Chapter 5

Consistency, Replication and Fault Tolerance

Content….

5.1 Introduction to replication and consistency, Data-Centric and Client-Centric Consistency Models, Replica Management

5.2 Fault Tolerance: Introduction, Process resilience, Reliable client-server and group communication, Recovery

Reasons for Replication

- Data are replicated to increase the reliability of a system.
- Replication for performance:
  - Scaling in numbers
  - Scaling in geographical area

- Caveat/Caution :
  - Gain in performance
  - Cost of increased bandwidth for maintaining replication

More on Replication

- Replicas allows remote sites to continue working in the event of local failures.
- It is also possible to protect against data corruption.
- Replicas allow data to reside close to where it is used.
- Even a large number of replicated “local” systems can improve performance: think of clusters.
- This directly supports the distributed systems goal of enhanced scalability.

Replication and Scalability

- Replication is a widely-used scalability technique: think of Web clients and Web proxies.
- When systems scale, the first problems to surface are those associated with performance – as the systems get bigger (e.g., more users), they get often slower.
- Replicating the data and moving it closer to where it is needed helps to solve this scalability problem.
- A problem remains: how to efficiently synchronize all of the replicas created to solve the scalability issue?
- Dilemma: adding replicas improves scalability, but incurs the (oftentimes considerable) overhead of keeping the replicas up-to-date!!!
- As we shall see, the solution often results in a relaxation of any consistency constraints.

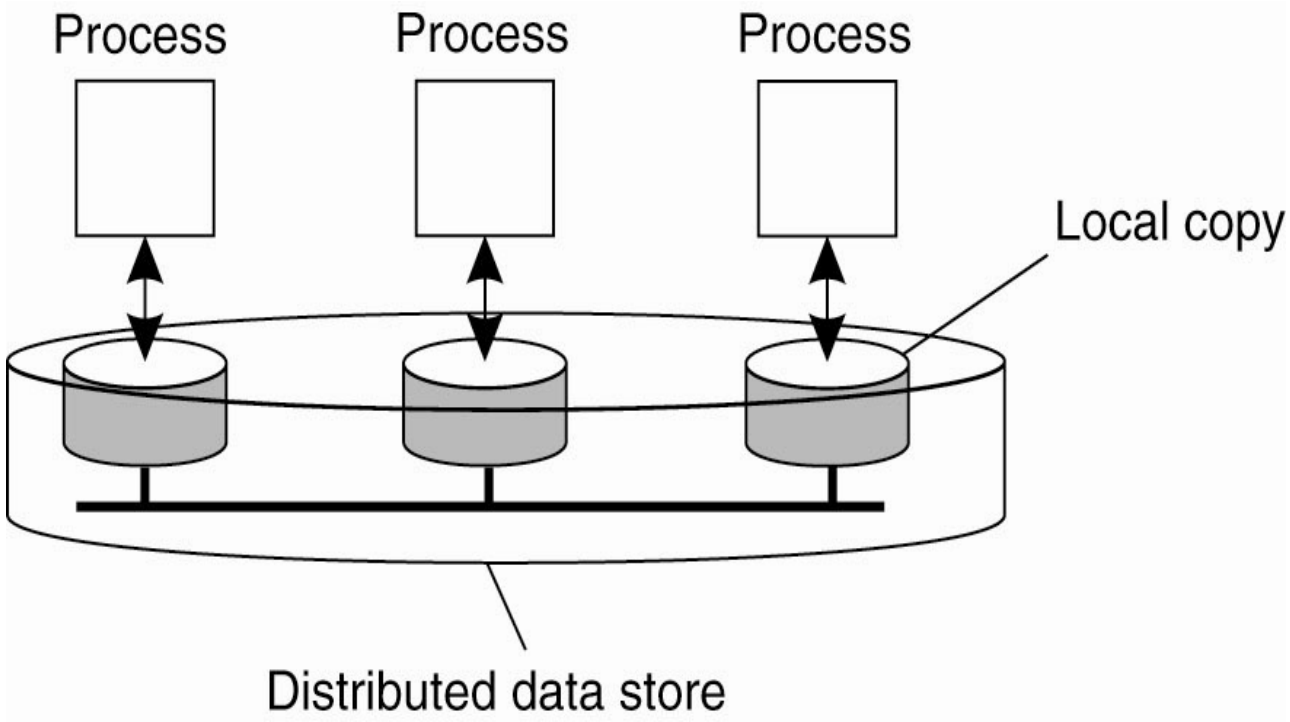
Replication and Consistency

- But if there are many replicas of the same thing, how do we keep all of them up-to-date? How do we keep the replicas *consistent*?
- Consistency can be achieved in a number of ways, however, it is not easy to keep all those replicas consistent
- We will study a number of ***consistency models***

Replication and Consistency Models

- Data-Centric Consistency Models
- Client-Centric Consistency Models,

Data-centric Consistency Models



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

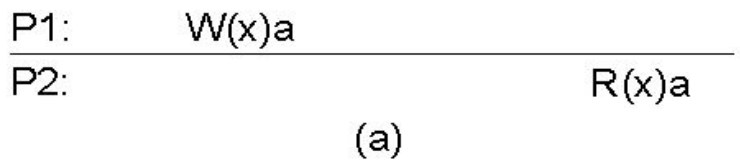
- A “consistency model” is a CONTRACT between a DS data-store and its processes.
- If the processes agree to the rules, the data-store will perform properly and as advertised.

Consistency Model Diagram Notation

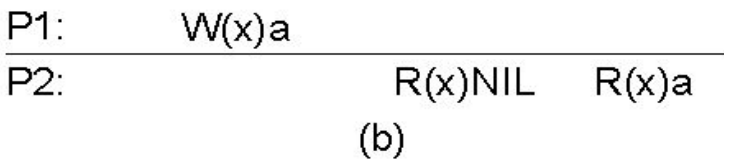
- $W_i(x)a$ – a write by process ‘i’ to item ‘x’ with a value of ‘a’. That is, ‘x’ is set to ‘a’.

  (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



(a)



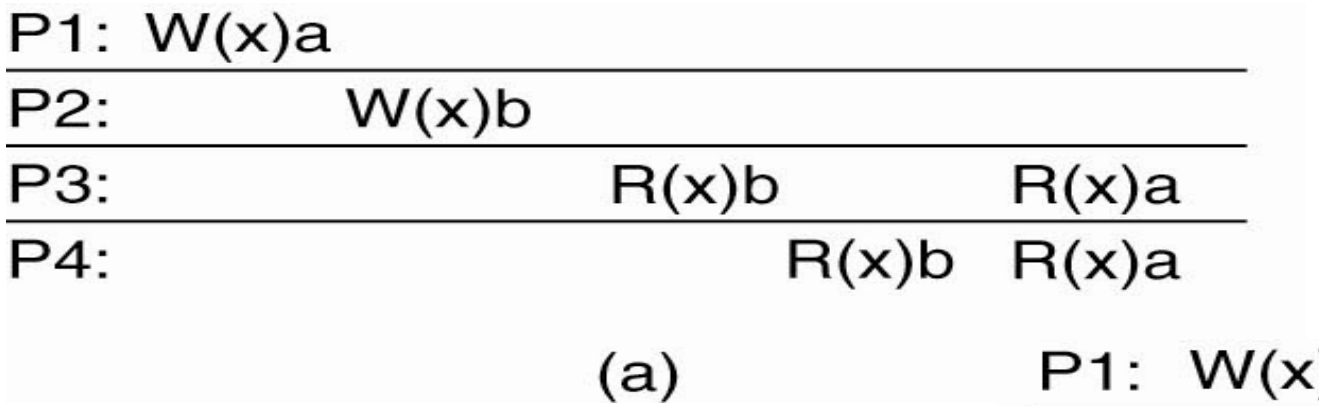
(b)

- Behavior of two processes, operating on same data item:
  - A strictly consistent data-store.
  - 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 DS. This is the consistency model "Holy Grail" – *not at all easy in the real world*, and all but *impossible* within a DS.
- So, other, ***Weaker Model (or "less strict")*** models have been developed which has 2 types ***Sequential and Casual Consistency Models***.
- A weaker consistency model, which represents a relaxation of the rules. It is also possible to implement.

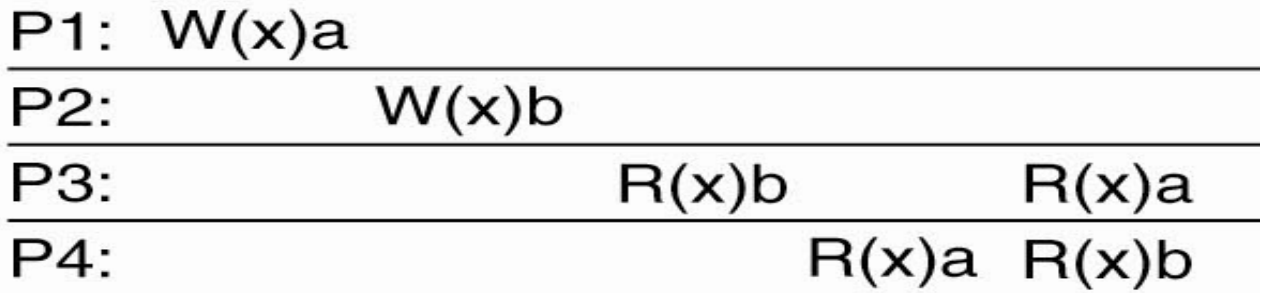
Sequential Consistency (1)

- 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 (2)



(a)



(b)

(a) A sequentially consistent data store.
(b) A data store that is not sequentially consistent.

