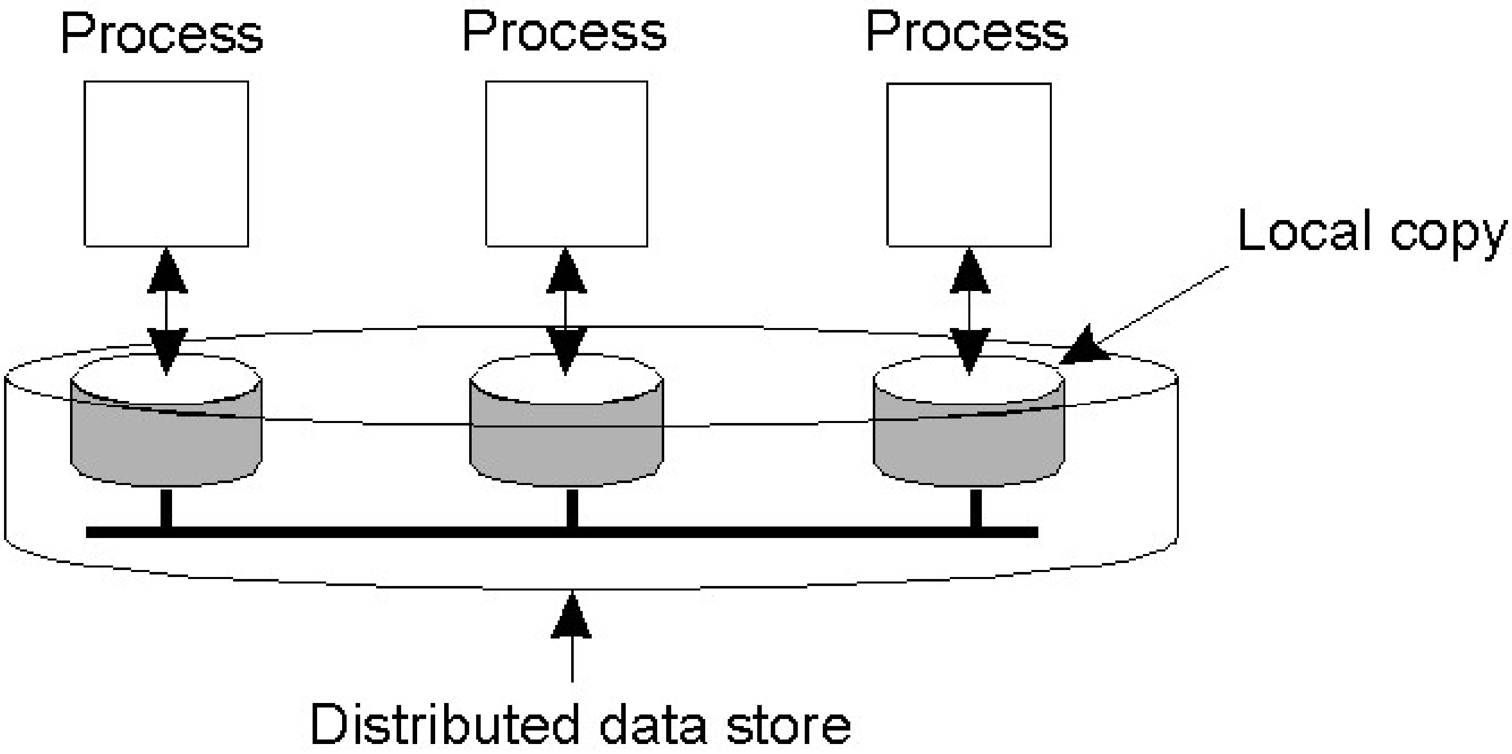
Distributed Computing (2022)

Consistency and Replication

Interprocess Communication

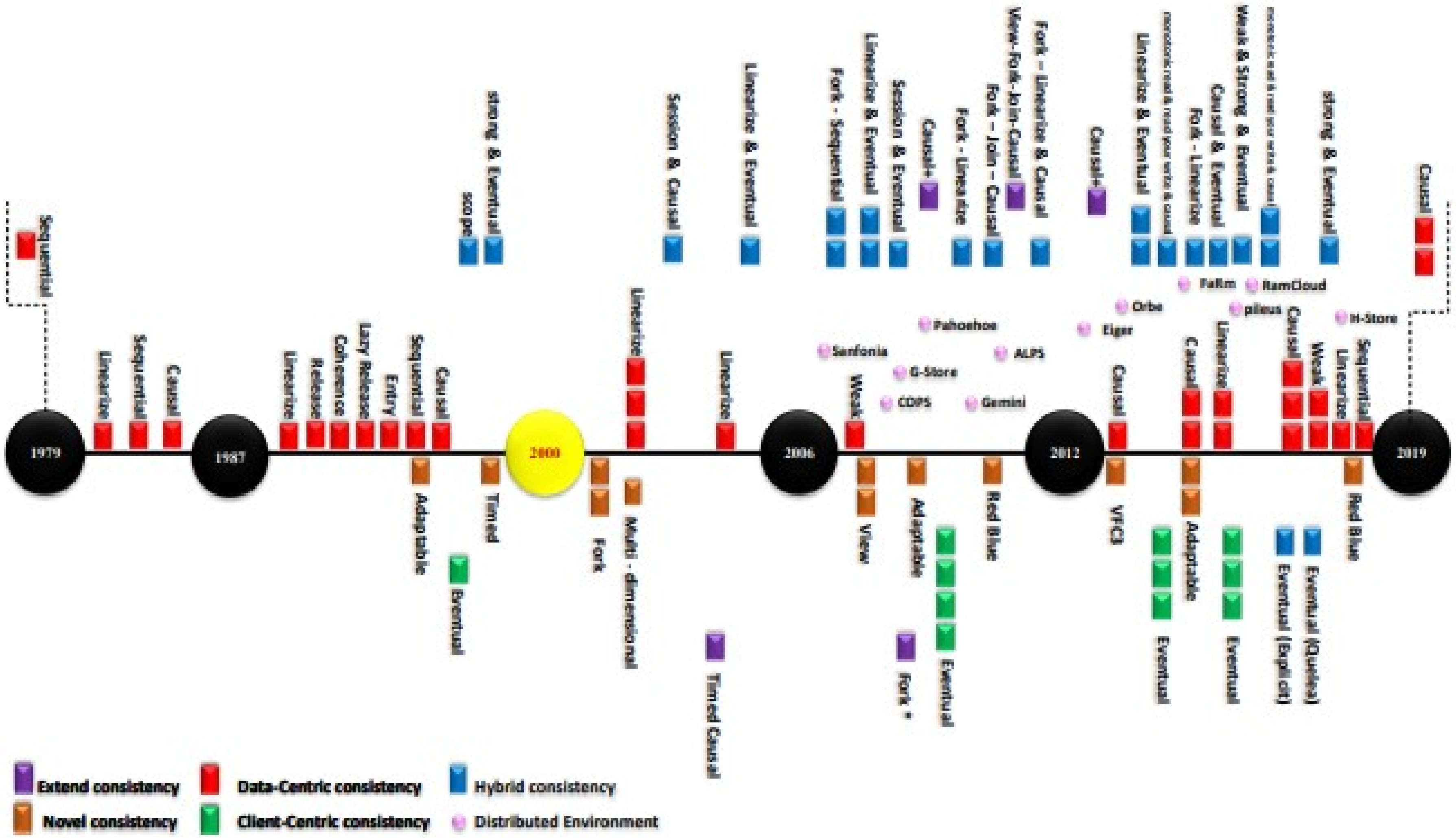
- Shared Memory,
- Message Passing Paradigm

Data-Centric Consistency Models



The general organization of a logical data store, physically distributed and replicated across multiple processes.

Evolution of Consistency Models

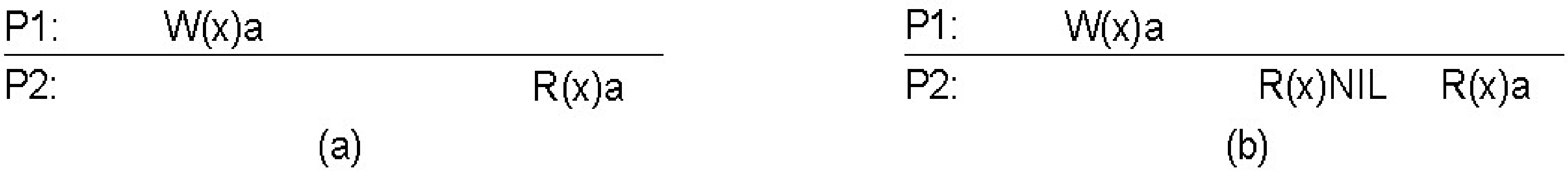


Design and Implementation issues of DSM

- Granularity
- Structure of shared –memory space
- Memory Coherence and Access Synchronization
- Data Location and access
- Replacement strategy
- Thrashing
- Heterogeneity

Strict Consistency

Any read on a data item x returns a value corresponding to the results of the most recent write on x



P1: W(x)a

P2: R(x)a

(a)

P1: W(x)a

P2: R(x)NIL R(x)a

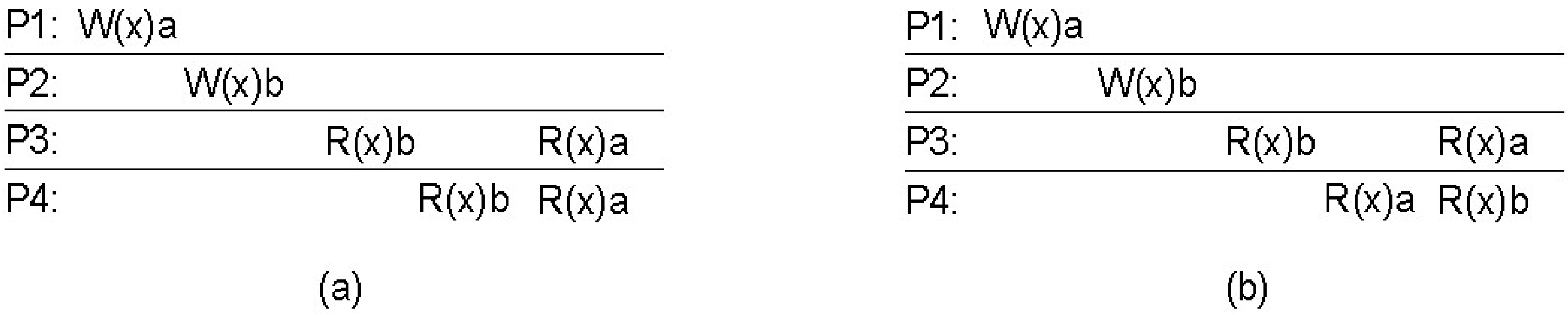
(b)

Behavior of two processes, operating on the same data item.

- A strictly consistent store.
- A store that is not strictly consistent.

Sequential Consistency (1/single copy)

The result of any execution is the same as if the read and the write Operations by all processes on the data store were executed in some Sequential order and the operations of each individual process appear in this sequence in the order specified by its program.



P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)b R(x)a

(a)

P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)a R(x)b

(b)

a) A sequentially consistent data store.

b) A data store that is not sequentially consistent.

