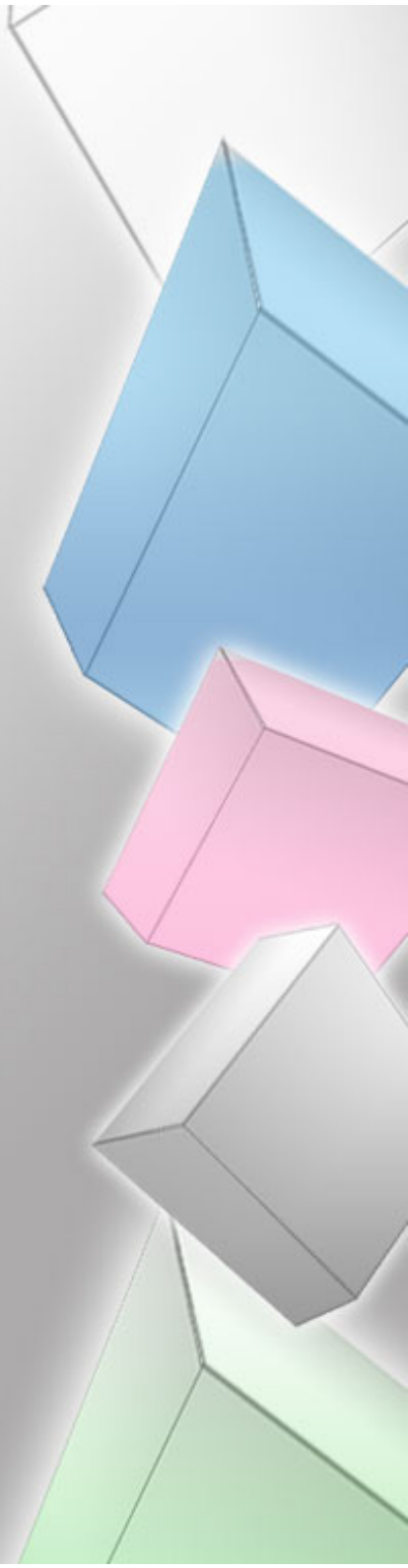
Advanced Topics in Operating Systems

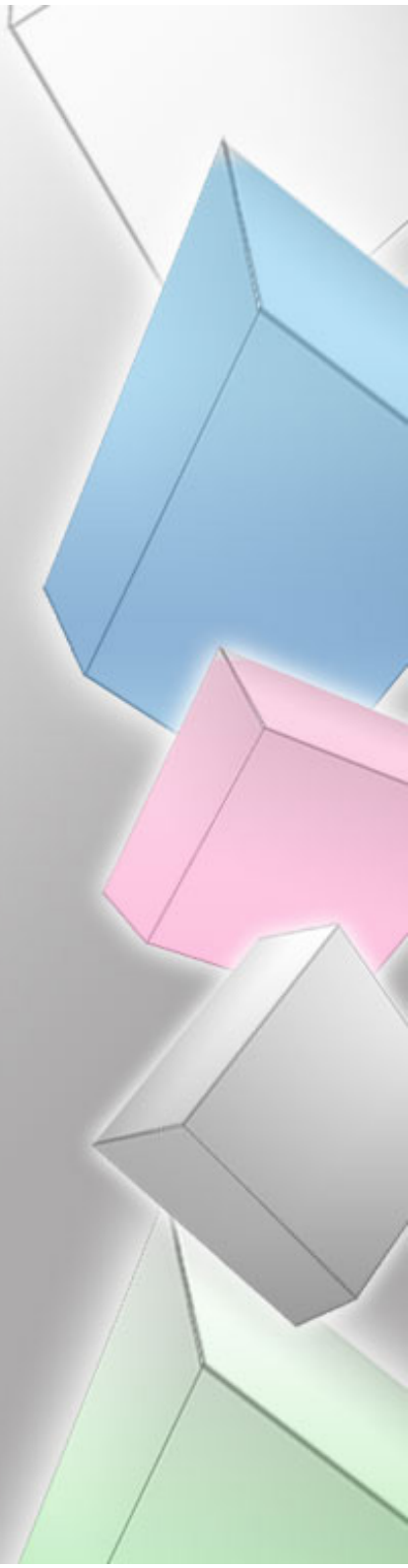
MSc in Computer Science
UNYT-UoG

Dr. Marenglen Biba
8-9-10 January 2010



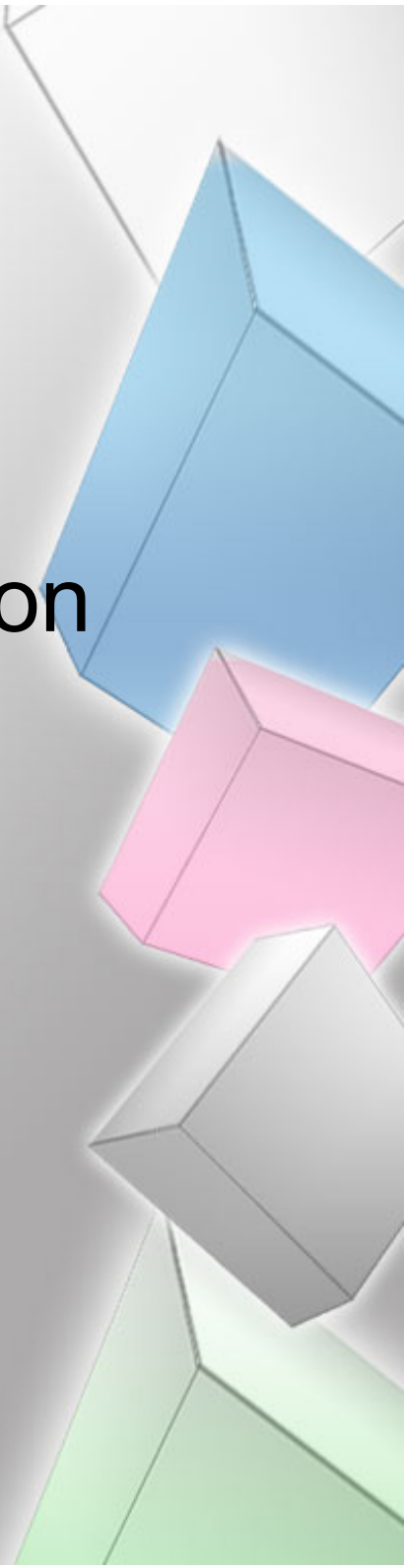
Lesson 8

01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems



Fault tolerance

- Concepts
- Process Resilience
- Reliable Client-Server Communication
- Reliable Group Communication
- Recovery



Dependable Systems

Basics

- A component provides services to clients. To provide services, the component may require the services from other components => a component may depend on some other component.
- Requirements for Dependability
  - **Availability** Readiness for usage
  - **Reliability** Continuity of service delivery
  - **Safety** Very low probability of catastrophes
  - **Maintainability** How easy can a failed system be repaired

Definitions

- A system is said to **fail** when it cannot meet its promises.
  - In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be (completely) provided.
- An **error** is a part of a system's state that may lead to a failure.
  - For example, when transmitting packets across a network, it is to be expected that some packets have been damaged when they arrive at the receiver.
- The cause of an error is called a **fault**. Clearly, finding out what caused an error is important.
  - For example, a wrong or bad transmission medium may easily cause packets to be damaged.

Fault Tolerance

- Building dependable systems closely relates to controlling faults. A distinction can be made between preventing, removing, and forecasting faults.
- For our purposes, the most important issue is fault tolerance, meaning that a system can provide its services even in the presence of faults. In other words, the system can tolerate faults and continue to operate normally.
- Faults:
  - Transient: occur once and then disappear
  - Intermittent: appears, disappears, and so on
  - Permanent: exists until the faulty component is replaced

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 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

Figure 8-1. Different types of failures.

Crash Failures

Problem

- Clients cannot distinguish between a crashed component and one that is just a bit slow

Consider a server from which a client is expecting output

- Is the server perhaps exhibiting timing or omission failures?
- Is the channel between client and server faulty?

Assumptions we can make

- Fail-silent: The component exhibits omission or crash failures; clients cannot tell what went wrong
- Fail-stop: The component exhibits crash failures, but its failure can be detected (either through announcement or timeouts)
- Fail-safe: The component exhibits arbitrary, but benign failures (they can't do any harm). Clients know!

Failure Masking by Redundancy

- If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes.
- The key technique for masking faults is to use redundancy.
- Three kinds are possible:
  - Information redundancy: Example => Hamming code
  - Time redundancy: perform operation again
  - Physical redundancy: extra equipment of processes