Sequential Consistency (3)

Process P1	Process P2	Process P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

Three concurrently-executing processes

Sequential Consistency (4)

```
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)

Four valid execution sequences for these processes. The vertical axis is time.

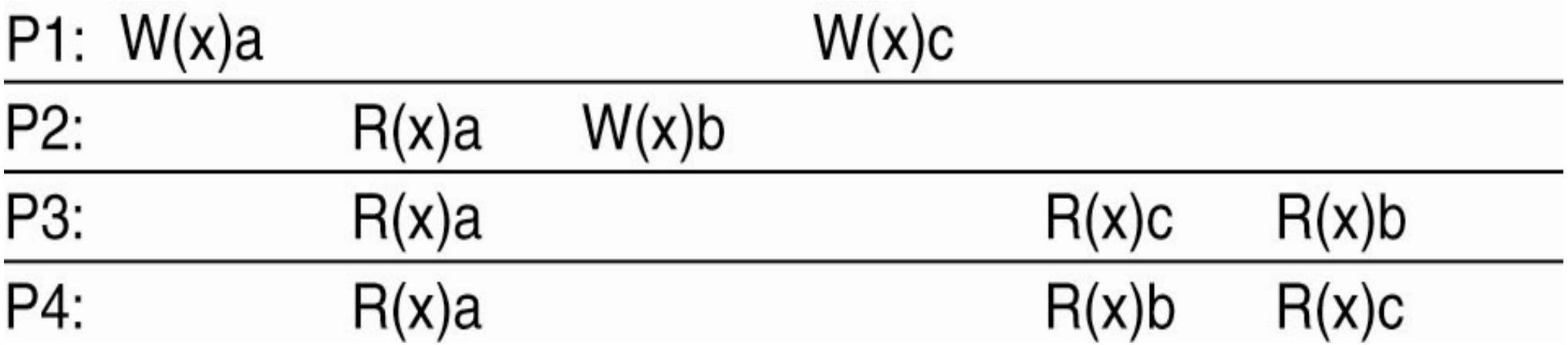
Causal Consistency

- 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 (1)

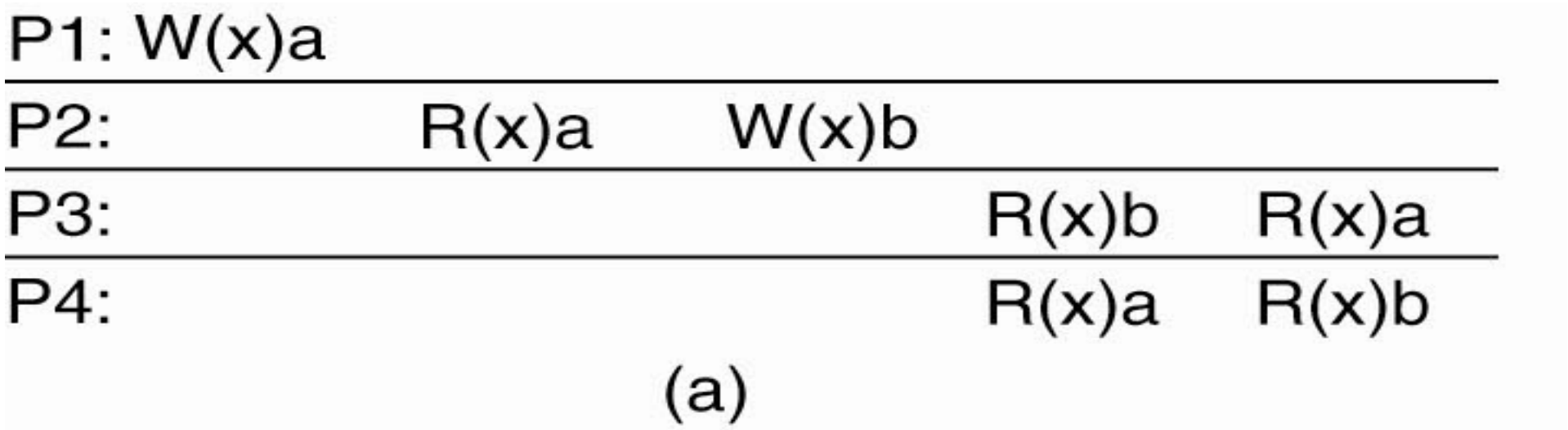
- For a data store to be considered causally consistent, it is necessary that the store obeys the following conditions:
  - Writes that are potentially causally related
    - must be seen by all processes in the same order.
  - Concurrent writes
    - may be seen in a different order on different machines.

Causal Consistency



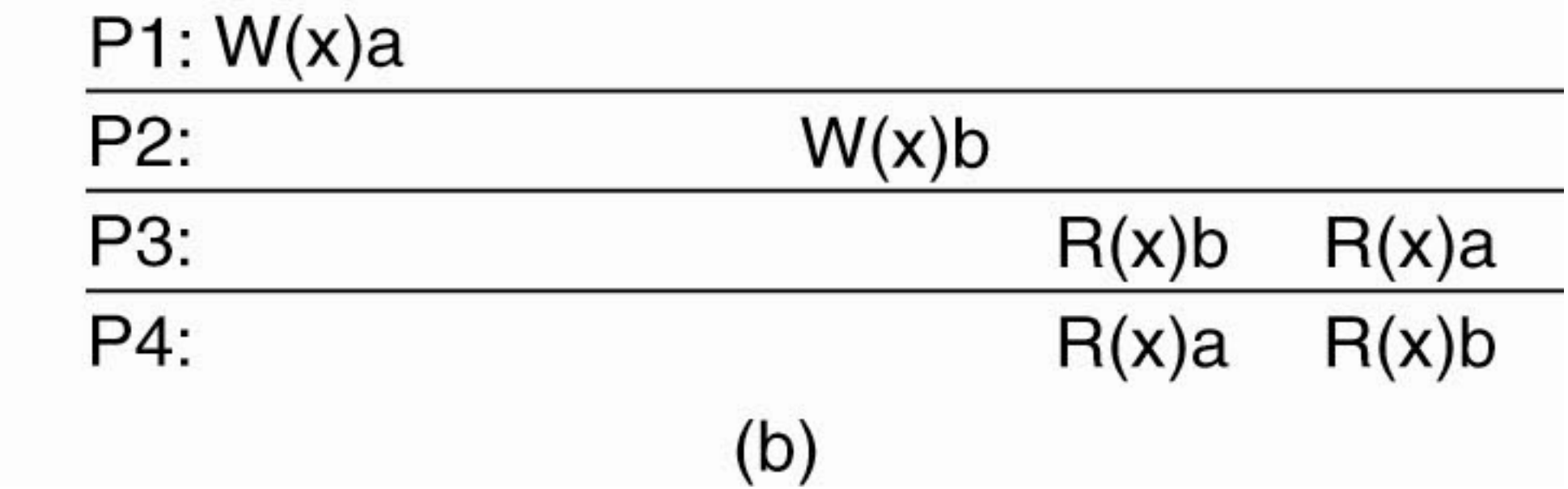
This sequence is allowed with a
causally-consistent store, but not with a sequentially consistent store

Causal Consistency



(a) A violation of a causally-consistent store

Causal Consistency (3)



(b) A correct sequence of events in a causally-consistent store

Grouping Operations

- Accesses to locks are sequentially consistent.
- No access to a lock is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to locks have been performed.

Basic Idea:

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

Grouping Operations

- 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 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.

Client-Centric Consistency Models

- The previously studied consistency models concern themselves with maintaining a consistent (globally accessible) data-store in the presence of concurrent read/write operations.

- Here, the emphasis is more on maintaining a consistent view of things ***for the individual client process*** that is currently operating on the data-store.

More Client-Centric Consistency

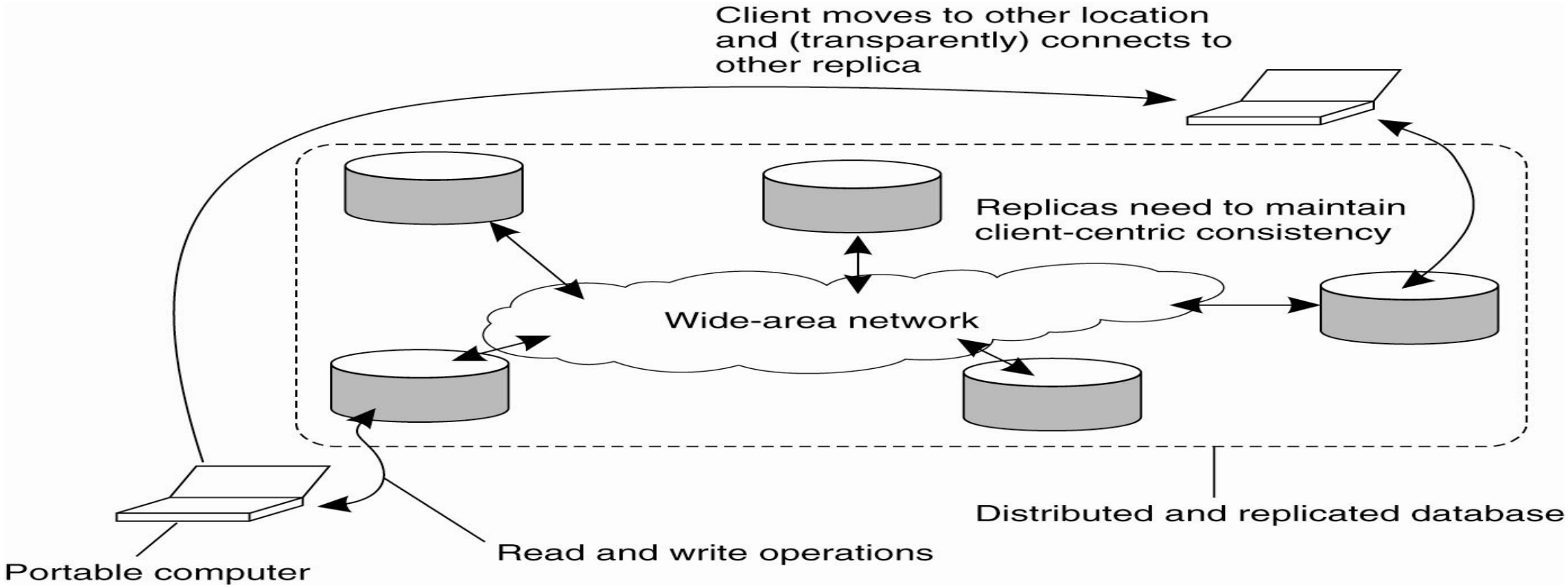
- How fast should updates (writes) be made available to read-only processes?
  - Think of most database systems: *mainly read*.
  - Think of the DNS: *write-write conflicts* do no occur, only *read-write conflicts*.
  - Think of WWW: as with DNS, except that heavy use of client-side caching is present: *even the return of stale pages is acceptable to most users*.

- These systems all exhibit a high degree of acceptable inconsistency … with the *replicas* gradually becoming consistent over time.