Linearizability Consistency (2)

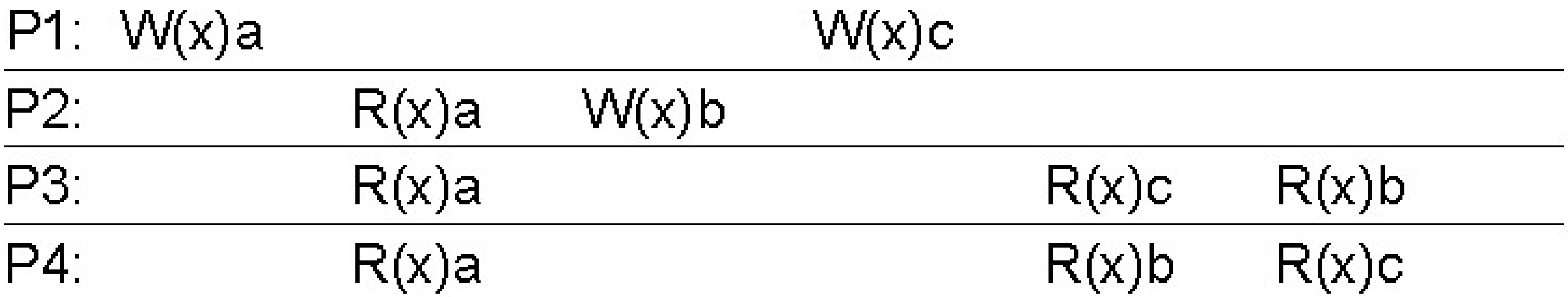
The result of any execution is the same as if the read and the write Operations by all processes on the data store were executed in some Sequential order and the operations of each individual process appear in this sequence in the order specified by its program.
In addition, if ts OP1(x) < ts OP2(y), then operation OP1(x) should Precede OP(y) in this sequence.

Causal Consistency (1)

Necessary condition:

Writes that are potentially casually related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

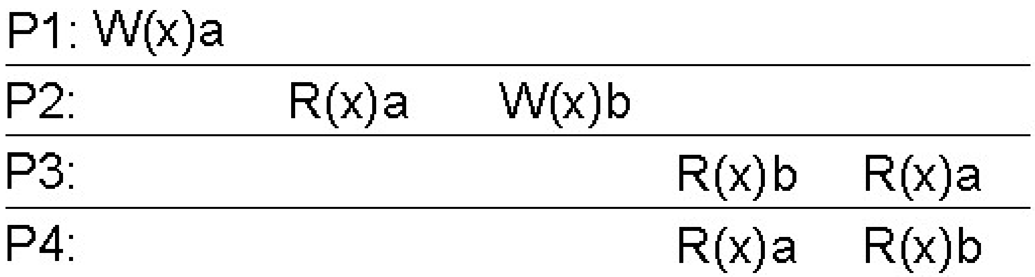
Causal Consistency (2)



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

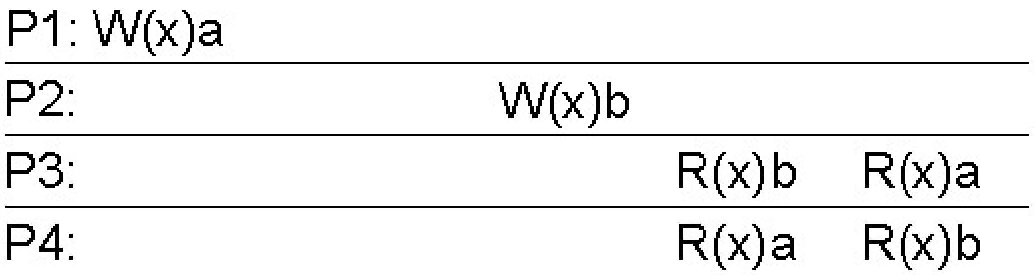
This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Causal Consistency (3)



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)



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

(b)

a) A violation of a casually-consistent store.

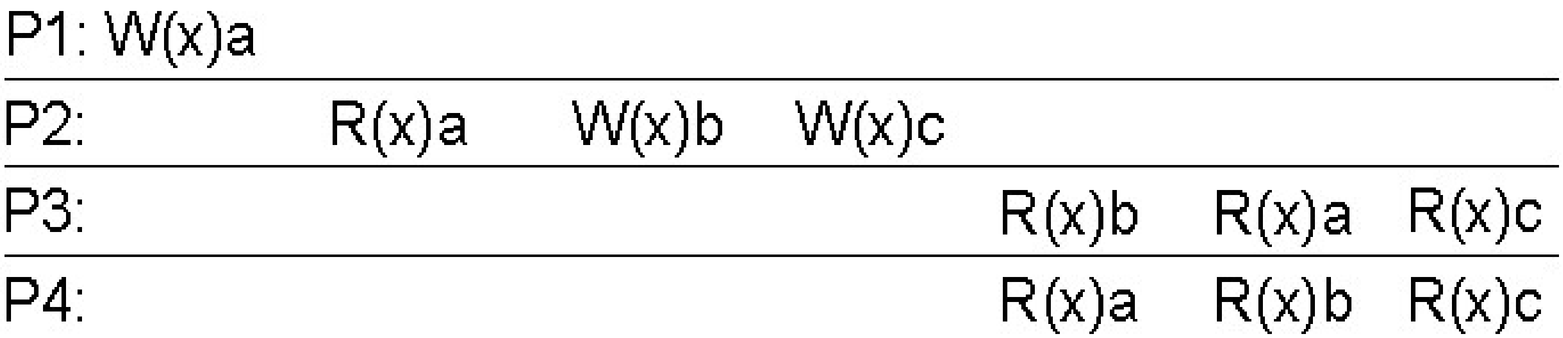
b) A correct sequence of events in a casually-consistent store.

FIFO Consistency (1) Pipelined RAM

Necessary Condition:

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.

FIFO Consistency (2)



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 events of FIFO consistency

Weak Consistency (1)

Properties:

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

(All processess see all ops on synch. Variable in the same process in the same order.)

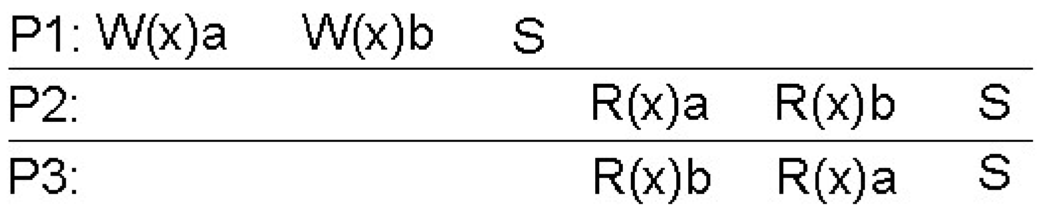
- No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.

(Syn. Forces all writes that are in progress or partially completed at some local copies but not others to complete everywhere.)

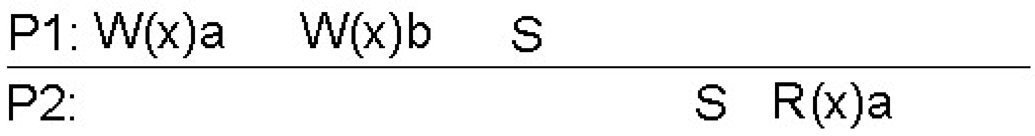
- No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

(when a data item is accessed either for reading or for writing all the syncs. have to be completed)

Weak Consistency (2)



(a)

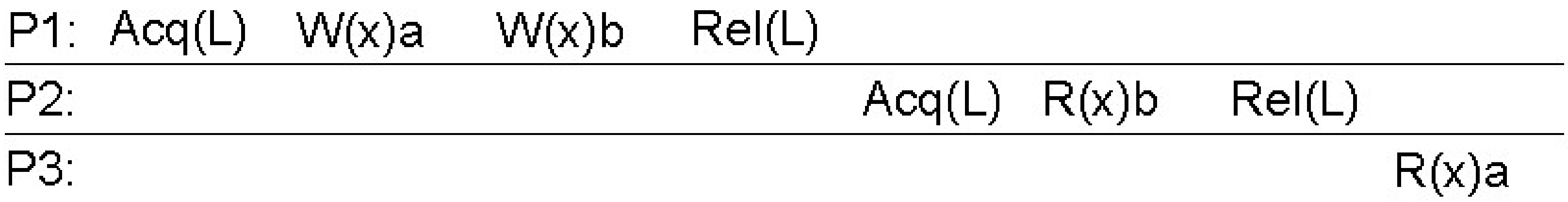


(b)

a) A valid sequence of events for weak consistency.

b) An invalid sequence for weak consistency.

Release Consistency (1)



P1: Acq(L) W(x)a W(x)b Rel(L)

P2: Acq(L) R(x)b Rel(L)

P3: R(x)a

A valid event sequence for release consistency.

Release Consistency (2)

Rules:

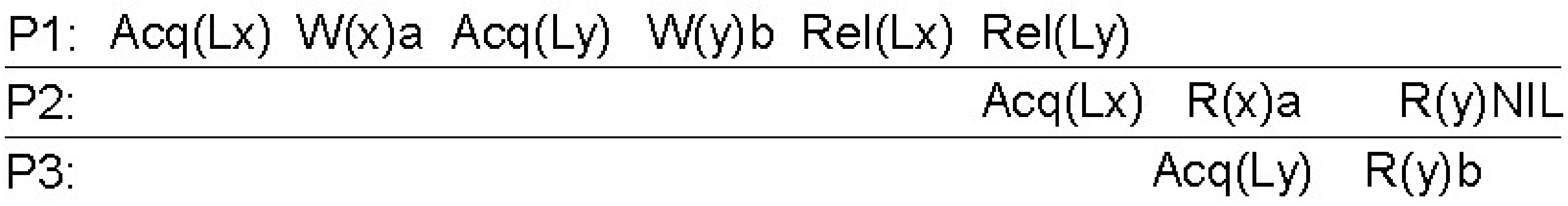
- Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
- Before a release is allowed to be performed, all previous reads and writes by the process must have completed
- Accesses to synchronization variables are FIFO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

Entry Consistency (1)

Conditions:

- 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.(At an acquire, all remote changes to the guarded data must be made visible)
- 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. **(the process should enter critical region and ensure mutual exclusion)**
- 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. (the process entering CR in non exclusive mode should check with the owner of the variable if the copy is most recent)

Entry Consistency (1)



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

A valid event sequence for entry consistency.

Summary of Consistency Models

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)