Physical Redundancy

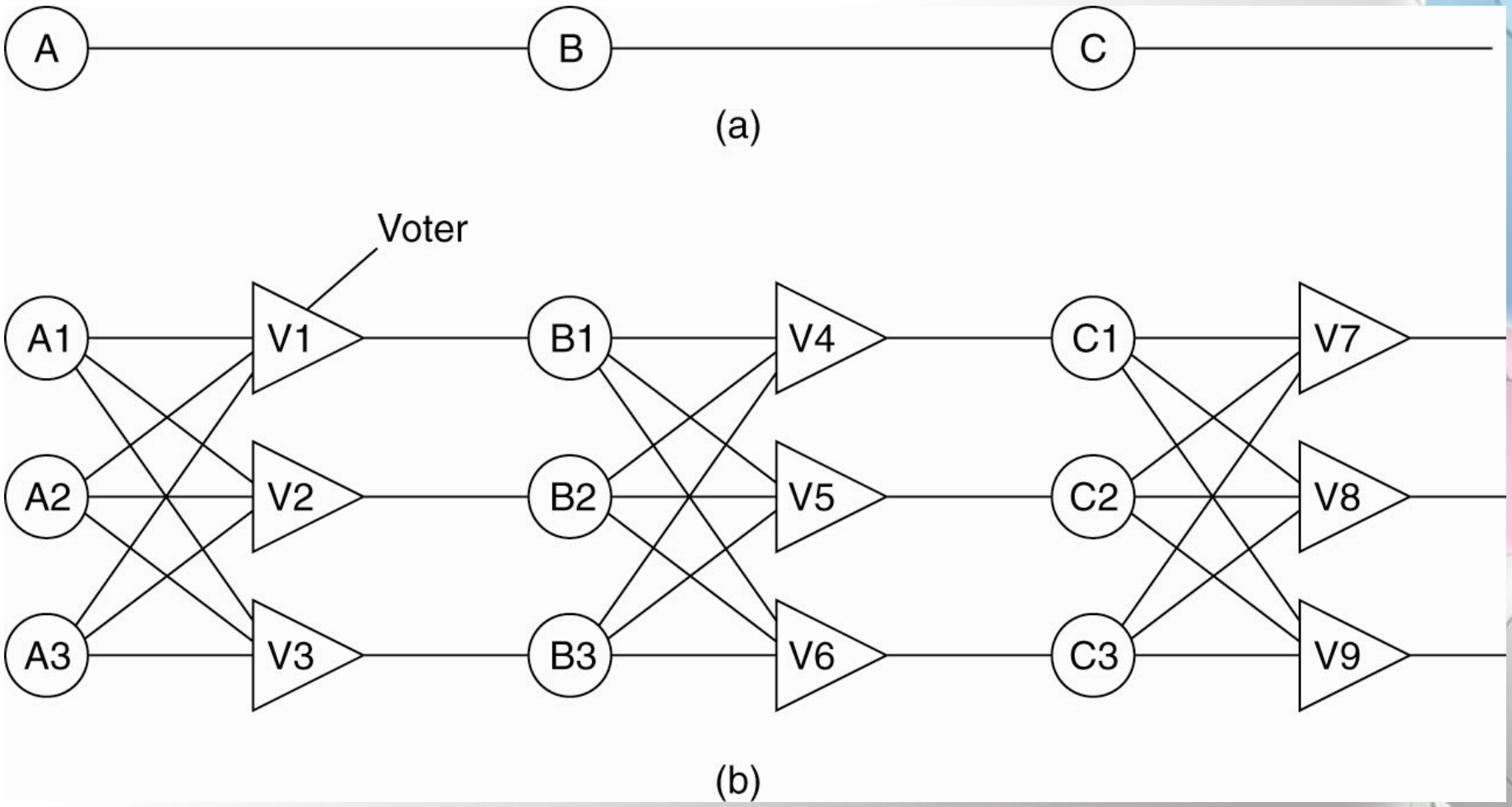
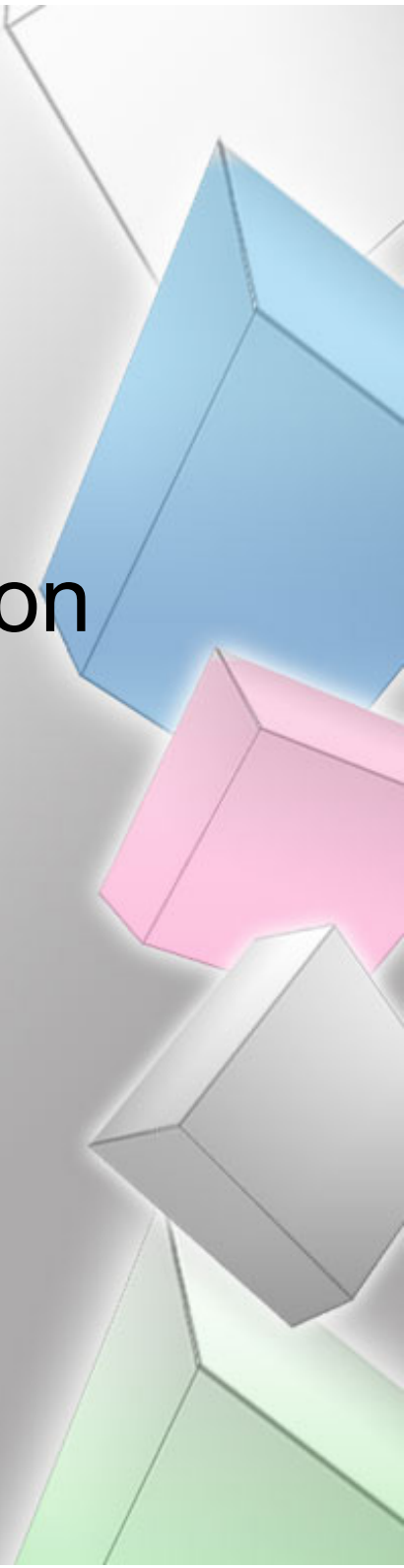


Figure 8-2. Triple modular redundancy.