Toward Eventual Consistency

- The only requirement is that all replicas will *eventually* be the same.
- All updates must be guaranteed to propagate to all replicas … *eventually*!
- This works well if every client always updates the same replica.
- Things are a little difficult if the clients are *mobile*.

Eventual Consistency



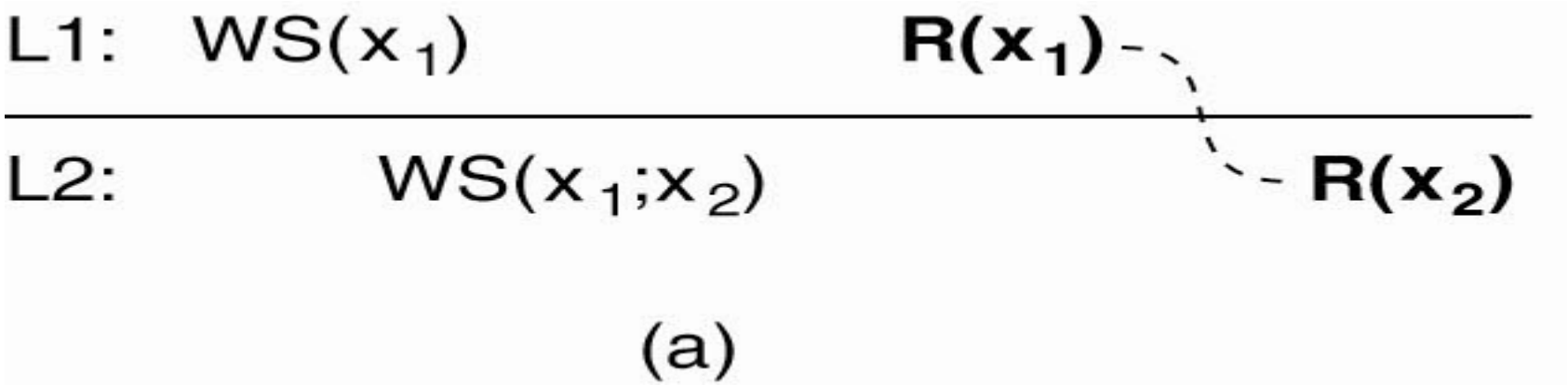
The principle of a mobile user accessing different replicas of a distributed database

Monotonic Reads (1)

A data store is said to provide monotonic-read consistency if the following condition holds:

- 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.

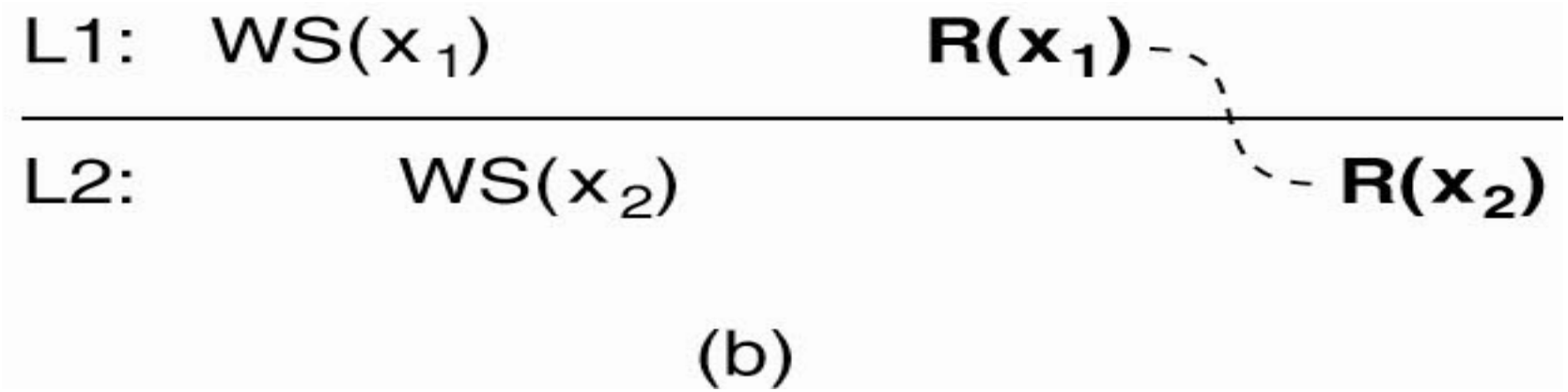
Monotonic Reads (2)



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.

Monotonic Reads (3)



(b)

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

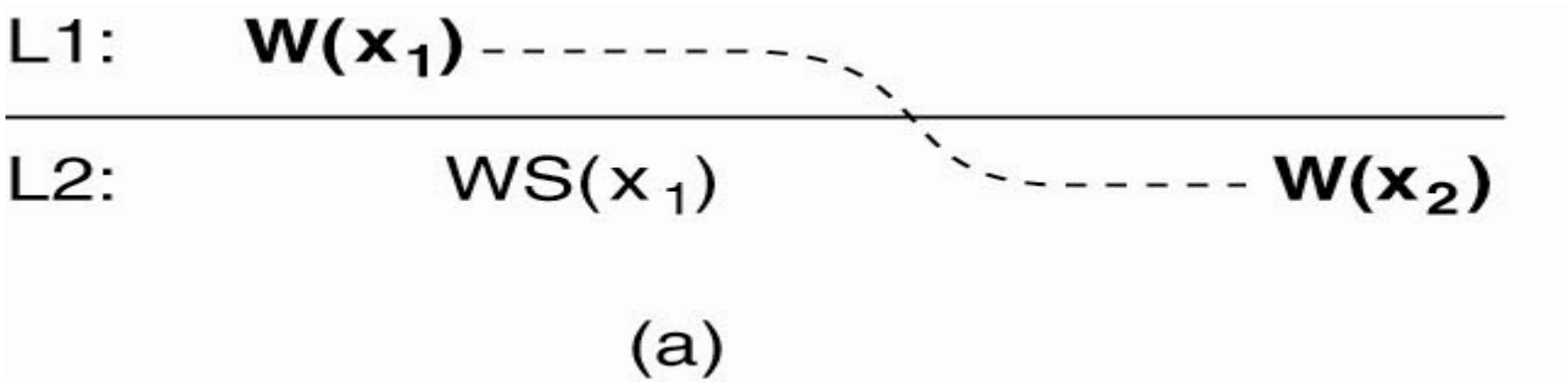
(b) A data store that does not provide monotonic reads.

Monotonic Writes (1)

In a monotonic-write consistent store, the following condition holds:

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

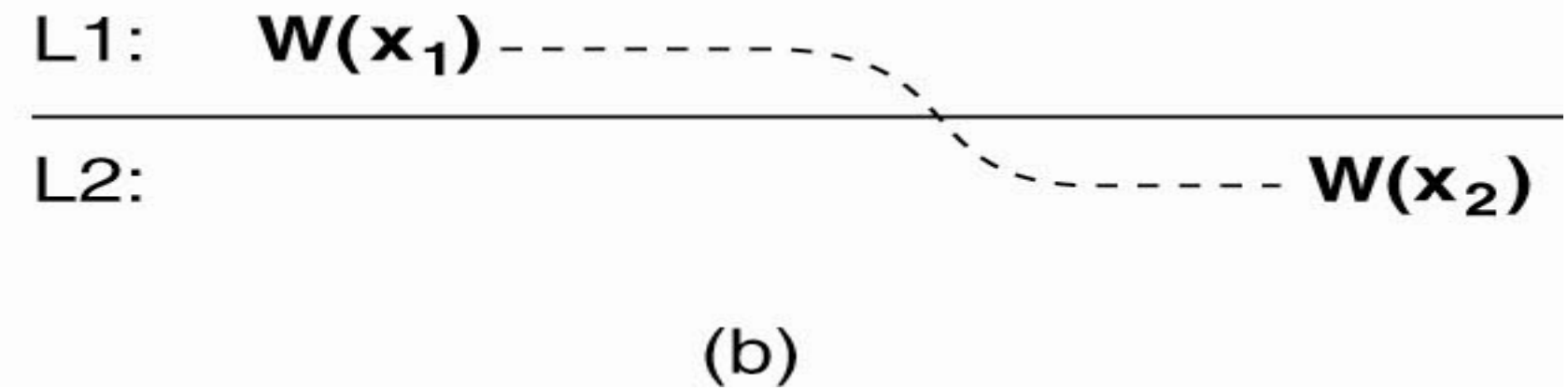
Monotonic Writes (2)



(a)

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.

Monotonic Writes (3)



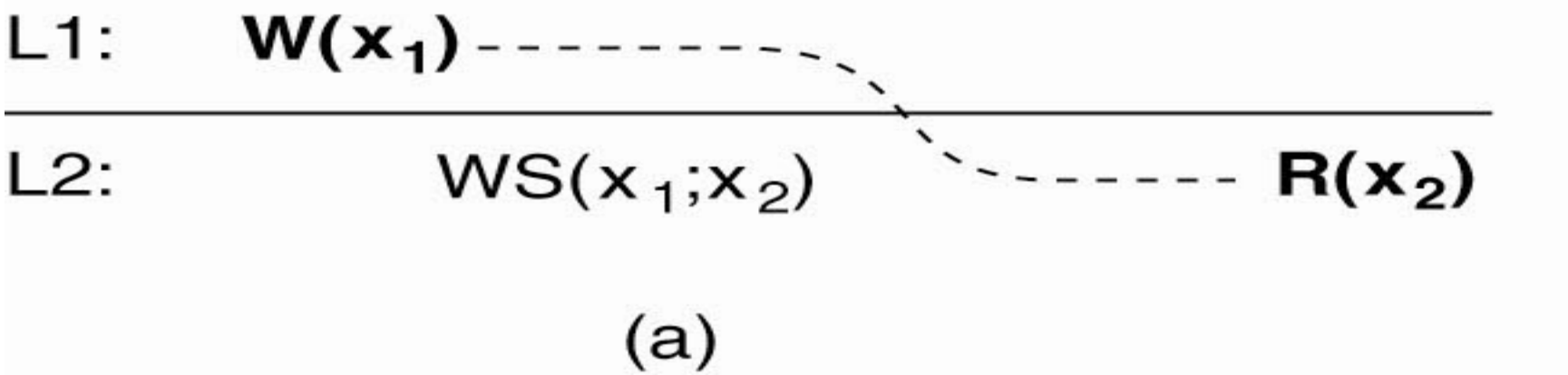
The write operations performed by a single process P at two different local copies of the same data store.

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

Read Your Writes (1)

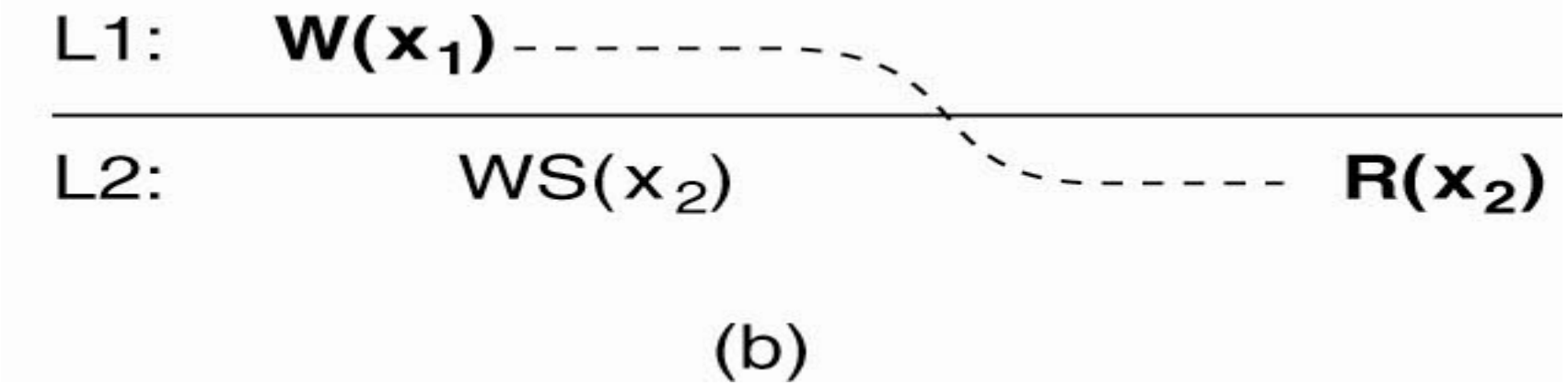
- A data store is said to provide read-your-writes consistency, if the following condition holds:
  - The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

Read Your Writes (2)



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

Read Your Writes (3)



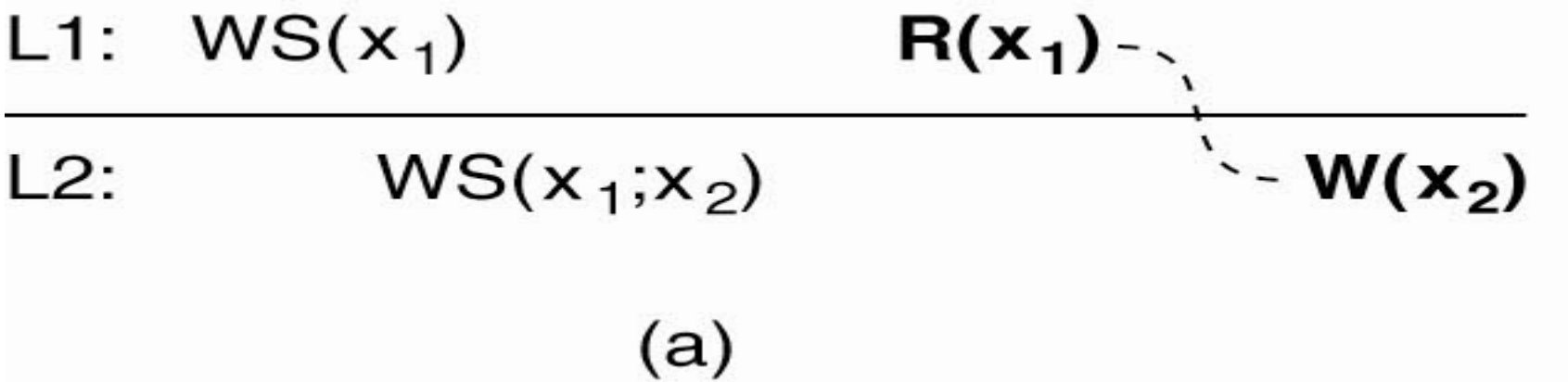
(b) A data store that does not

Writes Follow Reads (1)

- A data store is said to provide writes-follow-reads consistency, if the following holds:

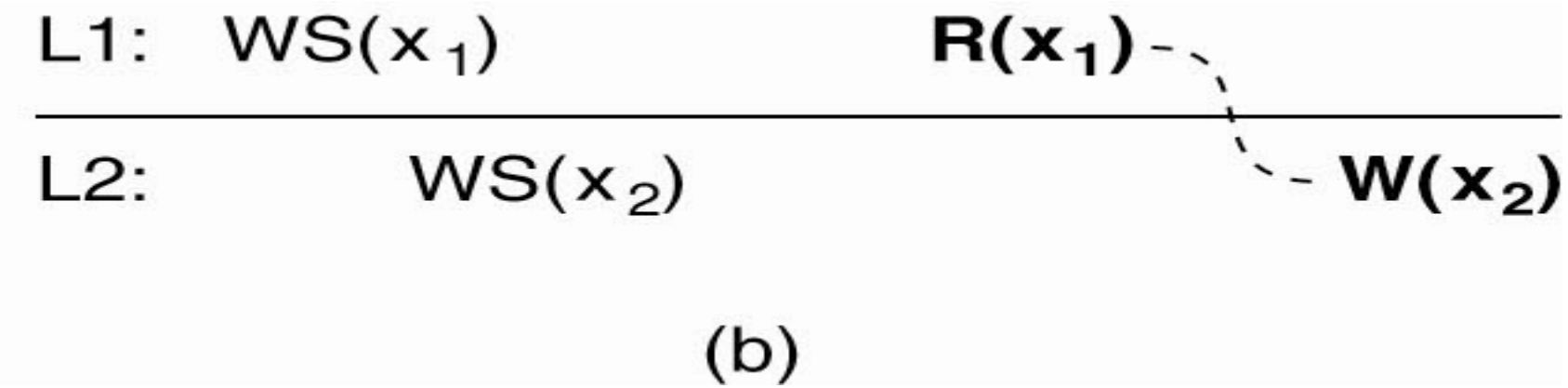
  A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.

Writes Follow Reads (2)



(a) A writes-follow-reads consistent data store

Writes Follow Reads (3)

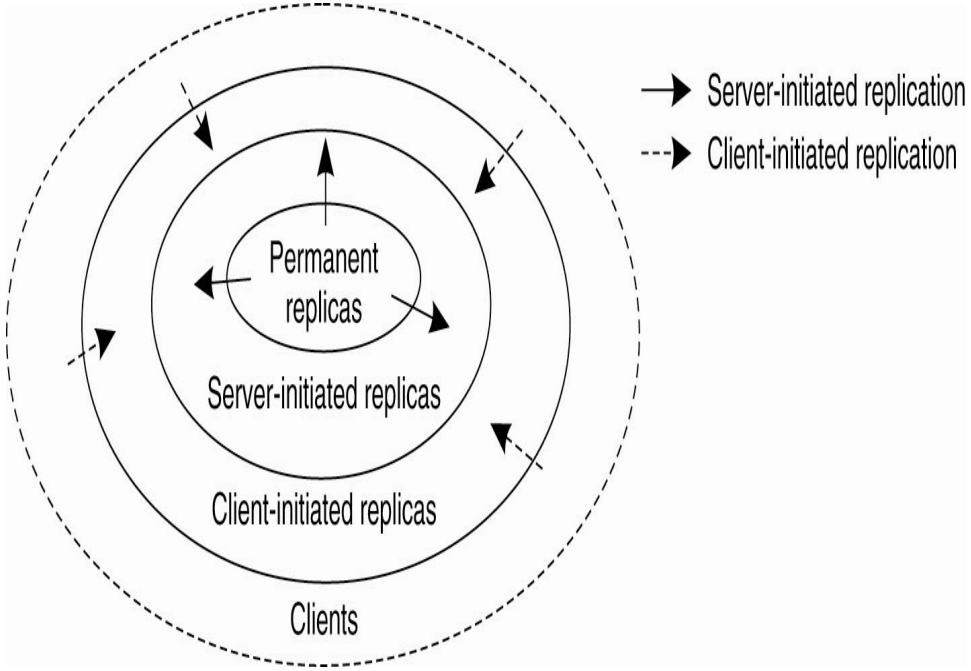


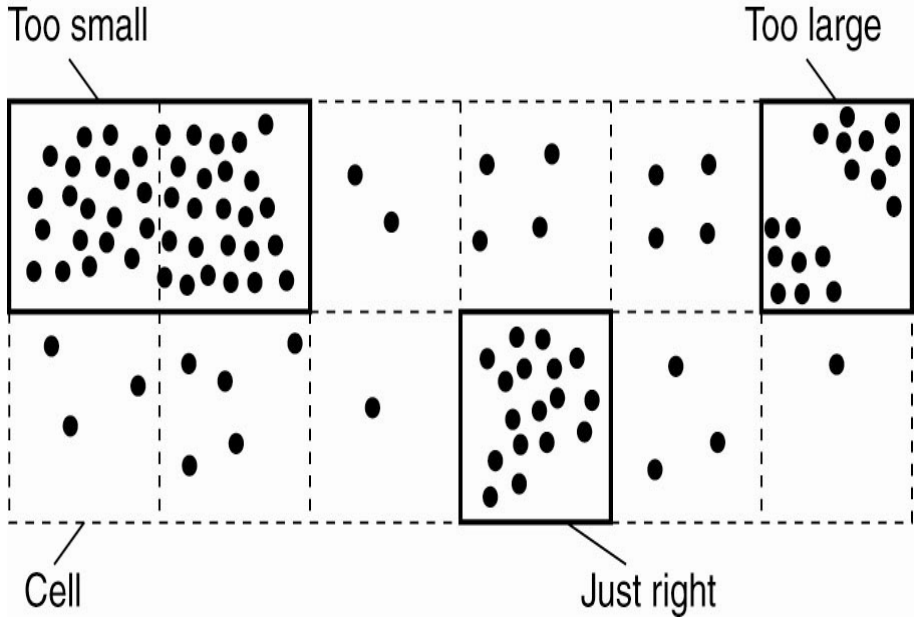
(b) A data store that does not provide writes-follow-reads consistency

Replica Management