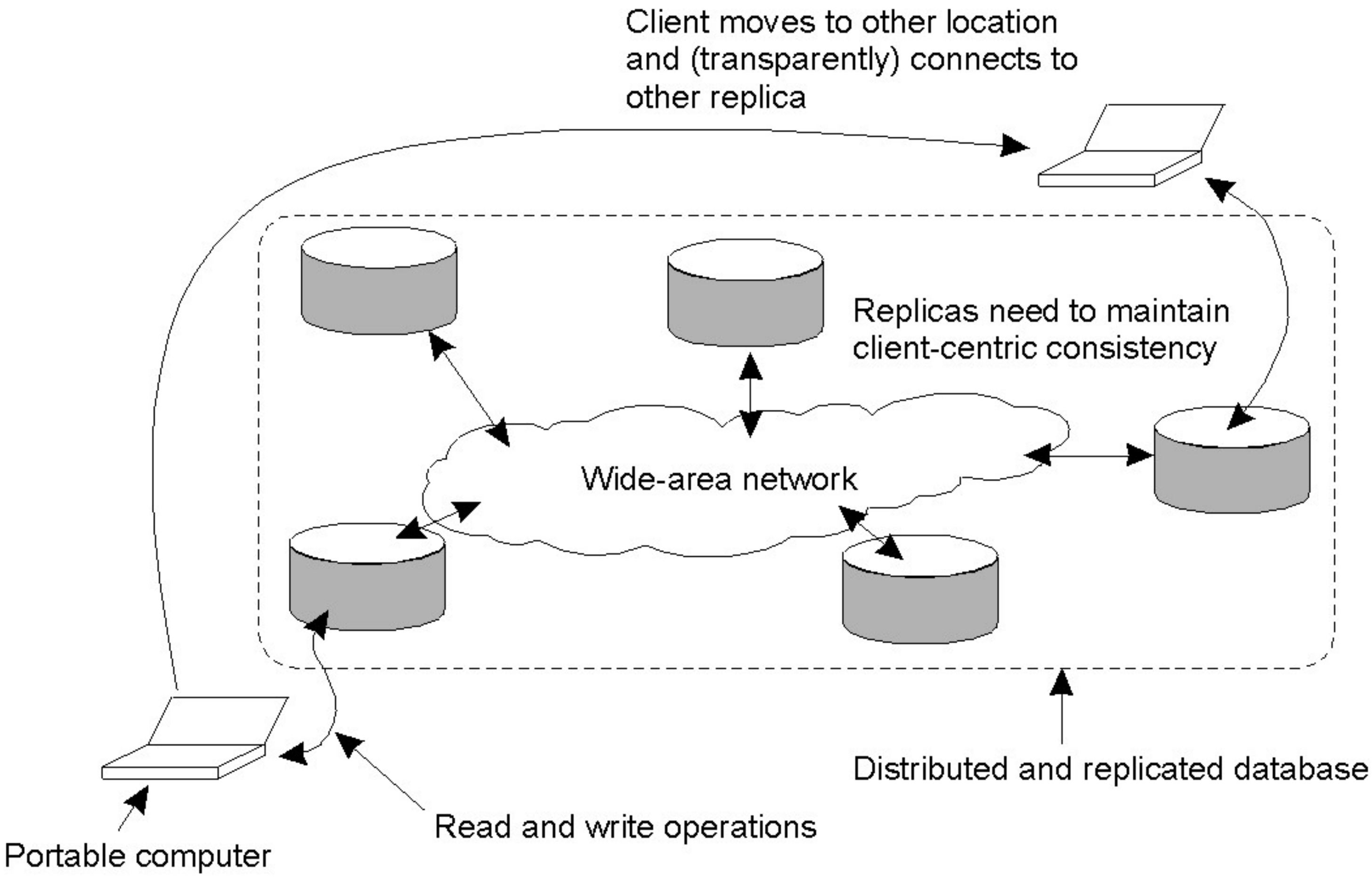
Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

a) Consistency models not using synchronization operations.

b) Models with synchronization operations.

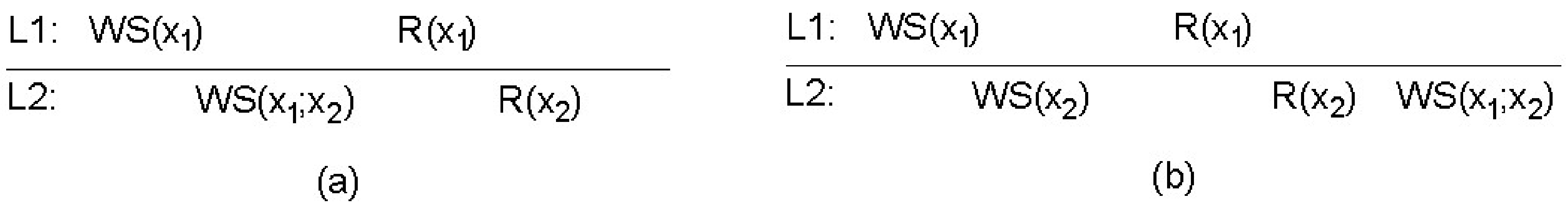
Eventual Consistency


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

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

Monotonic Reads

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. ie. *If a process has seen a value of x at time t , it will never see an older version of x at a later time.*


L1: WS(x_1) R(x_1)
L2: WS(x_1;x_2) R(x_2)
(a)
L1: WS(x_1) R(x_1)
L2: WS(x_2) R(x_2) WS(x_1;x_2)
(b)

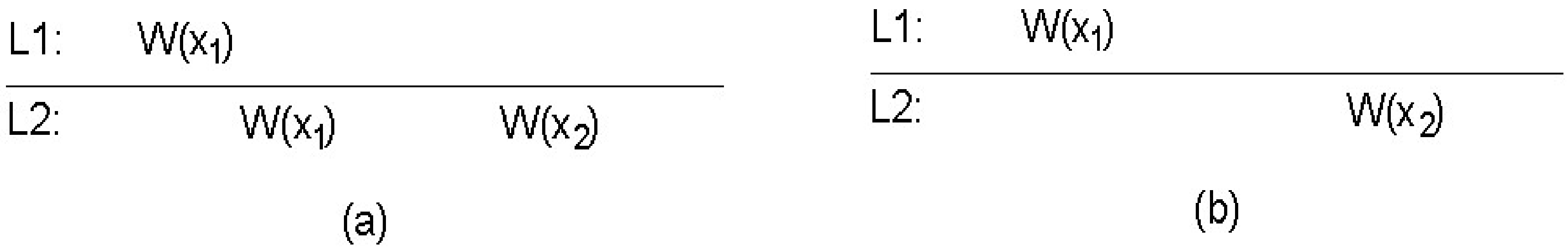
The read operations performed by a single process *P* at two different local copies of the same data store.

a) A monotonic-read consistent data store

b) A data store that does not provide monotonic reads.

Monotonic Writes

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



(a)

(b)

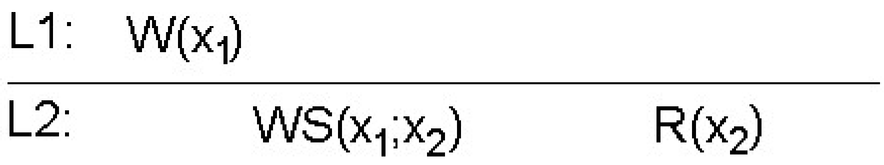
The write operations performed by a single process *P* at two different local copies of the same data store

a) A monotonic-write consistent data store.

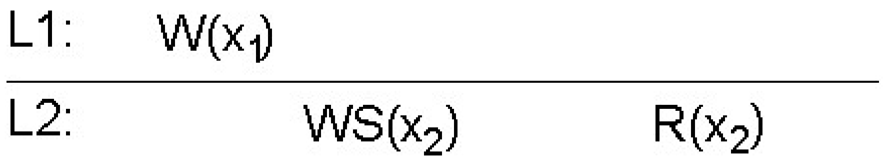
b) A data store that does not provide monotonic-write consistency.

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.



(a)



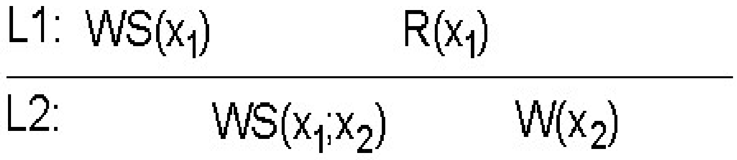
(b)

a) A data store that provides read-your-writes consistency.

b) A data store that does not.

Writes Follow Reads

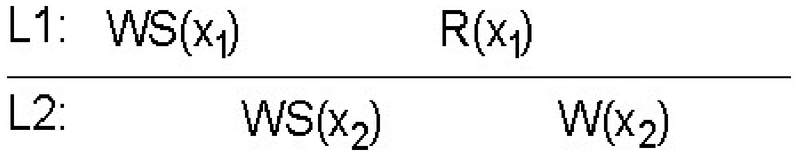
A write operation by a process on a data item x following a previous read operation on x by the same process , is guaranteed to take place on the same or a more recent value of x that was read.



L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_1;x_2)$ $W(x_2)$

(a)



L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_2)$ $W(x_2)$

(b)

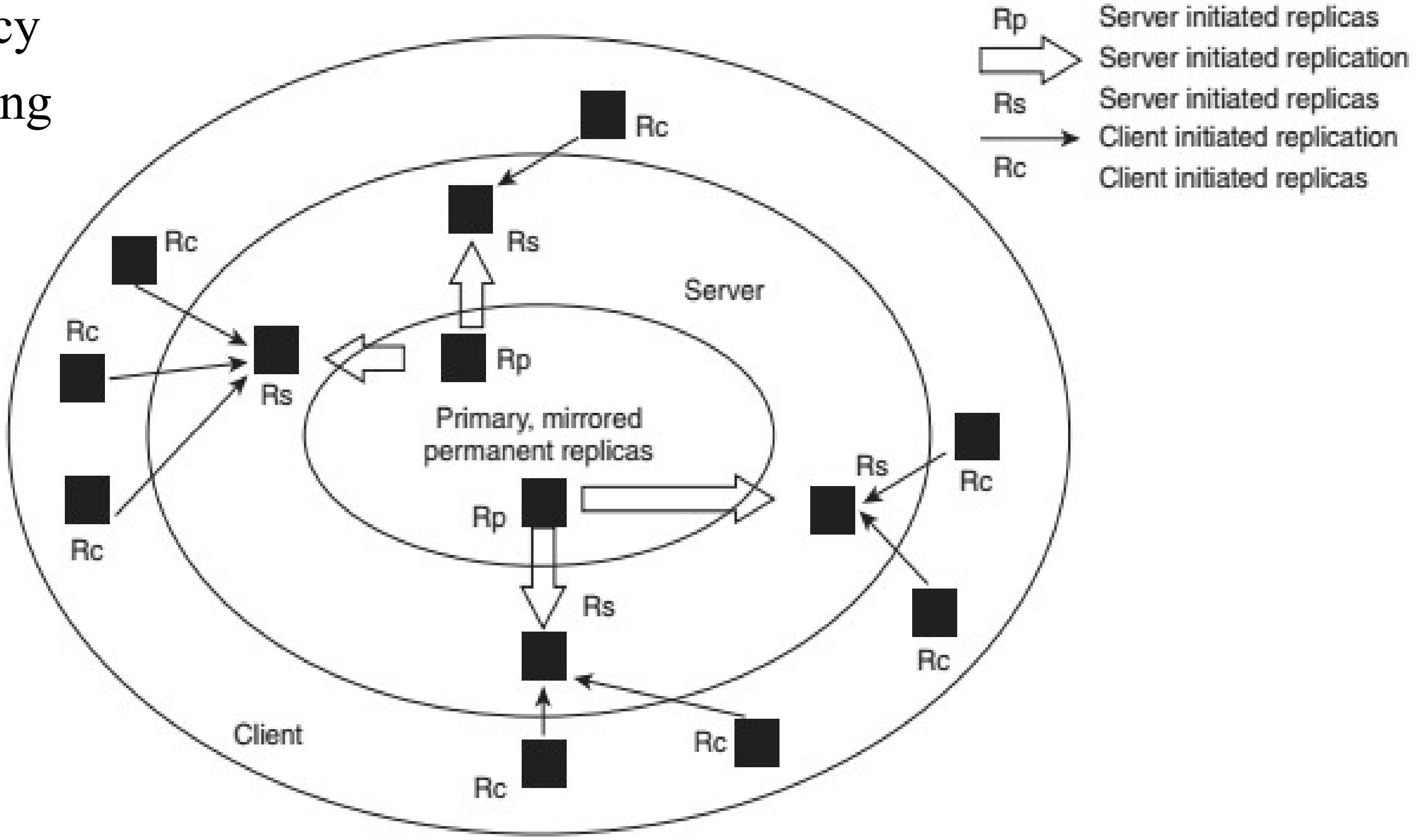
a) A writes-follow-reads consistent data store

b) A data store that does not provide writes-follow-reads consistency

It assures that reactions to articles are stored at a local copy only if the original is stored there as well.