Fault tolerance

- Concepts
- Process Resilience
- Reliable Client-Server Communication
- Reliable Group Communication
- Recovery



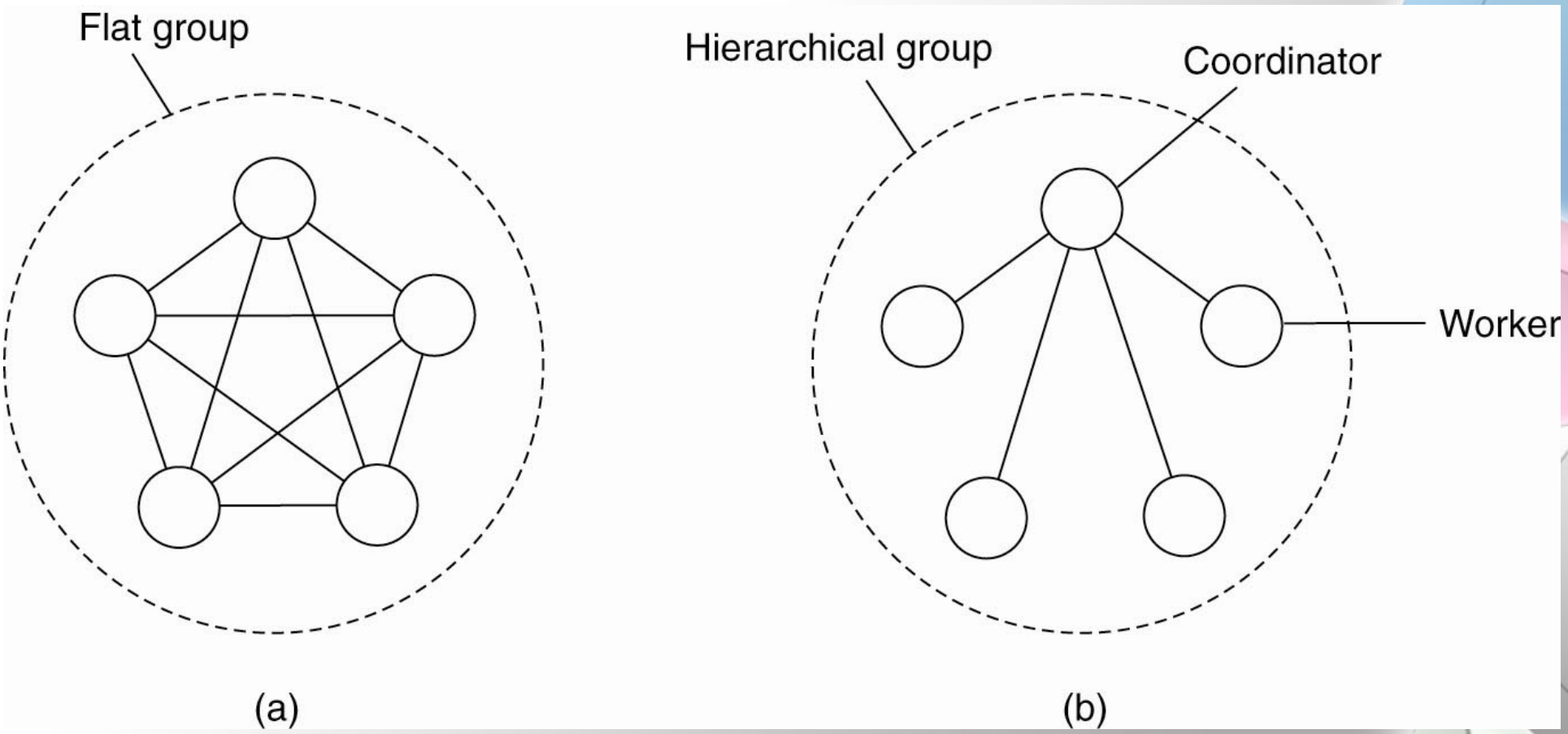
Process Resilience

Basic issue

Protect yourself against faulty processes by replicating processes into a group.

- Flat groups: Good for fault tolerance as information exchange immediately occurs with all group members
  - However, may impose more overhead as control is completely distributed (hard to implement).
- Hierarchical groups: All communication through a single coordinator => not really fault tolerant and scalable, but relatively easy to implement.

Flat Groups versus Hierarchical Groups



- Figure 8-3. (a) Communication in a flat group. (b) Communication in a simple hierarchical group.

Groups and failure masking

***K-fault* tolerant group**

- When a group can mask any *k* concurrent member failures (*k* is called degree of fault tolerance).