Content Replication and Placement

Regardless of which consistency model is chosen, we need to decide ***where***, ***when*** and ***by whom*** copies of the data-store are to be placed.





Choosing a proper cell size for server placement

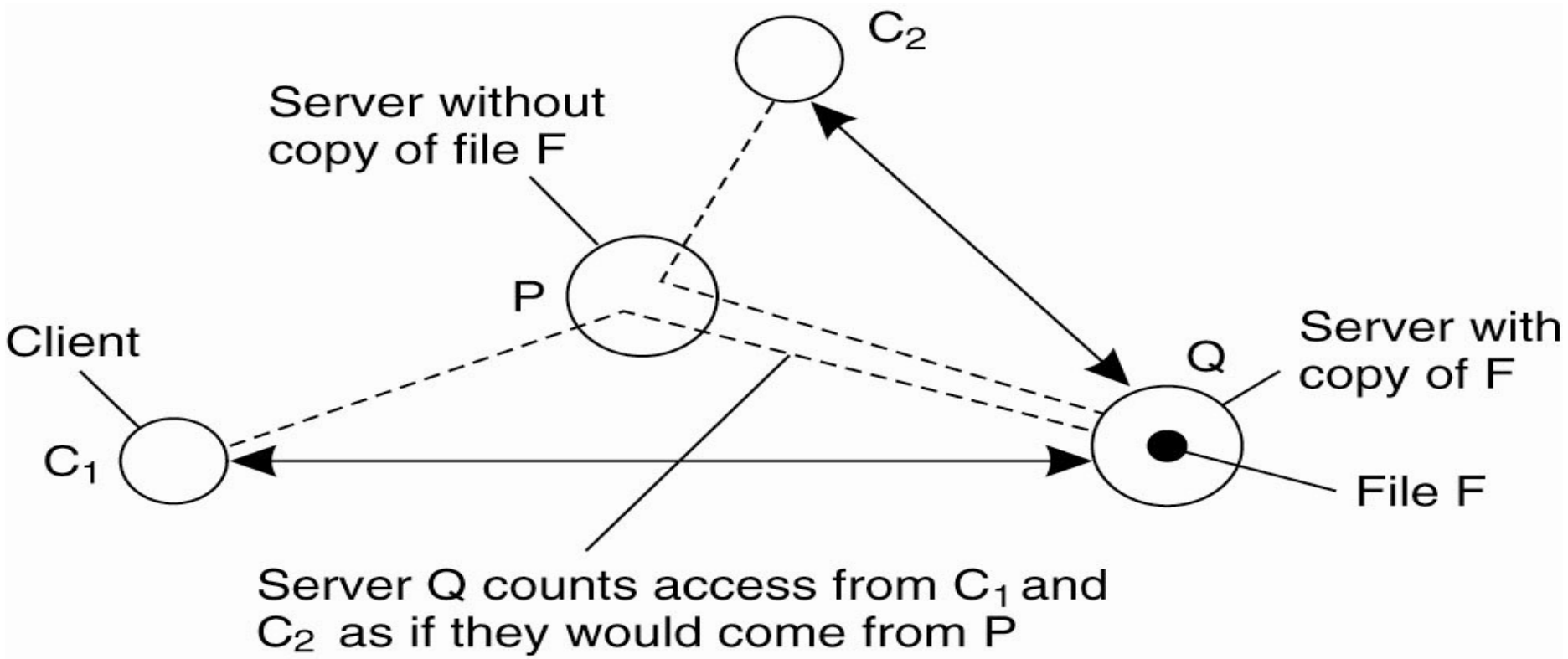
Replica Management

There are three Replica Placement Types

1. ***Permanent replicas***: tend to be small in number, organized as COWs (Clusters of Workstations) or mirrored systems.
2. ***Server-initiated replicas***: used to enhance performance at the initiation of the owner of the data-store. Typically used by web hosting companies to geographically locate replicas close to where they are needed most. (Often referred to as "push caches").
3. ***Client-initiated replicas***: created as a result of client requests – think of browser caches. Works well assuming, of course, that the cached data does not go *stale* too soon.

Replica Management

Server-Initiated Replicas



Counting access requests from different clients

Replica Management

Client-Initiated Replicas

- When a client initiates an update to a distributed data-store, what gets propagated?
- There are three possibilities:
  - Propagate *notification* of the update to the other replicas – this is an "invalidation protocol" which indicates that the replica's data is no longer up-to-date. Can work well when there's many writes.
  - Transfer the *data* from one replica to another – works well when there's many reads.
  - Propagate the *update* to the other replicas – this is "active replication", and shifts the workload to each of the replicas upon an "initial write".

Replica Management

Push vs. Pull Protocols

Another design issue relates to whether or not the updates are *pushed* or *pulled*?

1. ***Push-based/Server-based Approach***: sent "automatically" by server, the client does *not* request the update. This approach is useful when a high degree of consistency is needed. Often used between permanent and server-initiated replicas.
2. ***Pull-based/Client-based Approach***: used by client caches (e.g., browsers), updates are requested by the client from the server. No request, no update!

5.2 Fault Tolerance

- Introduction,
- Process resilience,
- Reliable client-server and group communication, Recovery

Basic Concepts (1/3)

- **What is Failure?**
  - System is said to be in failure state when it cannot meet its promise.
