementation Strategies

1 2 3



At most once	At least once	Exactly once
Message pulled once	Message pulled one or more times; processed each time	Message pulled one or more times; processed once
May or may not be received	Receipt guaranteed	Receipt guaranteed
No duplicates	Likely duplicates	No duplicates
Possible missing data	No missing data	No missing data

4. **No Semantics:** Nothing is guaranteed, and client and servers take their chances

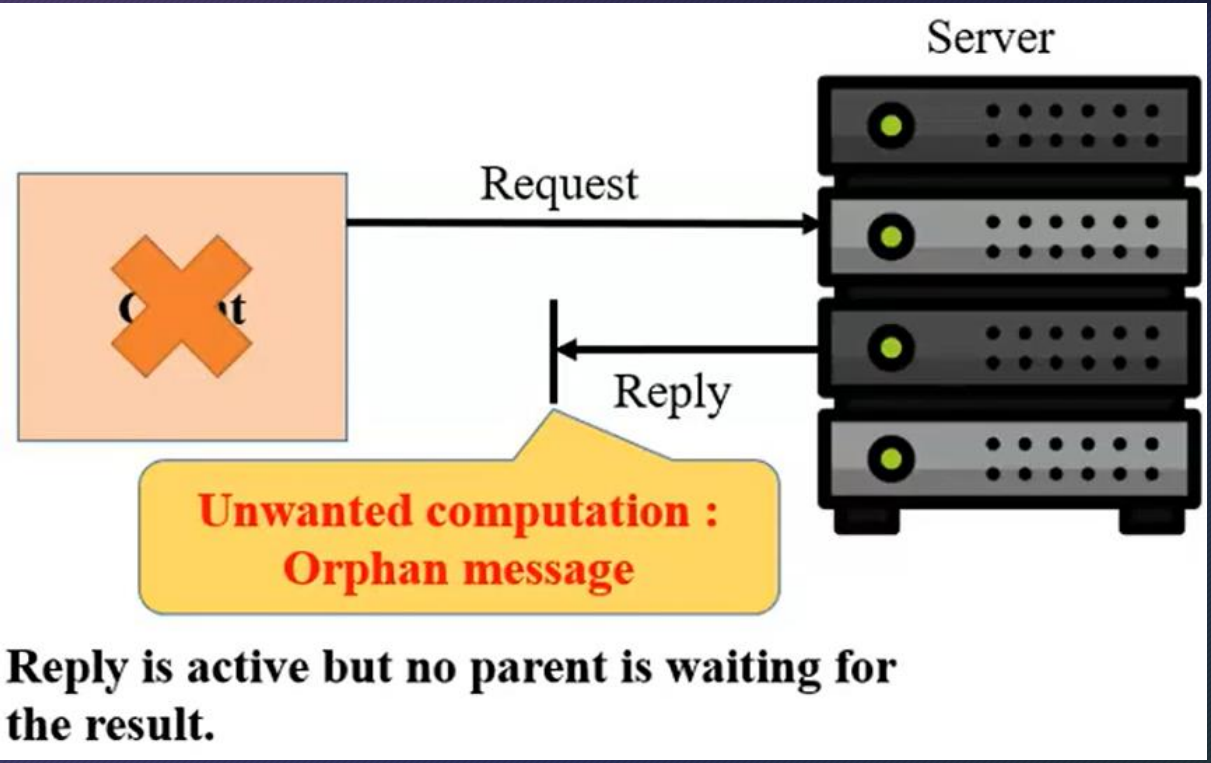
4. Lost Reply Message (Server -> Client)

- Lost replies are difficult to deal with.
- Why was there NO reply?
    - ANS: Is the server DEAD or SLOW?

- Some operations can be repeatedly safely without no damage.
- Example: Requesting 1024 bytes repetitively will cause no harm. A request that has this property is said to be IDEMPOTENT

- Non-Idempotent requests (i.e., electronic transfer of funds) are a little harder to deal with.
    - One way is to structure all requests in idempotent way
    - Employ unique sequence numbers to client
    - Include a field in the header which will identify whether the message is initial or retransmitted.

5. Client Crashes after Sending Request



Reply is active but no parent is waiting for the result.

Four orphan messages solutions

1. Extermination
[The orphan is simply killed-off]

2. Reincarnation
[Each client session has an epoch associated with it, making orphans easy to spot]

3. Gentle Reincarnation
[When a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed]

4. Expiration [If the RPC cannot be completed within a standard amount of time. It is assumed to have expired]

Summary

- The techniques for reliable communication
  - Use redundant bits to detect bit errors in packets
  - Use sequence number to detect packet loss
  - Recover from corrupted/ lost packets using acknowledgements and retransmissions.

THANK YOU

Reference: Distributed Operating Systems by Andrew S. Tanenbaum,
Online Contents