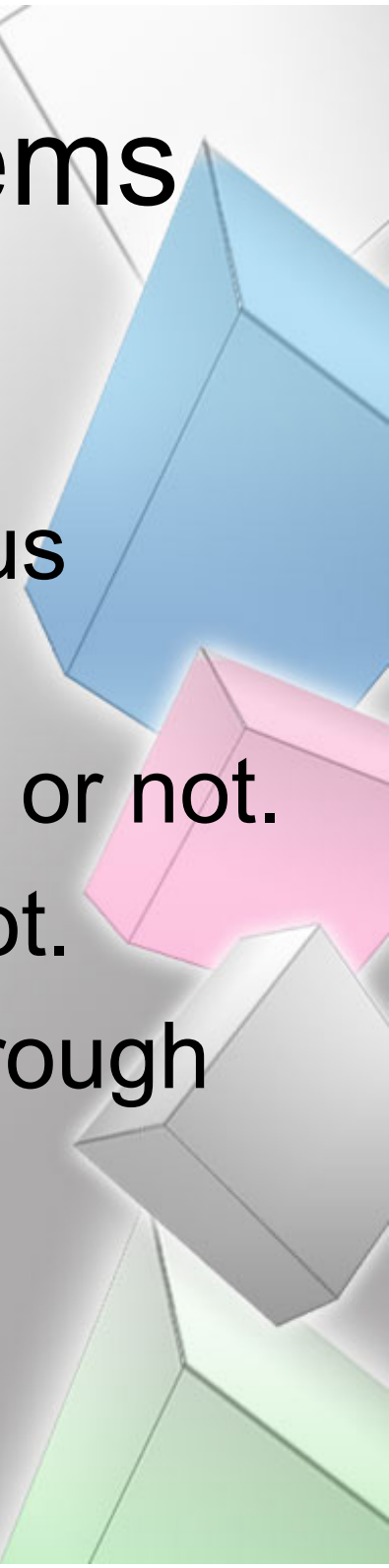
How large does a k-fault tolerant group need to be?

- Assume crash semantics => a total of *k+1* members are needed to survive *k* member failures.
- Assume arbitrary failure semantics, and group output defined by voting => a total of *2k+1* members are needed to survive *k* member failures.

Agreement in Faulty Systems

Possible cases:

1. Synchronous versus asynchronous systems.
2. Communication delay is bounded or not.
3. Message delivery is ordered or not.
4. Message transmission is done through unicasting or multicasting.