- **Why do Failure occurs?**
  - Failures occurs because of the error state of the system.
- **What is the reason for Error?**
  - The cause of an error is called a fault
- **Is there some thing 'Partial Failure'?**
- **Faults can be Prevented, Removed and Forecasted.**
- **Can Faults be Tolerated by a system also?**

Basic Concepts (2/3)

- **What characteristics makes a system Fault Tolerant?**
  - **Availability:** System is ready to used immediately.
  - **Reliability:** System can run continuously without failure.
  - **Safety:** Nothing catastrophic happens if a system temporarily fails.
  - **Maintainability:** How easy a failed system can be repaired.

- **What is the availability and reliability of following systems?**
  - If a system goes down for one millisecond every hour
  - If a System never crashes but is shut down for one weeks every March.

Basic Concepts (3/3)

- **Classification of Faults**
  - **Transient:** Occurs once and than disappears (For example, a fault in the network might result in a request that is being sent from one node to another to time out or fail.)
  - **Intermittent:** Occurs, vanishes on its own accord, than reappears and so on (eg medical life support equipment)
  - **Permanent:** They occurs and doesn't vanish until fixed manually.
- **Can you classify the Faults caused by following situations?**
  - **A flying bird obstructing the transmitting waves signals**
  - **A loosely connected power plug**
  - **Burnt out chips**
  - **Software Bugs**
- **Which Fault you think is more difficult to detect and why?**

Faults in Distributed Systems

- If in a Distributed Systems some fault occurs, the error may by in any of
  - The collection of servers or
  - Communication Channel or
  - Even both
- However, out of order server itself may not always be the fault we are looking for. Why?
- Dependency relations appear in abundance in DS.
- Hence, we need to classify failures to know how serious a failure actually is.

Failure Models

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* •*State Transition*	The 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

Failure Masking by Redundancy (1/3)

- A system to be fault tolerant, the best it can do is try to hide the occurrence of failure from other processes
- Key technique to masking faults is to use Redundancy.