Replicas

- Access latency
- Load balancing


Rp
Server initiated replicas
Server initiated replication
Rs
Server initiated replicas
Client initiated replication
Rc
Client initiated replicas
Rc
Rs
Server
Rc
Rc
Rs
Rp
Primary, mirrored
permanent replicas
Rs
Rc
Rp
Rc
Rs
Rc
Client
Rc
Rc

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

Permanent Replicas

Initial set of replicas

– Other replicas can be created from them

– Small and static set

- Example: Web site horizontal distribution

1. Replicate Web site on a limited number of machines on a LAN

– Distribute request in round-robin

2. Replicate Web site on a limited number of machines on a WAN **(mirroring)**

– Clients choose which sites to talk to

Server-Initiated Replicas (1)

Dynamically created at the request of the owner of the DS

- Example: push-caches

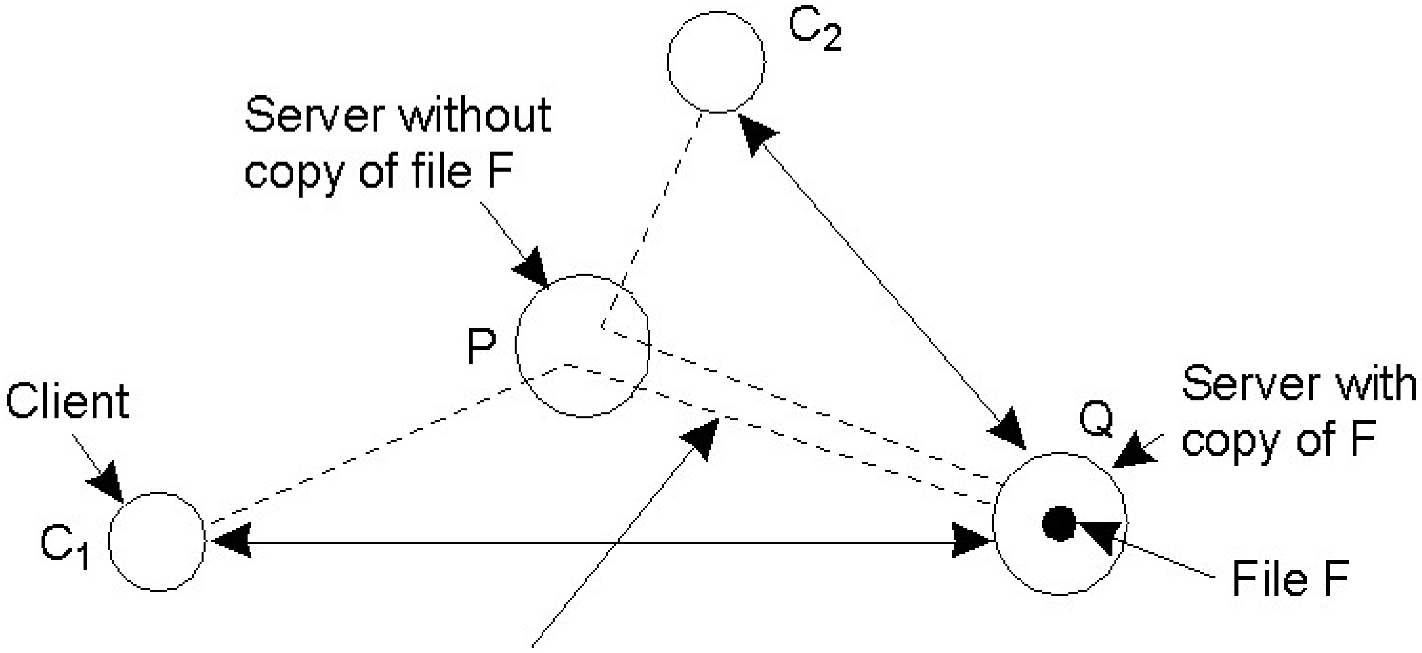
– Owners: web owners for CNN, Yahoo etc.

– Web hosting servers can **dynamically** create replicas close to the demanding client

- Need **dynamic policy** to create and delete replicas
- One is to keep track of Web page hits

– Keep a counter and access-origin list for each page

Server-Initiated Replicas



Server Q counts access from C_1 and C_2 as if they would come from P

1. Reduce load on servers
2. Specific files could be migrated.

Counting access requests from different clients.

Client-Initiated Replicas

These are caches
–Temporary storage (expire fast)

- Managed by clients
- Cache hit: return data from cache
- Cache miss: load copy from original server
- Kept on the client machine, or on the same LAN
- Multi-level caches

Replica Placement

Replication Models

1. Master–Slave: **Slaves are read-only**, updates only in master, slaves only synchronize with master.

2. Client–Server: Just like master–slave model, but updates **can happen in any slave replica too**. Which are first pushed to server, and then to others.

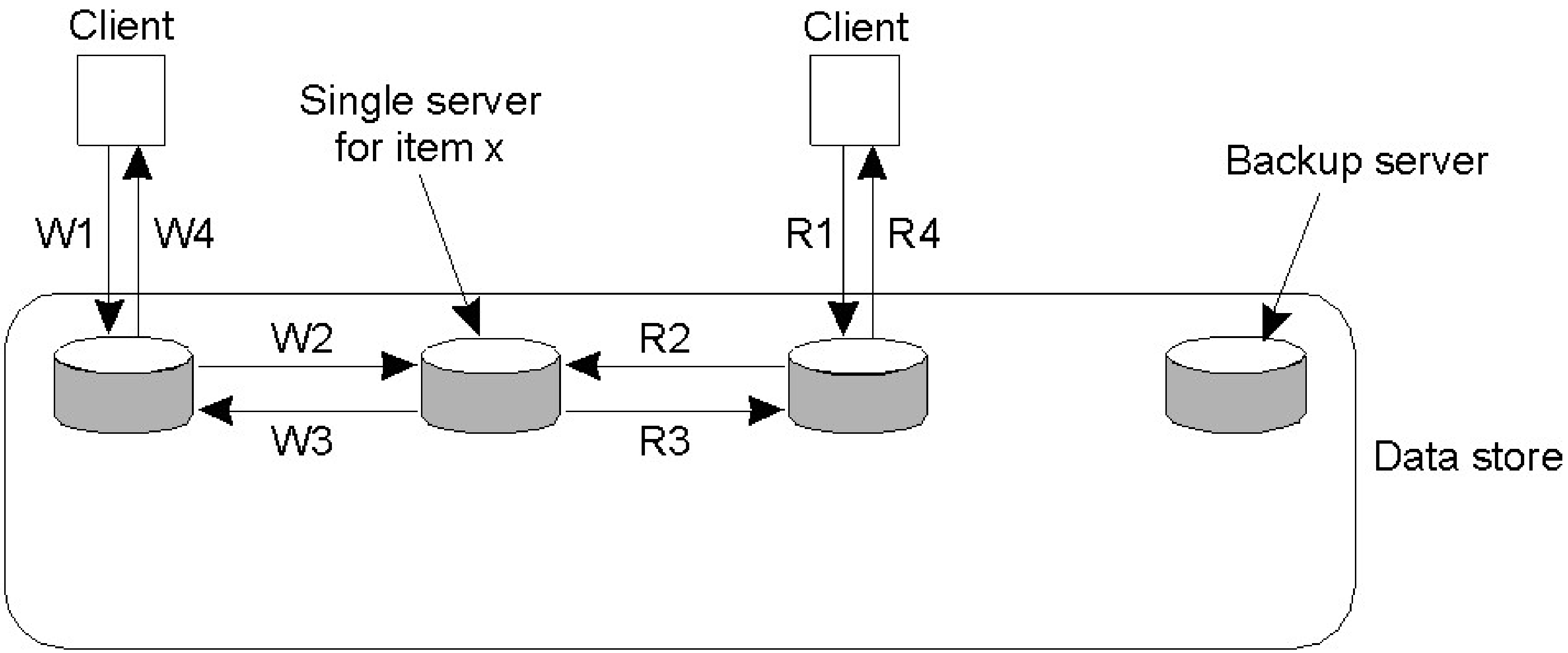
3. Peer–to–Peer: All replicas are of equal importance and are peer. Any replica can synchronize with any other besides any replica can propagate the update. All peers should have all the information

Replica Consistency

1. Optimistic:
2. Pessimistic:

Consistency Protocols: Remote-Write Protocols (1)

Primary –based : data item x is associated with a primary responsible for co-ord. writes

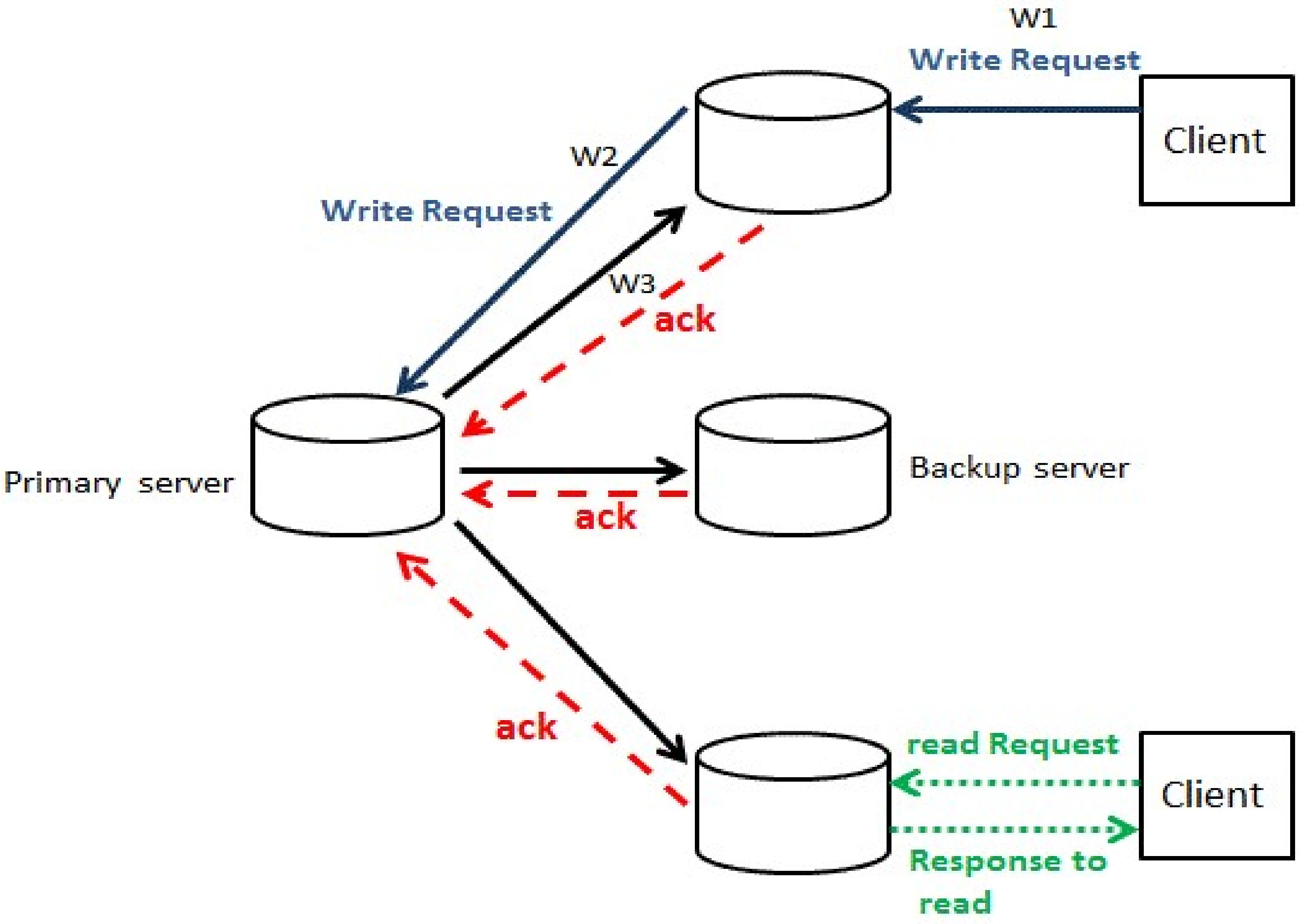


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

Primary-based remote-write protocol with a fixed server to which all read and write operations are forwarded.