Agreement in Faulty Systems

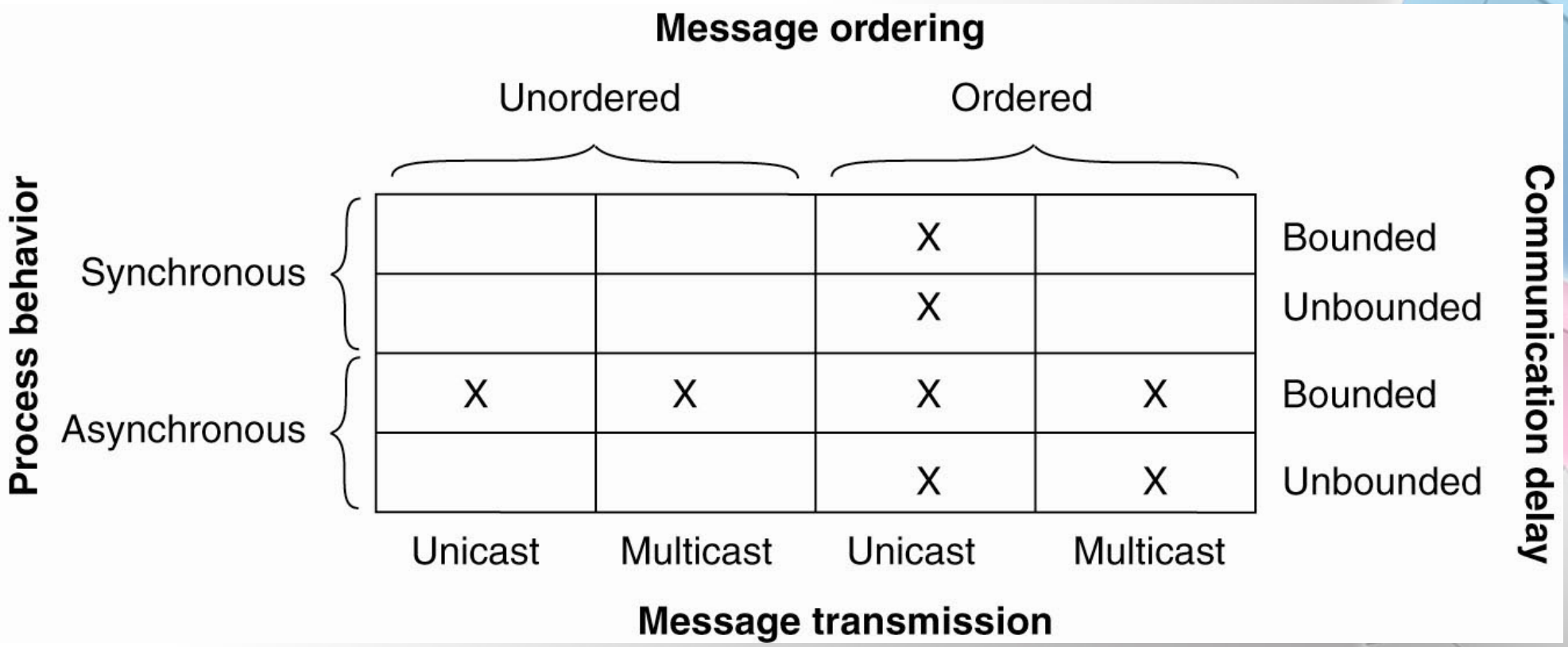
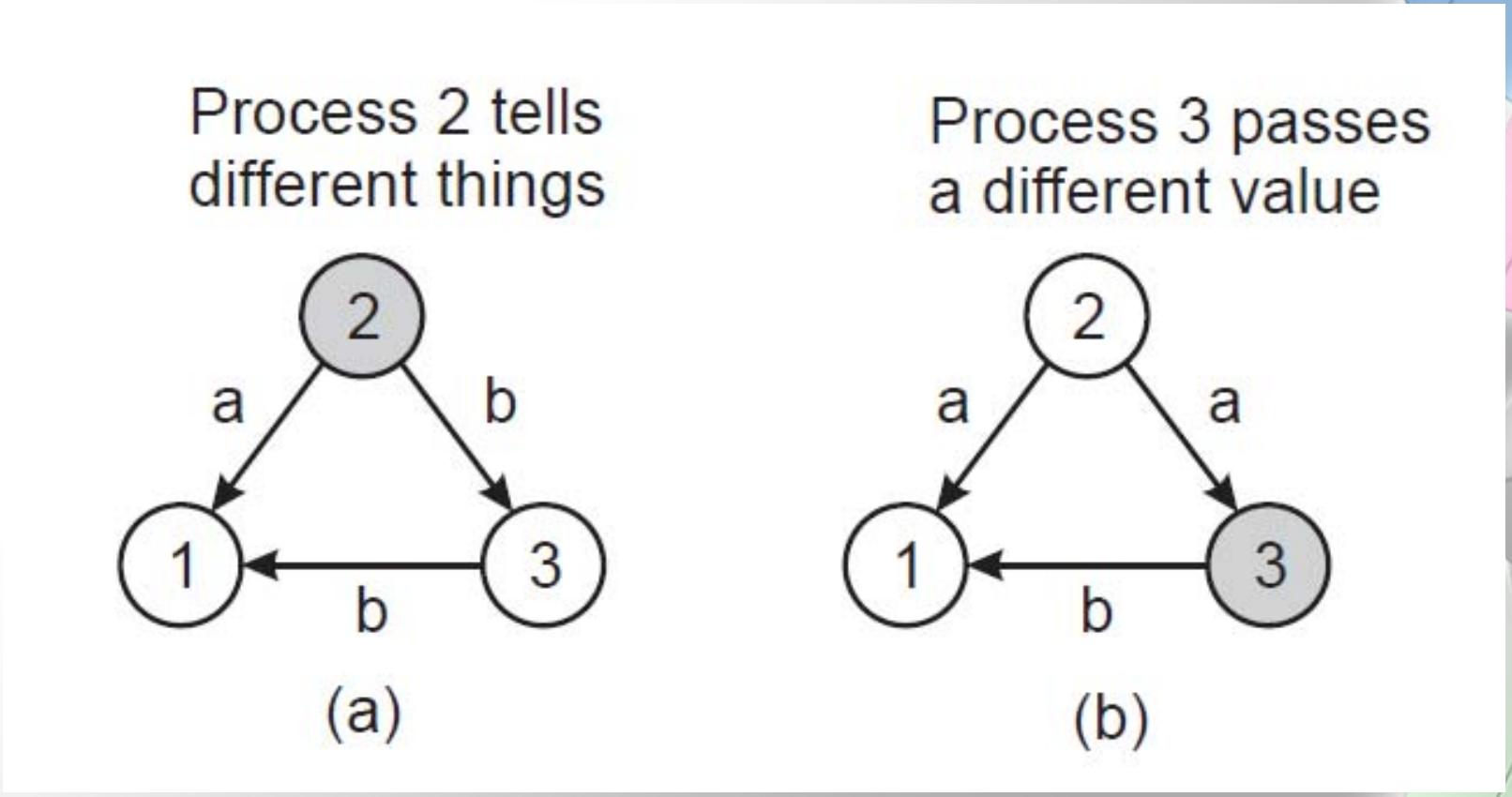


Figure 8-4. Circumstances under which distributed agreement can be reached.