  - **Information redundancy:** Extra bits are added to allow recovery from garbled bits
  - **Time redundancy:** An action is performed, and then, if need be, it is performed again.
  - **Physical redundancy:** Extra equipment or processes are added
- Issue: How much redundancy is needed?

Failure Masking by Redundancy (2/3)

- Some Examples of Redundancy Schemes
  - Hamming Code
  - Transactions
  - Replicated Processes or Components
  - Aircraft has four engines, can fly with only three
  - Sports game has extra referee.

Failure Masking by Redundancy (3/3)

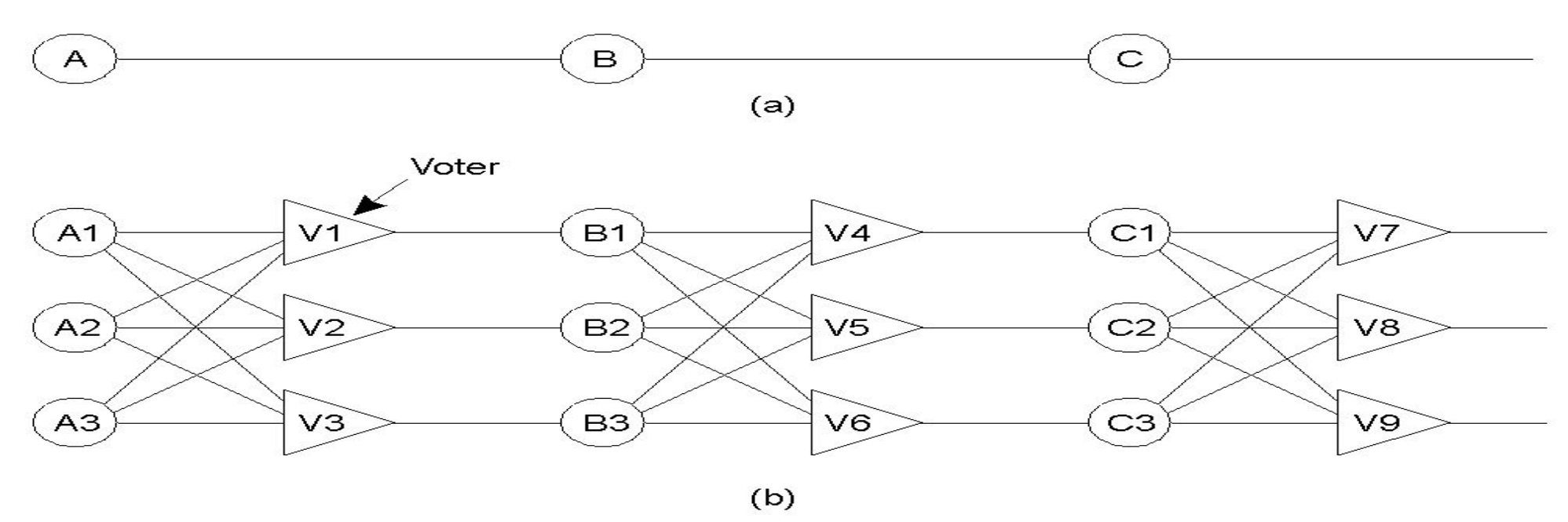


Figure: Fault Tolerance in Electronic Circuits

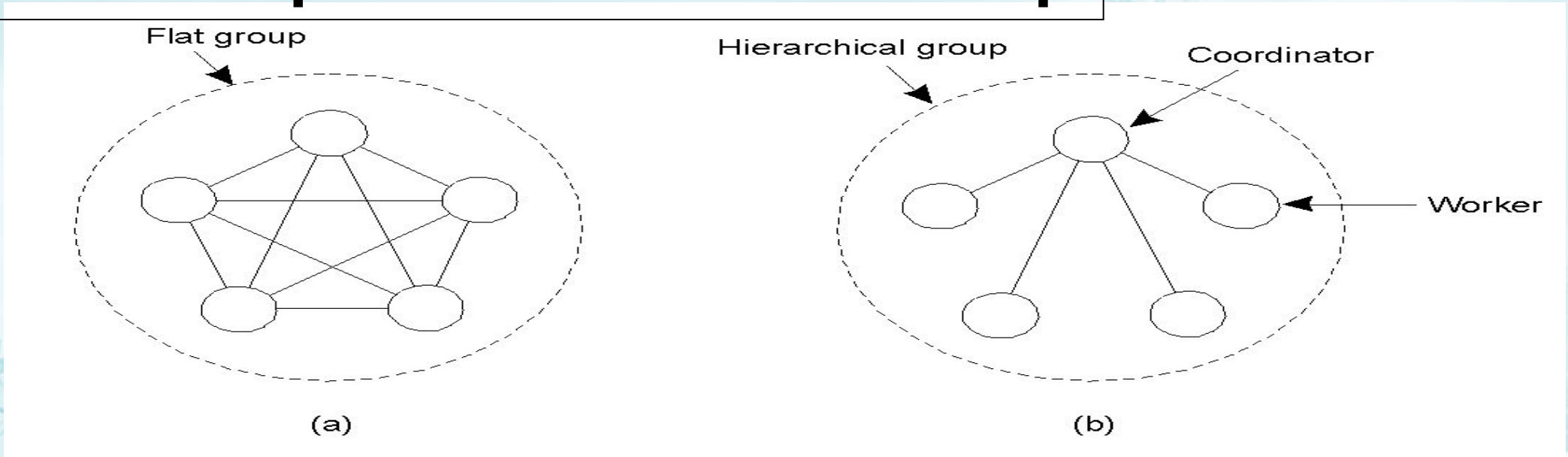
- **Triple modular redundancy:**
  - If two or three of the input are the same, the output is equal to that input.
  - If all three inputs are different, the output is undefined.

Process Resilience

- **Problem:**
  - How fault tolerance in distributed system is achieved, especially against Process Failures?
- **Solution:**
  - Replicating processes into groups.
  - Groups are analogous to Social Organizations.
  - Consider collections of process as a single abstraction
  - All members of the group receive the same message, if one process fails, the others can take over for it.
  - Process groups are dynamic and a Process can be member of several groups.
  - Hence we need some management scheme for groups.

Process Groups (1/2)

Flat Group vs. Hierarchical Group



- **Flat Group**
  - **Advantage:** Symmetrical and has no single point failure
  - **Disadvantage:** Decision making is more complicated. Voting
- **Hierarchical Group**
  - **Advantage:** Make decision without bothering others
  - **Disadvantage:** Lost coordinator Entire group halts

Process Groups (2/2)

Group Membership

- Group Server (Client Server Model)
  - Straight forward, simple and easy to implement
  - Major disadvantage Single point of failure
- Distributed Approach (P2P Model)
  - Broadcast message to join and leave the group
  - In case of fault, how to identify between a really dead and a dead slow member
  - Joining and Leaving must be synchronized on joining send all previous messages to the new member
  - Another issue is how to create a new group?

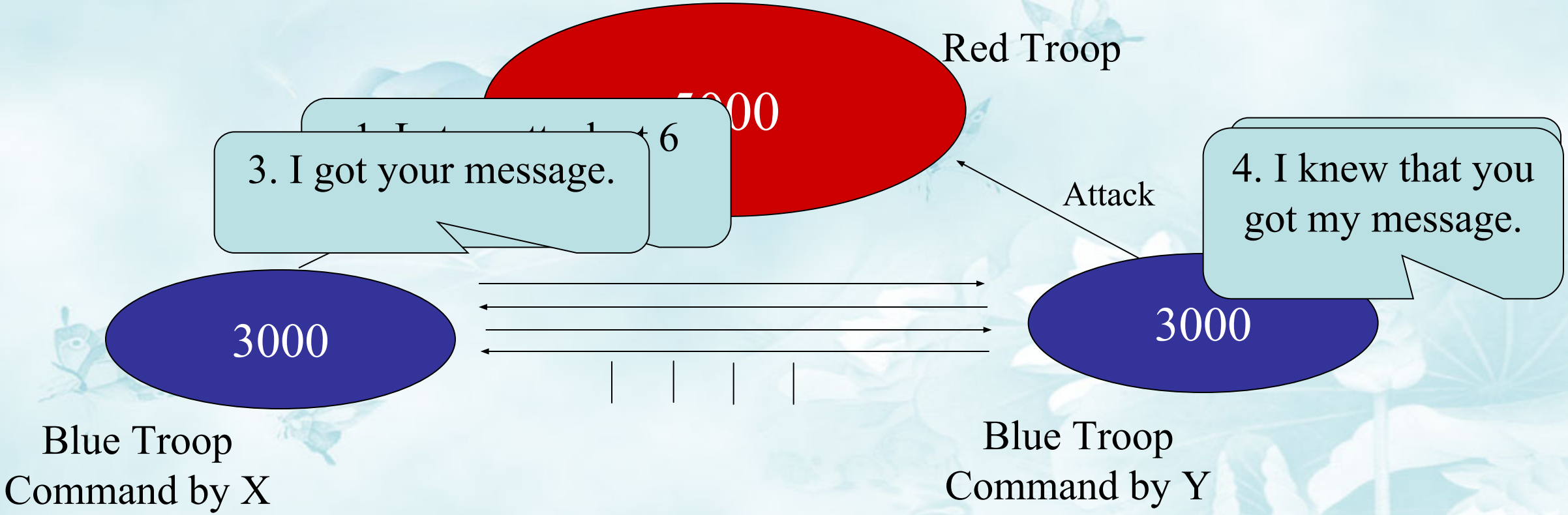
Failure Masking & Replication

- Replicate Process and organize them into groups
- Replace a single vulnerable process with the whole fault tolerant Group