Primary-Based replication Protocol



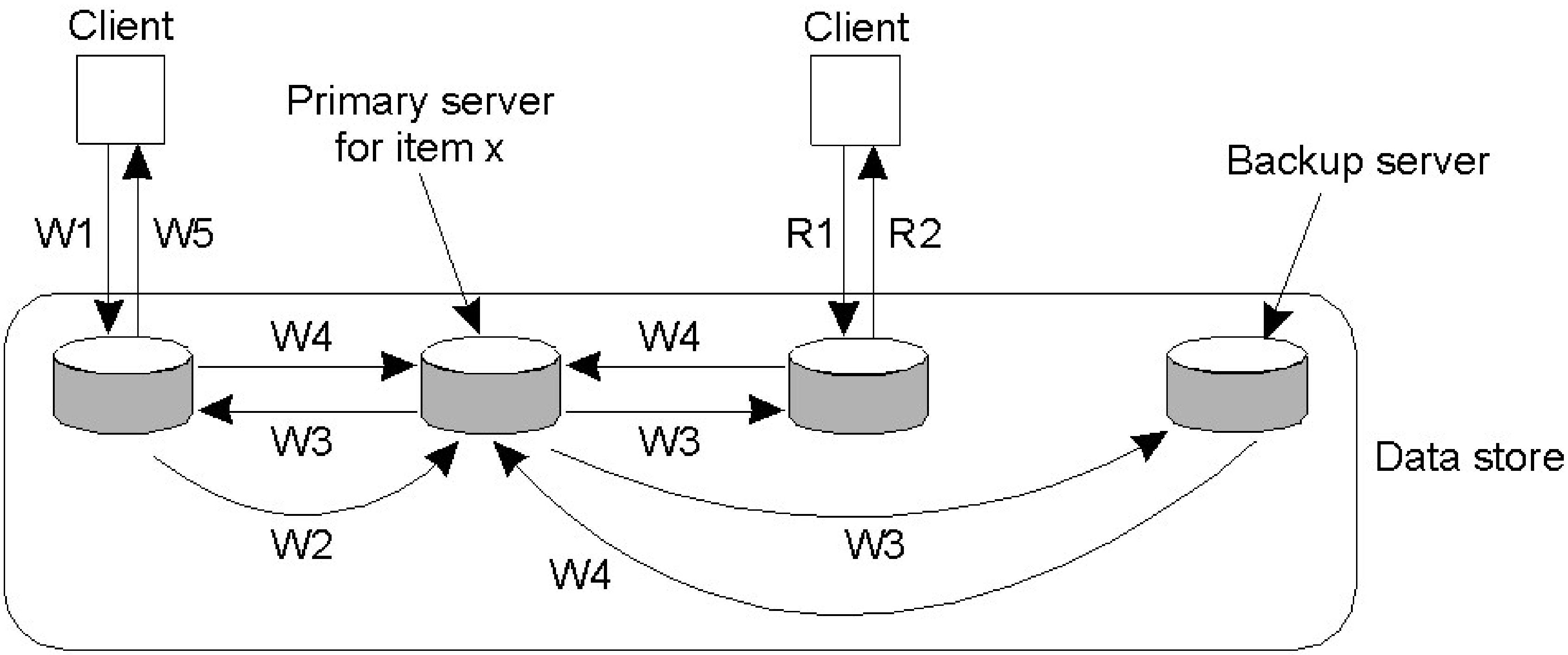
Primary-Based Remote-Write Protocols (cont.)

Replicated: primary-backup remote write protocol

–Primary copy and backups for each data item
–Read from local copy
–Write to the (remote) primary server
–Update backups

- Blocking vs. Non-Blocking update

Remote/Replicated -Write Protocols

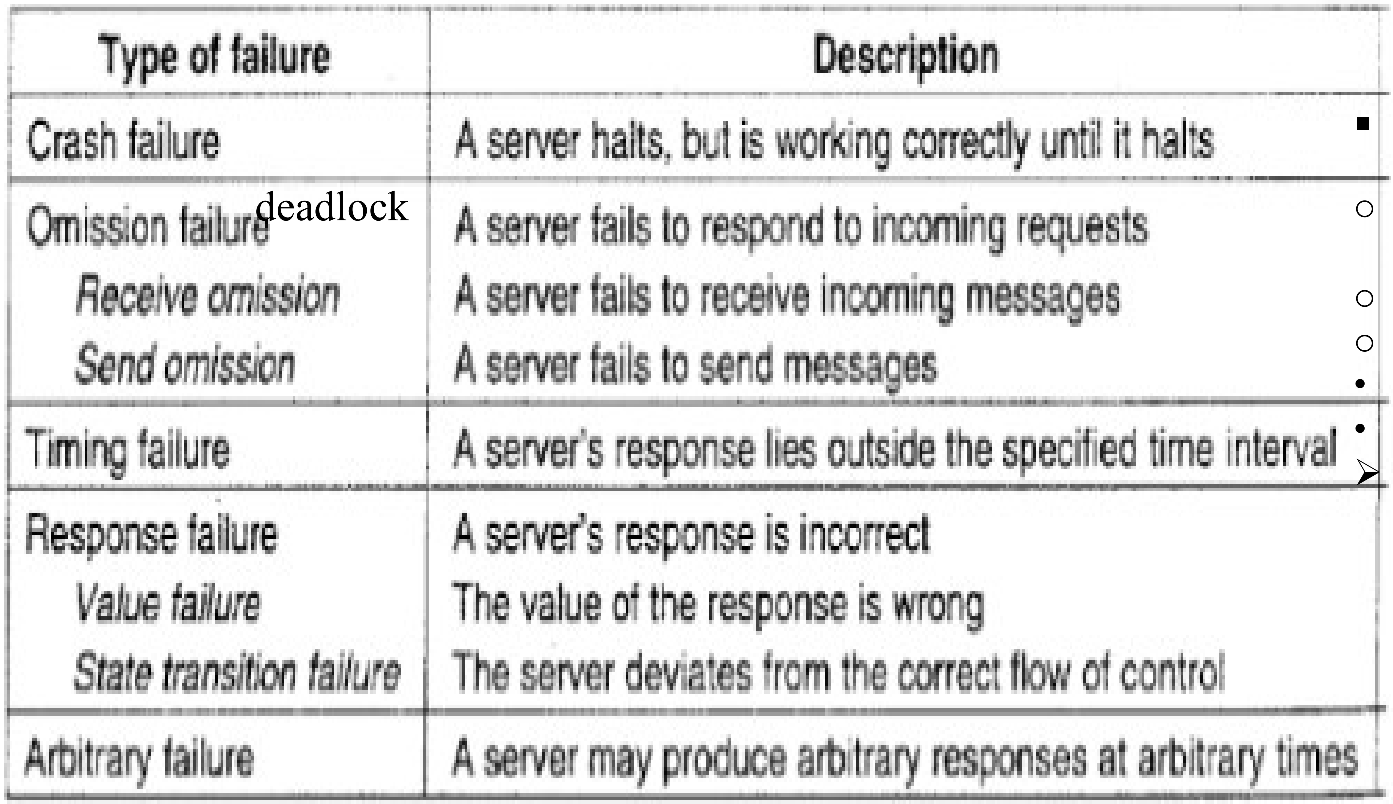


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

The principle of primary-backup protocol.

Fault Tolerance



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

deadlock

- ▪ OS Crash
- ○ client-server connection failure,
- ○ listen thread is not available
- ○ buffer size is not adequate
- • send buffer overflows
- • even link failure
- ➢ Providing data too soon or late

❑ Black hole routers

Failure Detection: Synchronous /Asynchronous

Failure Detection: Synchronous /Asynchronous

- ***Fail-stop failures*** refer to crash failures that can be reliably detected. This may occur when assuming non-faulty communication links and the failure detecting process P can place a worst-case delay (Final Timeout) on responses from Q.

- ***Fail-noisy failures*** are like fail-stop failures, except that P only eventually comes to the correct conclusion that Q has crashed. This means that there may be some a priori unknown time in which P's detections of the behaviour of Q are unreliable. (Incremental Timeout-Final Timeout)

- • When dealing with **fail-silent failures**, you assume that communication links are non-faulty, but the process P cannot distinguish crash failures from omission failures.

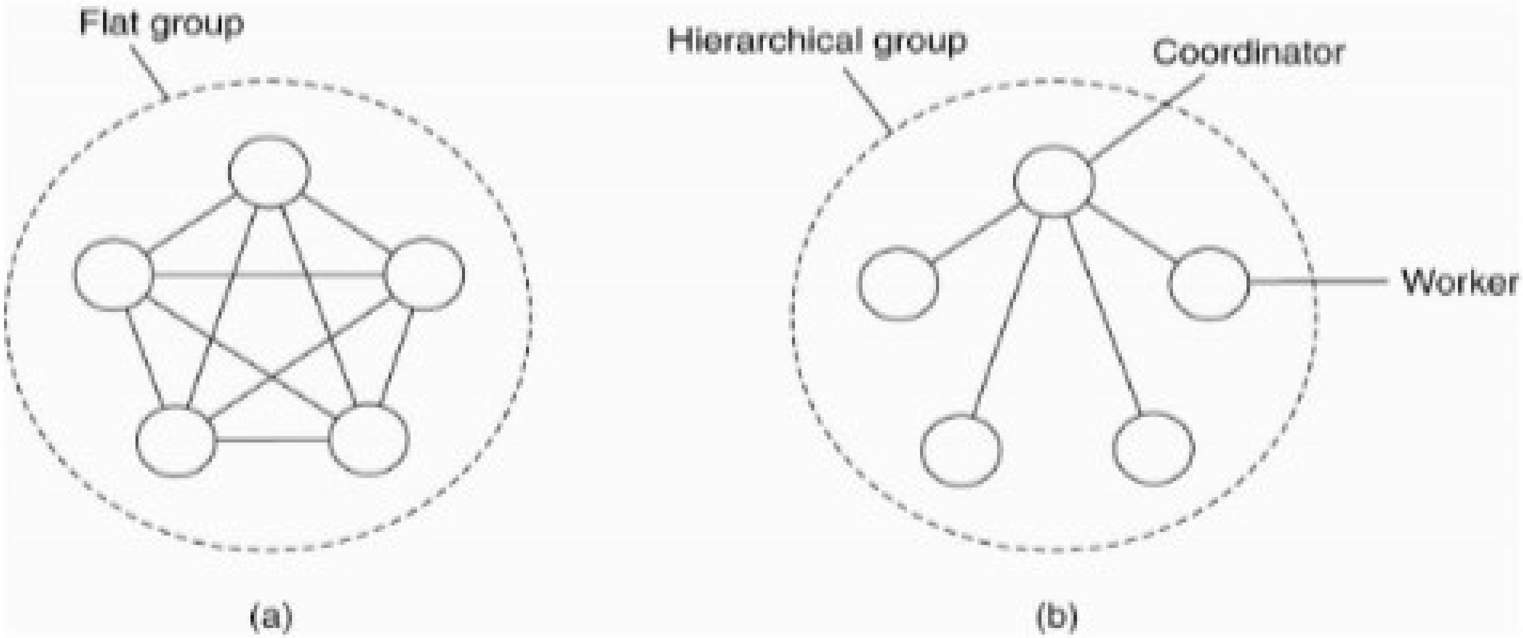
- **Fail-safe failures** cover the case of dealing with arbitrary failures by the process Q, yet these failures are benign, i.e. they cannot do any harm.

- • Finally, when dealing with **fail-arbitrary failures**, Q may fail in any possible way; failures may be unobservable in addition to being harmful to the otherwise correct behaviour of other processes. Clearly, the worst situation is that in which you have to deal with fail-arbitrary failures.