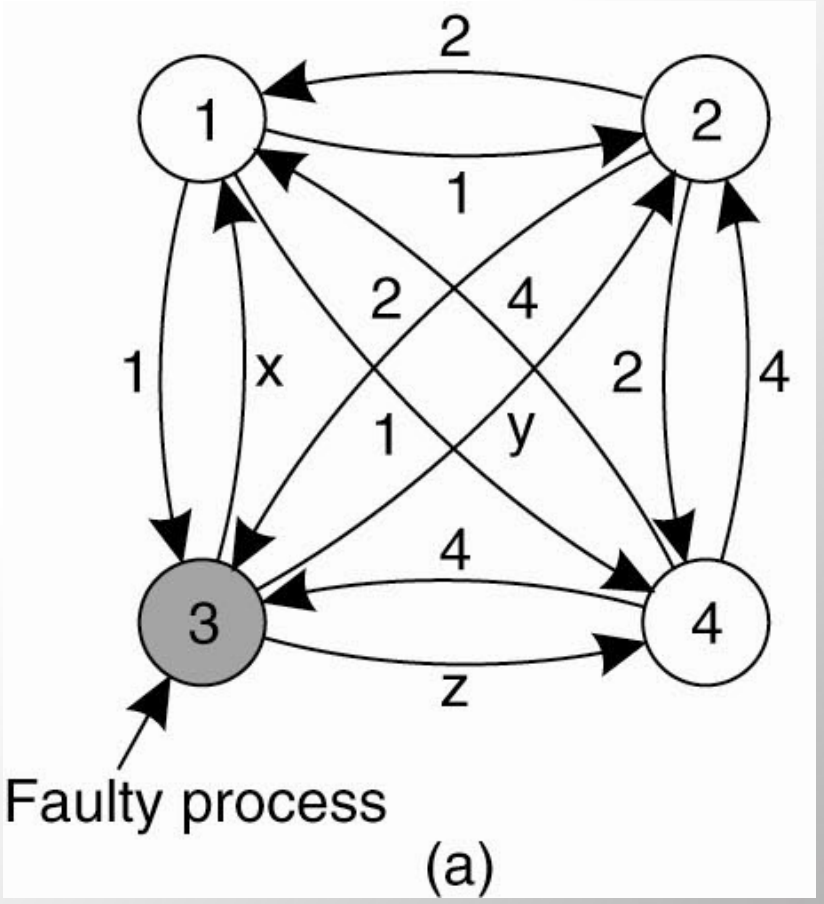
Byzantine Agreement Problem (Lamport, 1982)

Scenario

- Group members are not identical, i.e., we have a distributed computation => Nonfaulty group members should reach agreement on the same value.



Agreement in Faulty Systems



- Figure 8-5. The Byzantine agreement problem for three nonfaulty and one faulty process. (a) Each process sends their value to the others.

Agreement in Faulty Systems

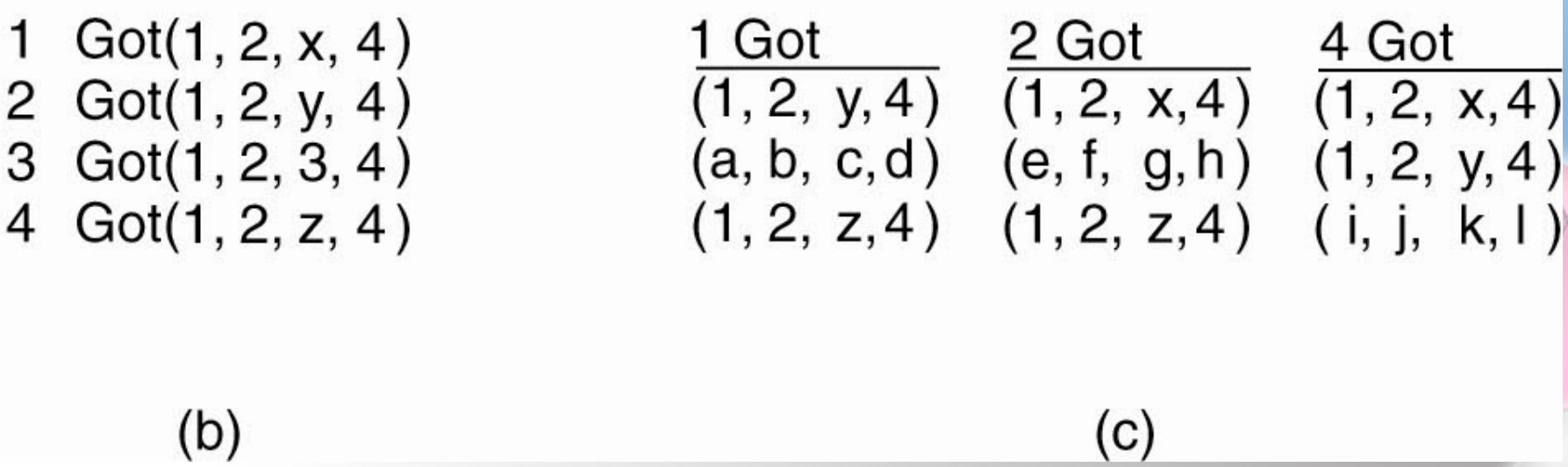
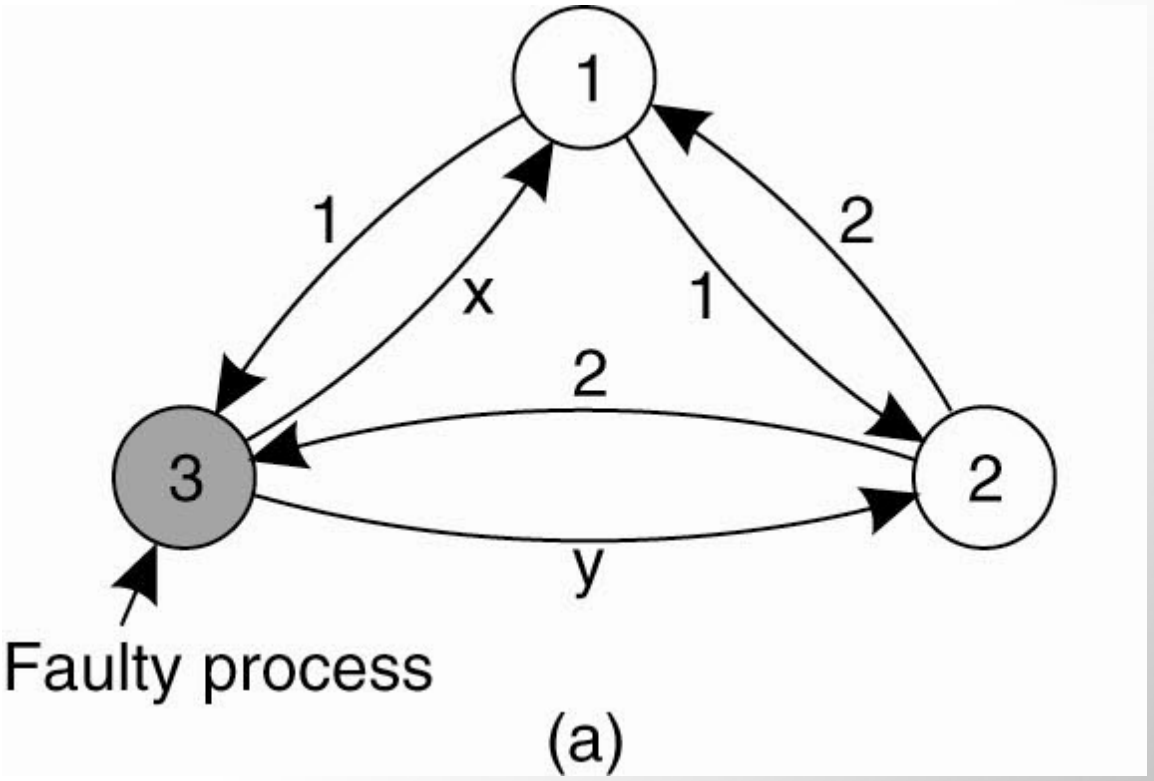