- A system is said to be **K fault tolerant** if it can survive faults in ***K*** components and still meet its specifications.
- How much replication is needed to support **K Fault Tolerance**?
  - K+1 or 2K+1 ?

Agreement in Faulty Systems

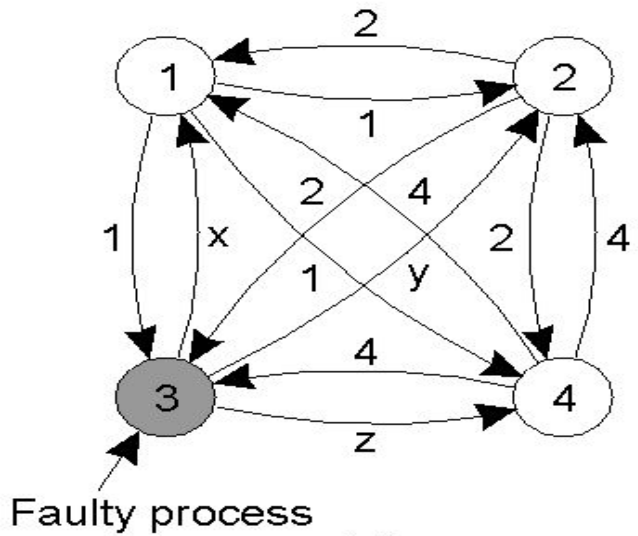
- Why we need Agreements?
- Goal of Agreement
  - Make all the non-faulty processes reach consensus on some issue
  - Establish that consensus within a finite number of steps.
- Problems of two cases
  - Good process, but unreliable communication
    - Example: Two-army problem
  - Good communication, but crashed process
    - Example: Byzantine generals problem

Two-army problem



→ It is easy to show that X and Y will never reach agreement, no matter how many acknowledgements they send. (Due to unreliable communication).

Byzantine generals problem



(a)

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)

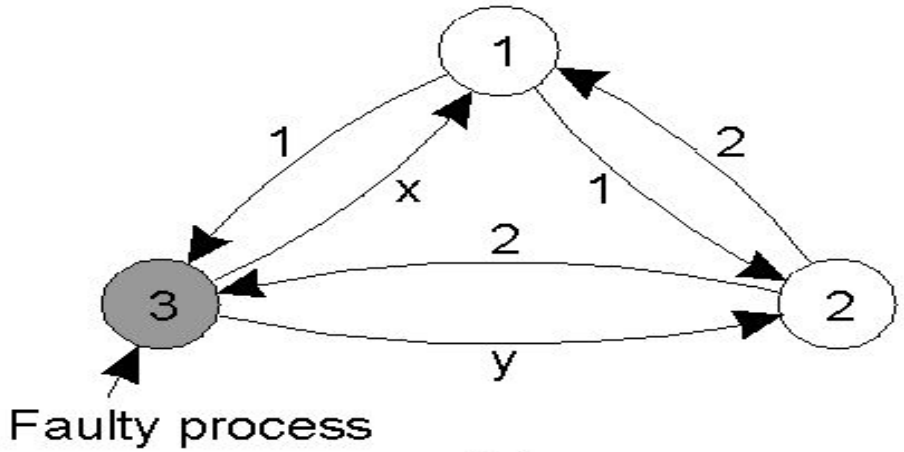
The Byzantine generals problem for 3 loyal generals and1 traitor.

a) The generals announce their troop strengths (in units of 1 thousand soldiers).

b) The vectors that each general assembles based on (a)

c) The vectors that each general receives in step 3.

Go forward one more step



(a)

More than two-thirds □ agreement

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)

The same as in previous slide, except now with 2 loyal generals and one traitor.

Lamport proved that in a system with m faulty processes, agreement can be achieved only if 2m+1 correctly functioning processes are present, for a total of 3m+1.

Reliable client-server communication

What about reliable point-to-point transport protocols ?

- TCP masks omission failures
  - … by using ACKs & retransmissions
- … but it does not mask crash failures !
  - E.g.: When a connection is broken, the client is only notified via an exception

RPC Semantics in the Presence of Failures

Five different classes of failures that can occur in RPC systems:

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

Server Crashes (1)

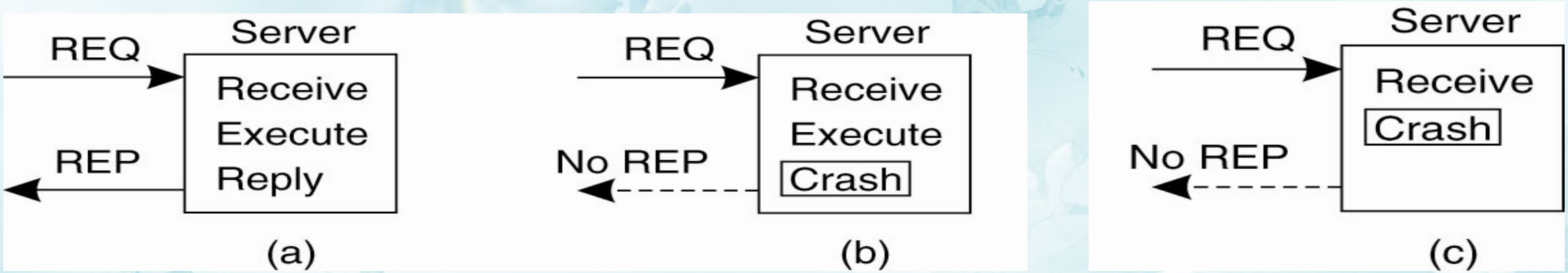


Figure 8-7. A server in client-server communication.
(a) The normal case.
(b) Crash after execution.
(c) Crash before execution.

Server Crashes (2)

- Three events that can happen at the server:
  - Send the completion message (M),
  - Print the text (P),
  - Crash (C).