Failure Masking

- *1. Information redundancy*:, parity, checksums, Hamming codes, forward error correction (FEC) codes, etc.
- *2. Time redundancy*: An additional time that is used to deliver the service of a system or multiple executions of an operation. For example, retransmission handling in networks, or if a transaction aborts, then rollback till the closest checkpoint restarts.
- *3. Physical redundancy* For example, extra processes can be added to a system so that if a small number of them get crashed, the system can still function correctly. Even in biology (mammals have two eyes, two ears, two lungs, etc.), aircraft (747s have four engines but can fly on three), and sports (multiple referees in case one misses an event). Another example is Apache Hadoop Distributed File System (HDFS) Please check the word as the abbreviation does not include “A” for Apache.

Resilience by Process Groups



- Group Organization and Management

Byzantine agreement problem (Byzantine Generals Problem)

Assumptions and Goals:

- Assume that processes are synchronous, messages are unicast while preserving ordering, and communication delay is bounded.
- Assume N processes, where each process i will provide a value vi to others.
- Goal- Let each process construct a vector V of length N, such that
    - ○ if process i is non-faulty, V [i] = vi
    - ○ ELSE V [i] is undefined/unknown
- Assume that there are at most k faulty processes.

Byzantine agreement problem (Byzantine Generals Problem)

Assumptions and Goals:

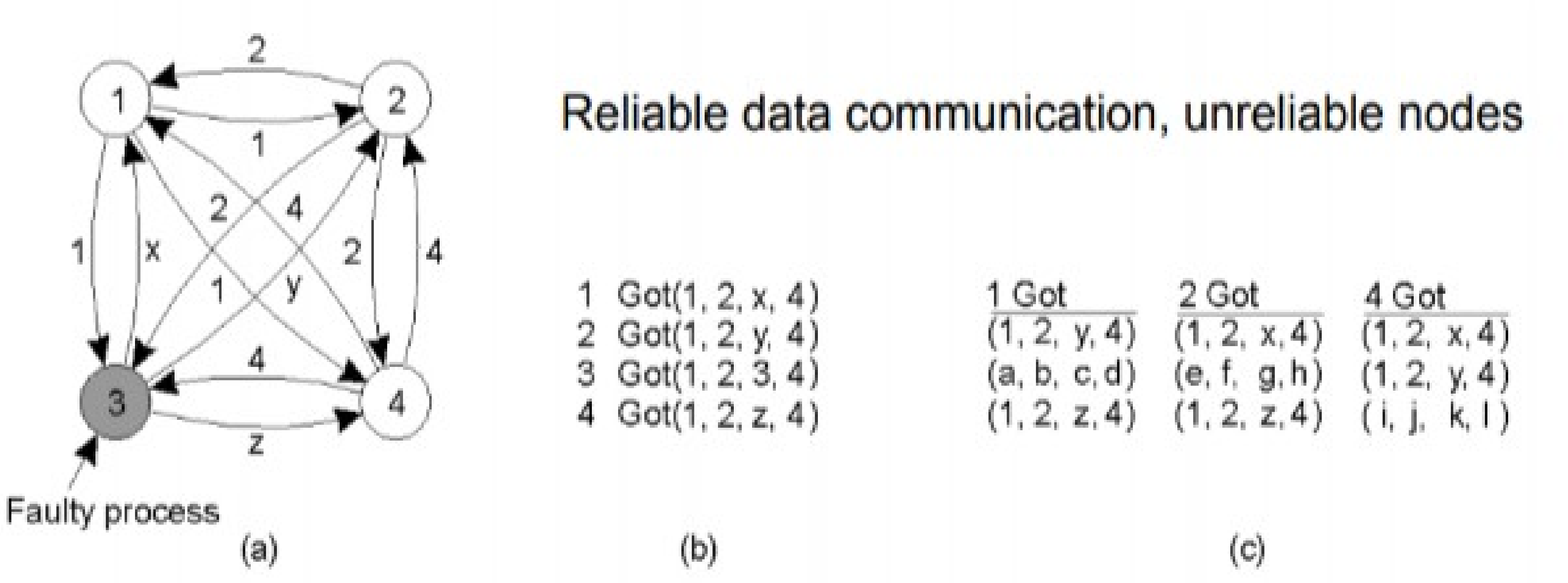
●Assume that processes are synchronous, messages are unicast while preserving ordering, and communication delay is bounded.

●Assume N processes, where each process i will provide a value vi to others.

●Goal- Let each process construct a vector V of length N, such that

- ○ if process i is non-faulty, V [i] = vi
- ○ ELSE V [i] is undefined/unknown

●Assume that there are at most k faulty processes.



Reliable data communication, unreliable nodes

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

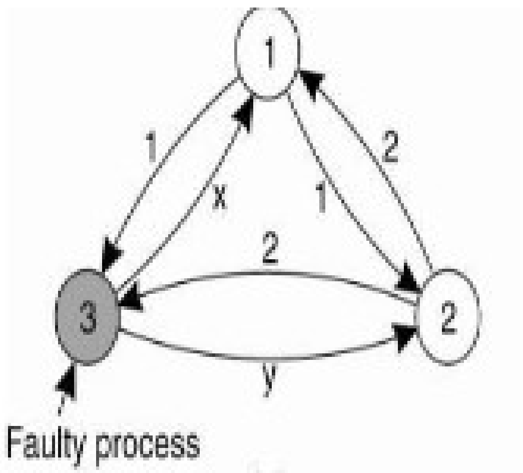
(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

(a) Each process sends its value to others
(b) The vectors, that each process assembles, are based on (a)
(c) The vectors that each process receives in step 3

1. The generals announce their troop strengths (for example, in units of 1-kilo soldiers) to the other members of a group by sending a message.
2. The vectors V, that each general assembles, are based on (a), each general knows its own strength. They then send their vectors to all the other generals.
3. Each general receives vectors in step 3. It is clear to all that General 3 is the traitor. In each ‘column’, the majority value is assumed to be correct.



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

BFT in Block Chain **Bitcoin**

- A blockchain is a chain of **time stamped** data blocks that **contain transactions** with each **block hashed to the previous one**. This provides immutability to a blockchain. These blocks on a chain **can only be added using a distributed consensus algorithm and also one should be able to identify the correct chain**.

- Consensus algorithms using a modified version of Proof of Work (PoW), created by Satoshi Nakamoto for Bitcoin became almost the first (BFT) algorithm which was open and distributed peer-to-peer (P2P) network that utilises a distributed network of anonymous nodes that are free to join and leave,based on the concept of “Mining” as compared to a traditional semi-closed group of nodes, deploying Practical Byzantine Fault Tolerance (pBFT) voting-based algorithm.

- PoW was designed for two things:

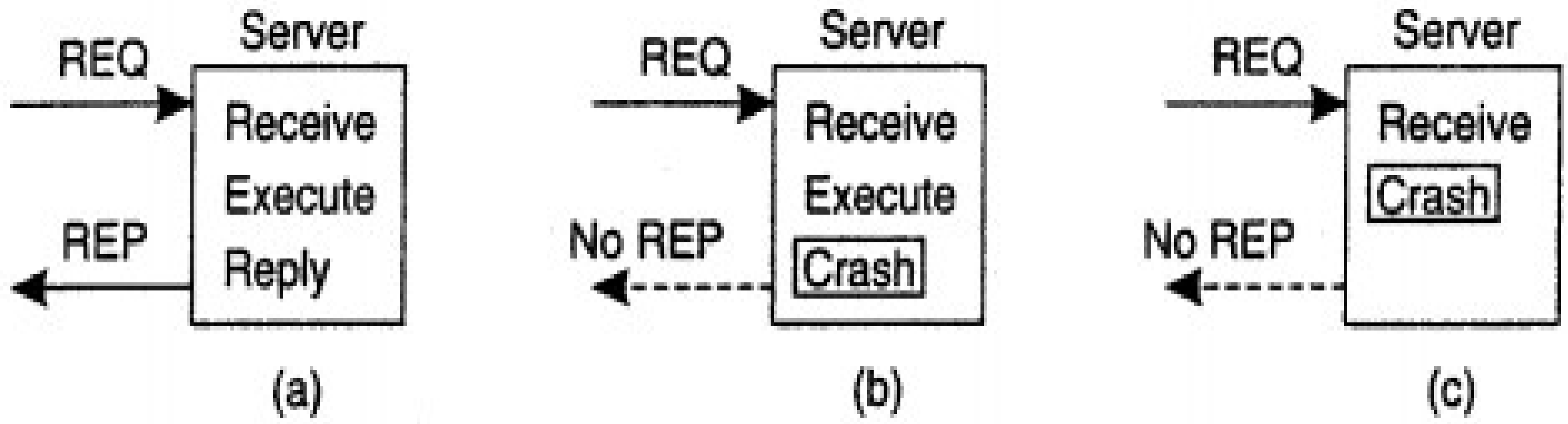
  - **to ensure the next blockchain is the one and only version of the truth and**
  - **to keep powerful adversaries from derailing a system and forking a chain.**

- PoW was designed for two things:
  - **to ensure the next blockchain is the one and only version of the truth and**
  - **to keep powerful adversaries from derailing a system and forking a chain.**
- PoW algorithm is not 100% tolerant to Byzantine faults, but due to the cost-intensive mining process and the underlying cryptographic techniques, it has proven to be one of the most secure and reliable implementations for blockchain networks.

- Nakamoto Consensus can be broken down into roughly four parts which are listed as follows:

  - **Block selection** : To “Mine” a page of the cryptocurrency ledger, known as a ‘block’, a node must **work hard** and figure out a cryptographic puzzle that is predicated on incrementing a nonce in the block until the correct value that represents the block’s hash and required zero bits for the beginning of the nonce is reached
  - **Incentive structure** :Once the equation is solved, the node wins the round of the lottery and is rewarded with currency, referred to as a ‘block reward’ (cryptocurrency). The nodes those who take part in the computation are called ‘miners. If one miner gets a different answer to the puzzle compared to others, their answers will be rejected
  - **Proof of Work (PoW):** As it costs a lot of electricity and computational power to mine for a block reward, lying is prevented as it will not pay off. Instead, miners do work, have proof of the work, and are rewarded.
  - **Scarcity**: A scarcity in Bitcoin is based on this premise by limiting the total number of Bitcoin that will be mined to 21 million. Additionally, Bitcoin can only be injected into a system through the mining process and it follows a deflationary scheme where the block reward is halved every 2,10,000 blocks (~4 years).
    - **Incentive structure** :Once the equation is solved, the node wins the round of the lottery and is rewarded with currency, referred to as a ‘block reward’ (cryptocurrency). The nodes those who take part in the computation are called ‘miners. If one miner gets a different answer to the puzzle compared to others, their answers will be rejected
    - Miners are incentivized to validate and secure a network honestly, as the reward that they receive for mining a block is Bitcoin. If the value of Bitcoin drops or the network becomes compromised.

RPC Semantics in the Presence of Failure



Server Management

- Server States
  - Stateful
  - Stateless
- Server Creation Semantics
  - Instance - Per - Call
  - Instance - Per – Session
  - Persistent Servers

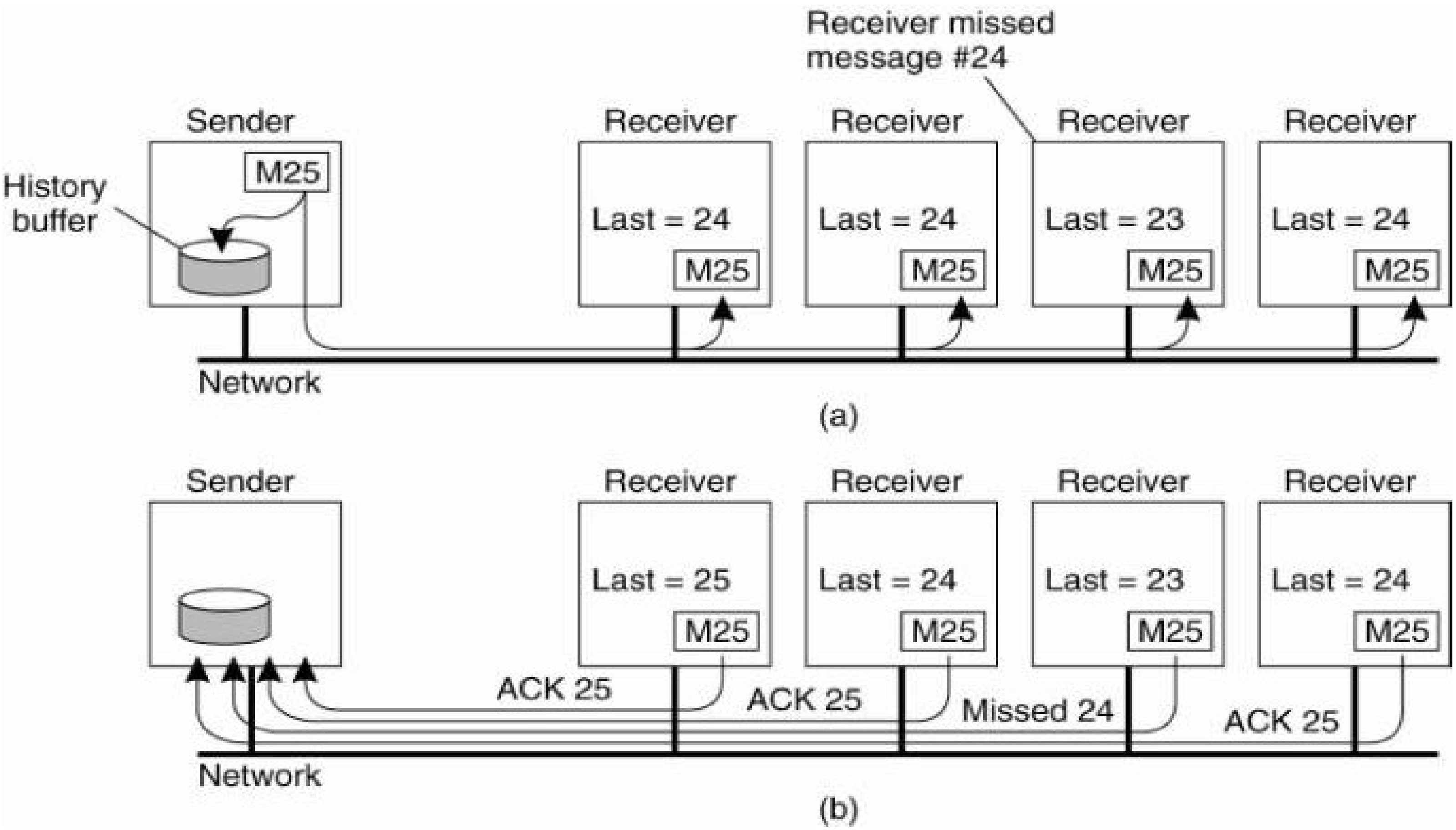
Parameter Passing

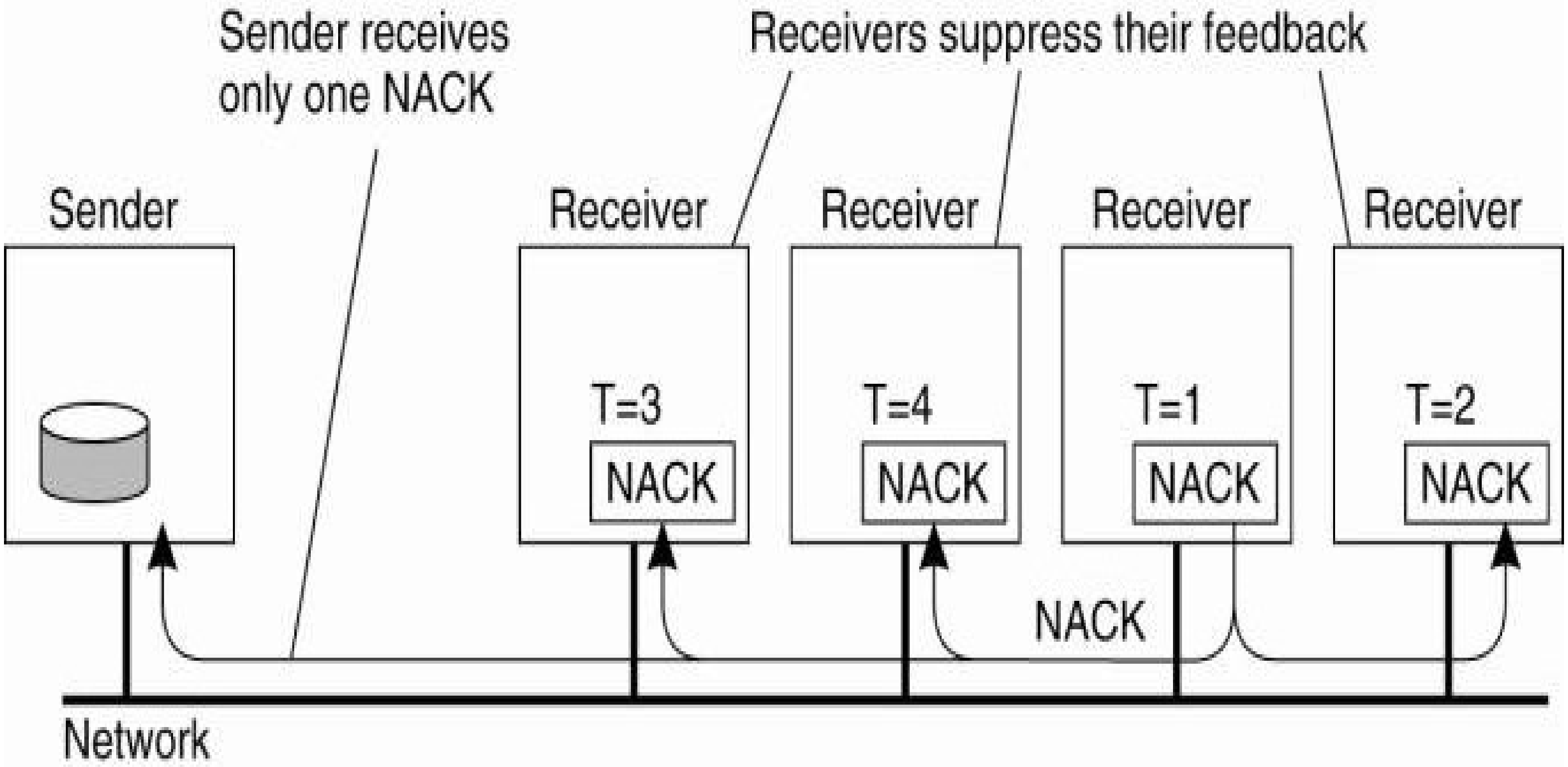
- Call – by – value
- Call by Reference

Call Semantics

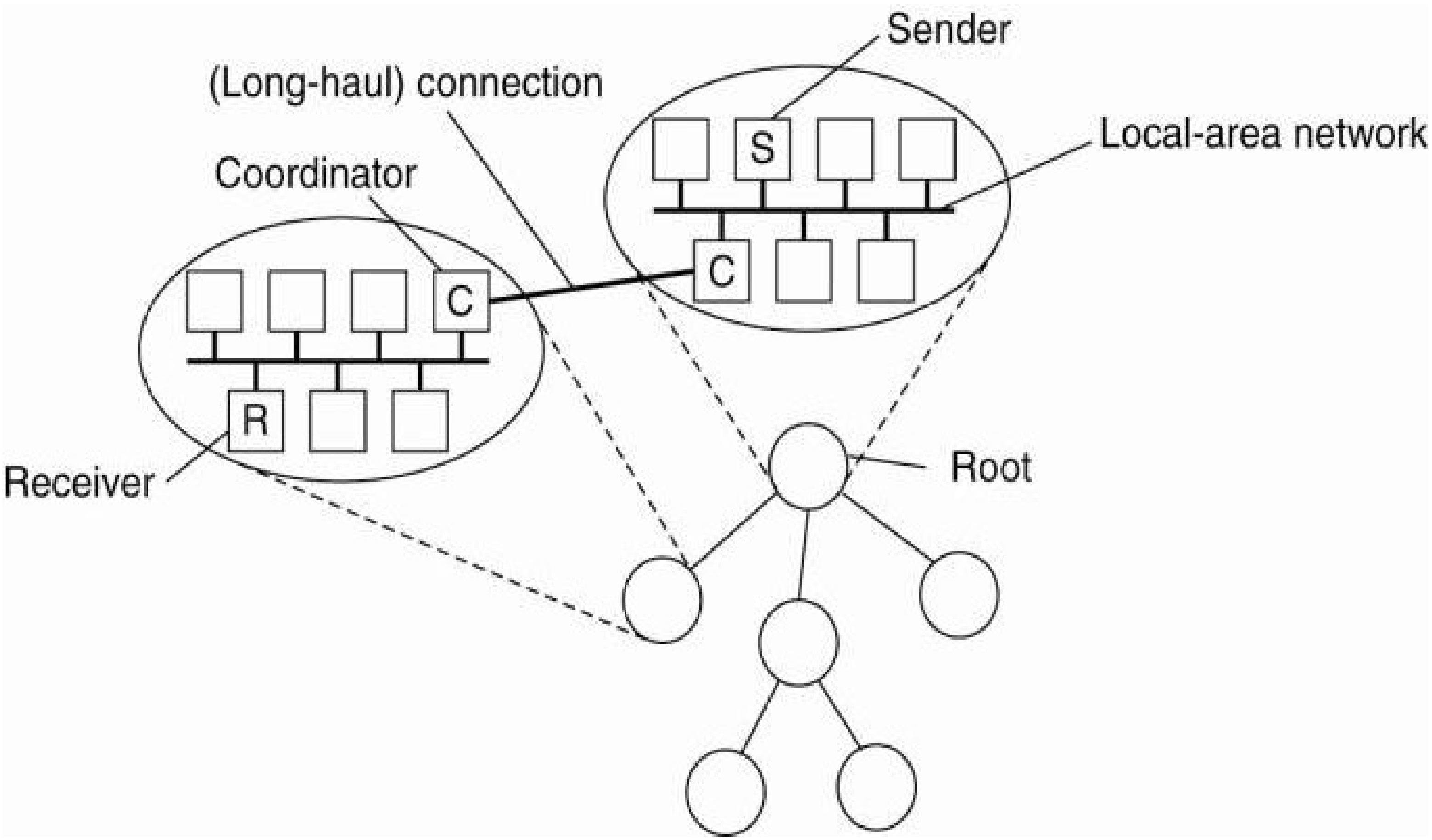
- May- be
- Last-one call
- Last – of –many
- At – least once
- Exactly once

Reliable group communication




Sender receives only one NACK
Receivers suppress their feedback
Sender
Receiver
Receiver
Receiver
Receiver
T=3
T=4
T=1
T=2
NACK
NACK
NACK
NACK
NACK
Network