Server Crashes (3)

- These 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.

Server Crashes (4)

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

- **Figure**. Different combinations of client and server strategies in the presence of server crashes.

Basic Reliable-Multicasting Schemes

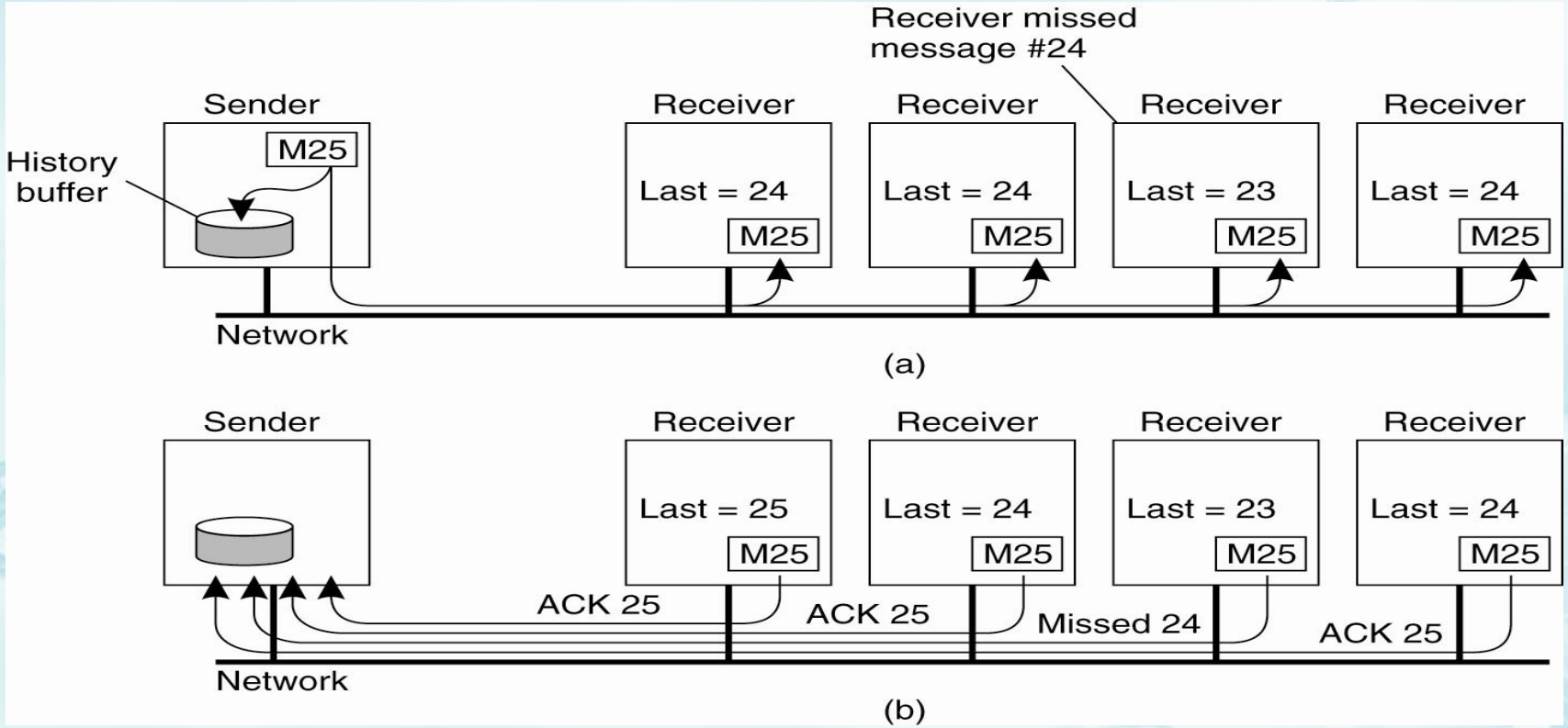


Figure :A simple solution to reliable multicasting when all receivers are known and are assumed not to fail. (a) Message transmission. (b) Reporting feedback.

Nonhierarchical Feedback Control

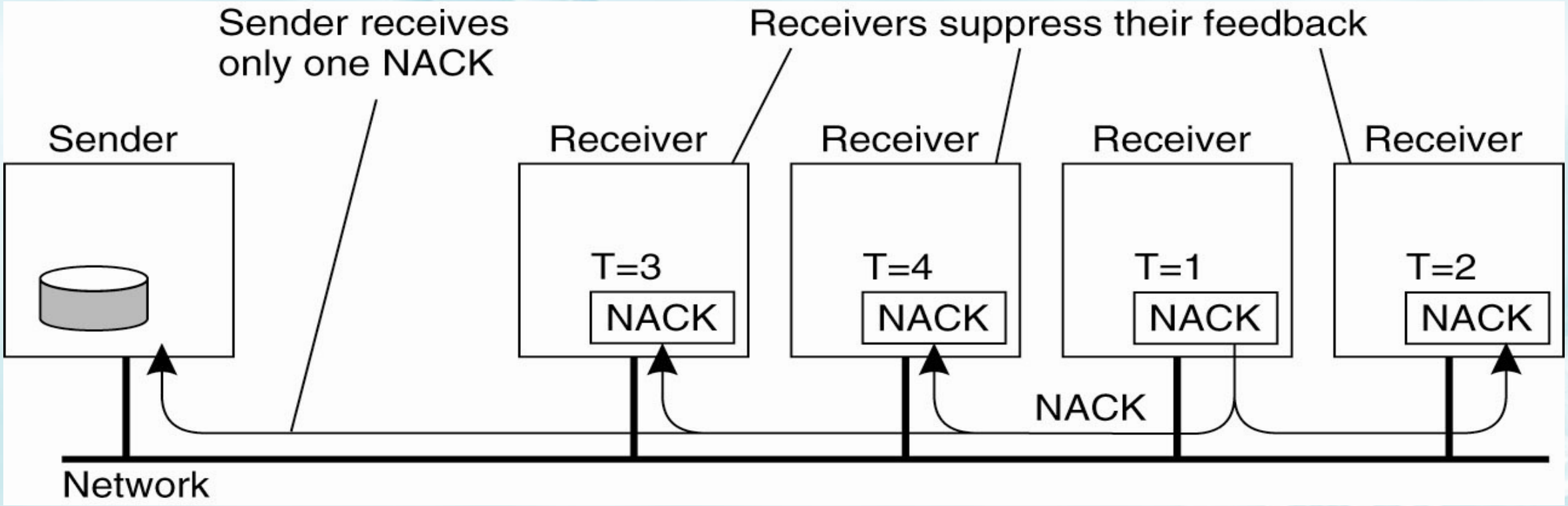


Figure : Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.

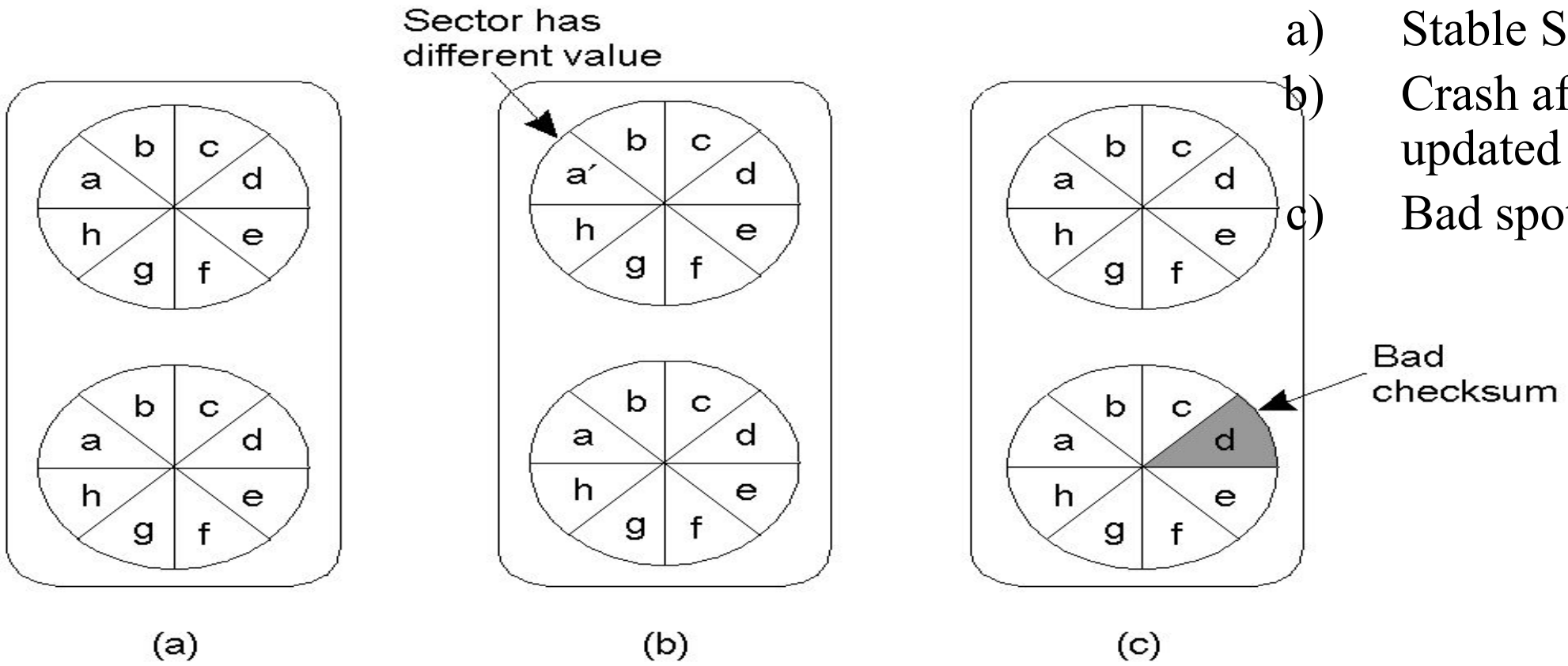
Recovery

Introduction

- Goal: replace an erroneous/wrong state with an error-free state
- Backward recovery: to restore such a recorded state when things go wrong

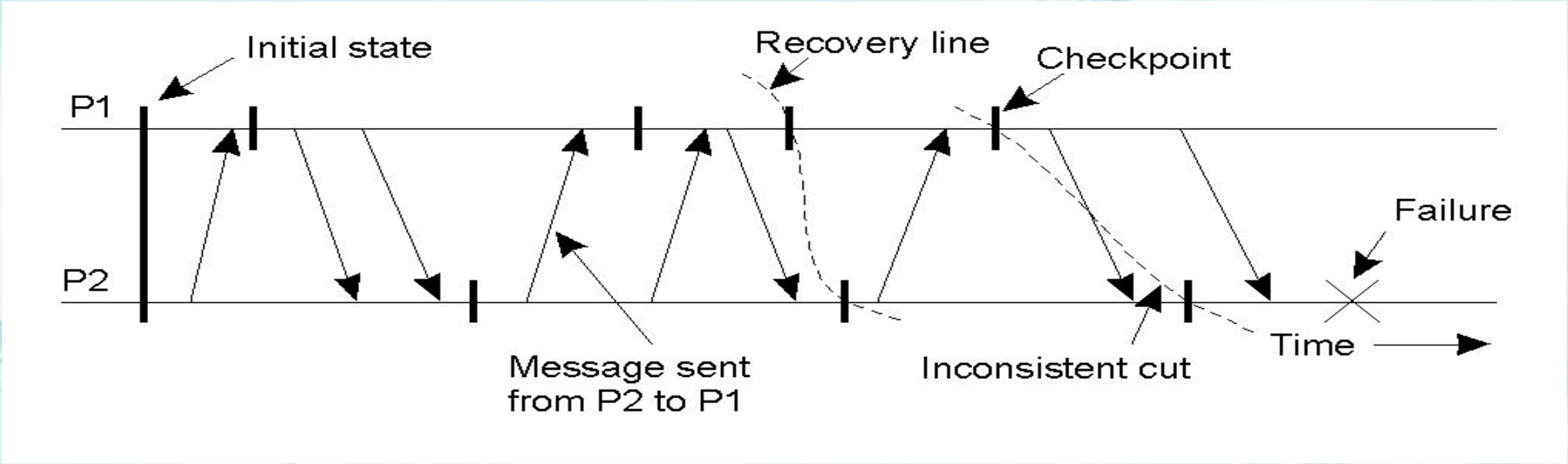
  Combining checkpoints and message logging
- Forward recovery: an attempt is made to bring the system in a correct new state from which it can continue to execute

Stable Storage



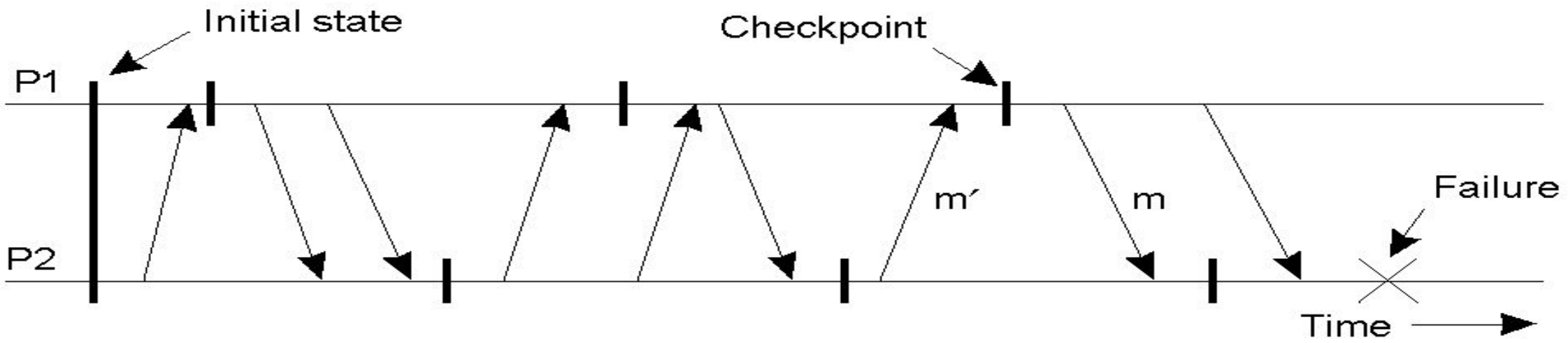
Stable storage is well suited to applications that require a high degree of fault tolerance

Checkpointing



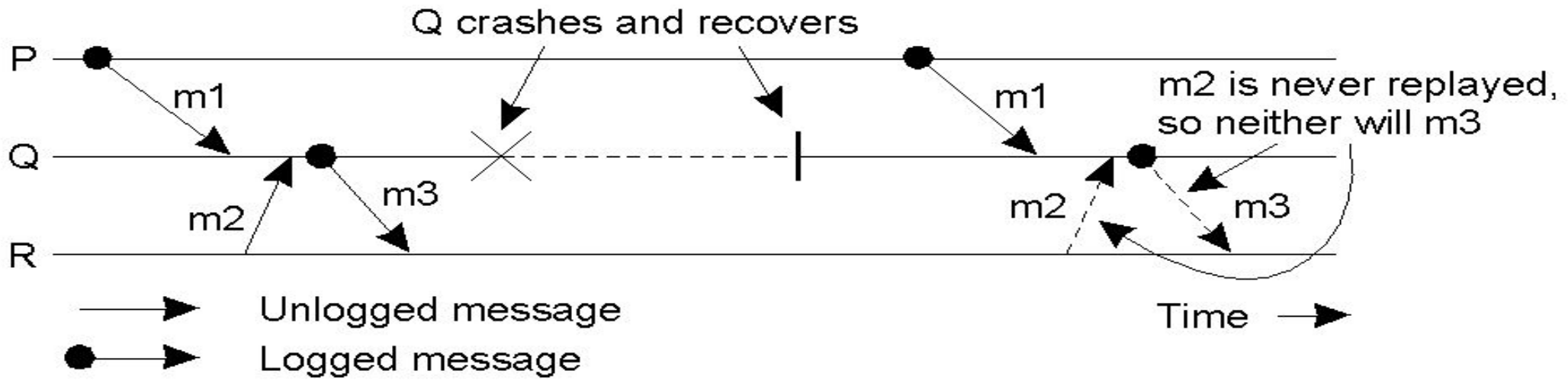
A recovery line: the most recent distributed ***snapshot***

Independent Checkpointing



The domino effect.

Message Logging



Incorrect replay of messages after recovery, leading to an orphan process.

Summarization (I)

- Fault tolerance is defined as the characteristic by which a system can mask the occurrence and recovery from failures
- There exist several types of failures : Crash failure, Omission failure, Timing failure, Response failure, Arbitrary/Byzantine failure
- Redundancy is the key technique needed to achieve fault tolerance
- Reliable group communication is suitable for small groups

Summarization (II)

- Atomic multicasting can be precisely formulated in terms of a virtual synchronous execution model
- Group membership change agreement on the same list of members using commit protocol
- Recovery in fault-tolerant systems is invariably achieved by checkpointing with message logging

References

- Andrew S. Tanenbaum and Maarten Van Steen, “Distributed Systems: Principles and Paradigms”, 2nd edition, Pearson Education.

- Pradeep K. Sinha , “Distributed Operating System” PHI Publication 20008.