Hierarchical feedback control



Virtual Synchrony View synchrony

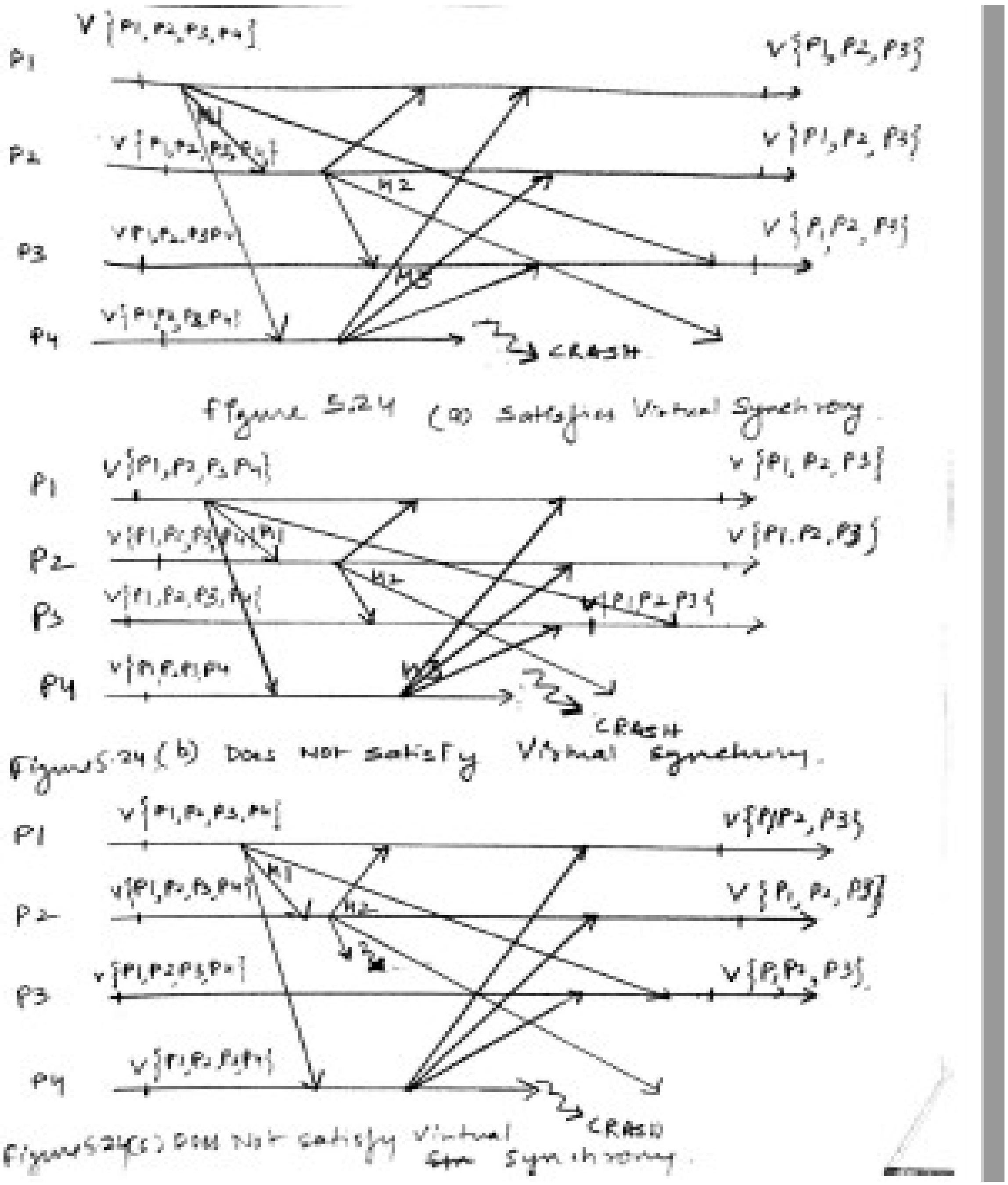


Figure 5.24 (a) Satisfies Virtual Synchrony

Figure 5.24 (b) Does Not Satisfy Virtual Synchrony

Figure 5.24 (c) Does Not satisfy Virtual Sm Synchrony

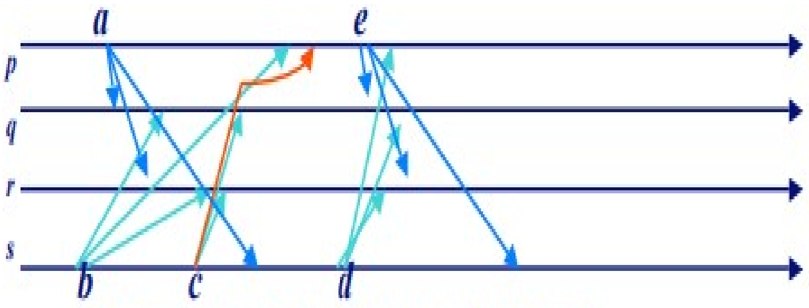
Message Ordering

- *Fifo* or *sender ordered* multicast

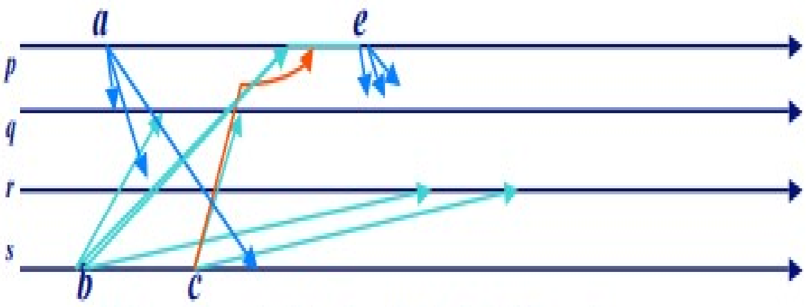
  Messages are delivered in the order they were sent (by any single sender)

- *Causal* or *happens-before* ordering:

  If send(a) → send(b) then deliver(a) occurs before deliver(b) at common destinations



delivery of c to p is delayed until after b is delivered



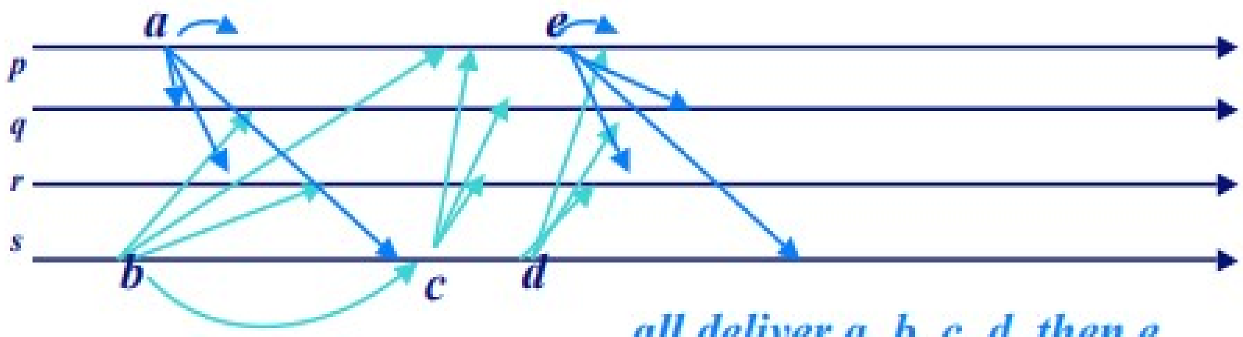
delivery of c to p is delayed until after b is delivered

e is sent (causally) after b

Message Ordering

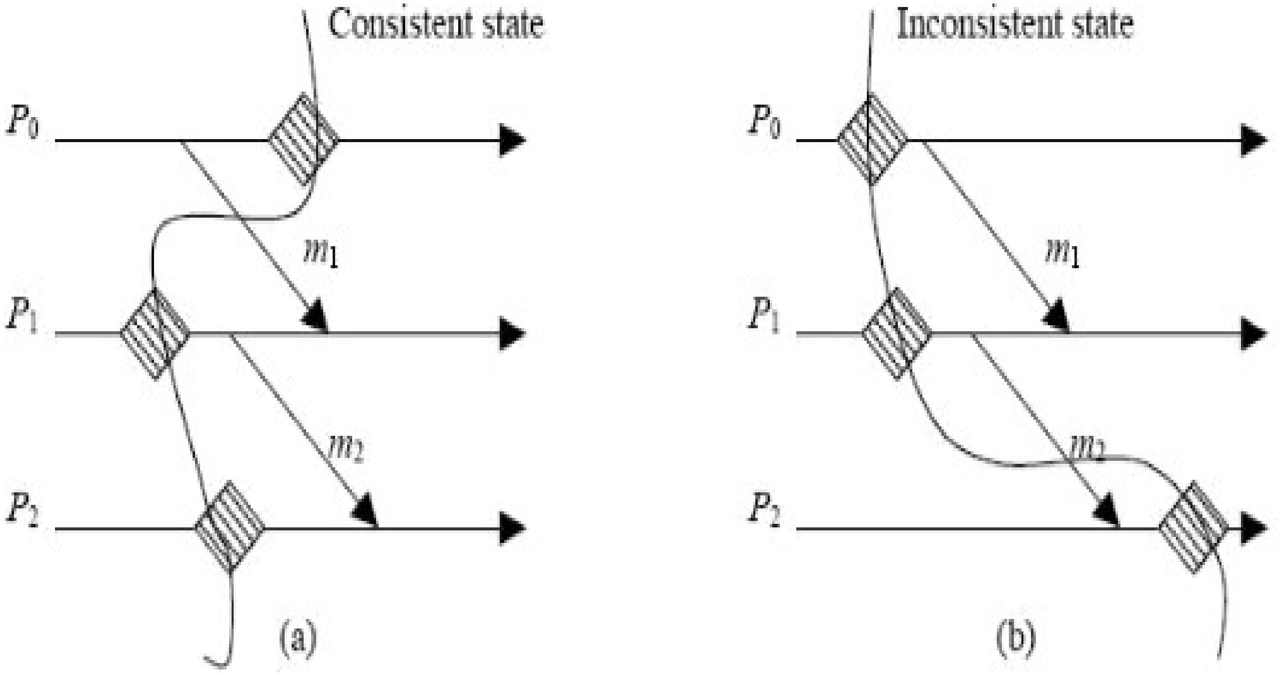
- *Total* or *locally total* multicast:

  Messages are delivered in same order to all recipients (including the sender)

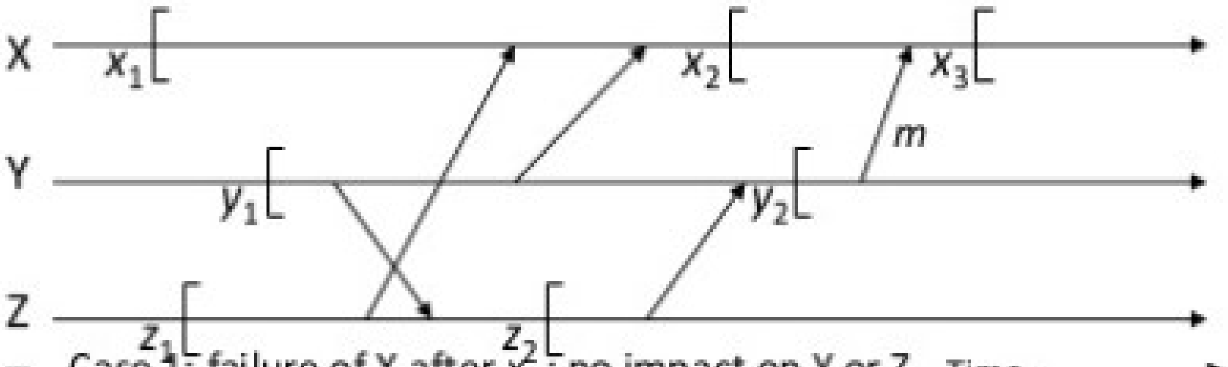


all deliver a, b, c, d, then e

Consistent Cut: Global state



Dominos Effect : Orphan management



- Case 1: failure of X after x_3 : no impact on Y or Z
- Case 2: failure of Y after sending msg. '*m*'
  - Y rolled back to y_2
  - 'm' ≡ orphan massage
  - X rolled back to x_2
- Case 3: failure of Z after z_2
  - Y has to roll back to y_1
  - X has to roll back to x_1
  - Z has to roll back to z_1

Domino Effect