Figure 8-5. (b) The vectors that each process assembles based on (a). (c) The vectors that each process receives from other processes in step 3.

Processes 1, 2 and 4 come to agreement on the values for V_1, V_2, V_4.

Failure of 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)

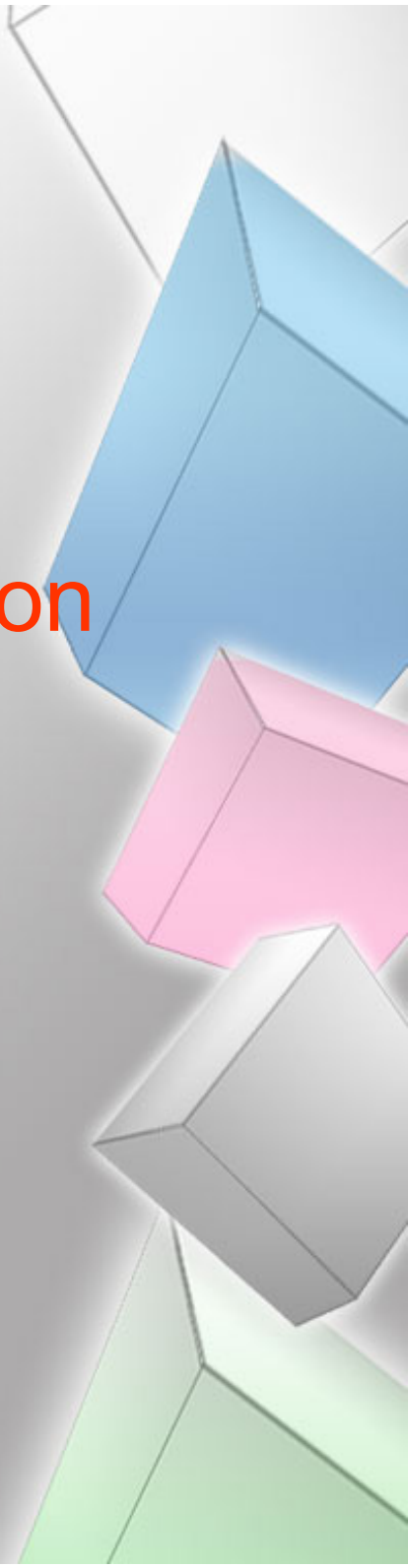
Figure 8-6. Failure of producing agreement.

Byzantine Agreement Requirements

- In their paper, Lamport et al. (1982) proved that in a system with *k* faulty processes, agreement can be achieved only if $2k + 1$ correctly functioning processes are present, for a total of $3k + 1$.
- Put in slightly different terms, agreement is possible only if *more* than two-thirds of the processes are working properly.

Fault tolerance

- Concepts
- Process Resilience
- Reliable Client-Server Communication
- Reliable Group Communication
- Recovery



RPC Semantics in the Presence of Failures

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

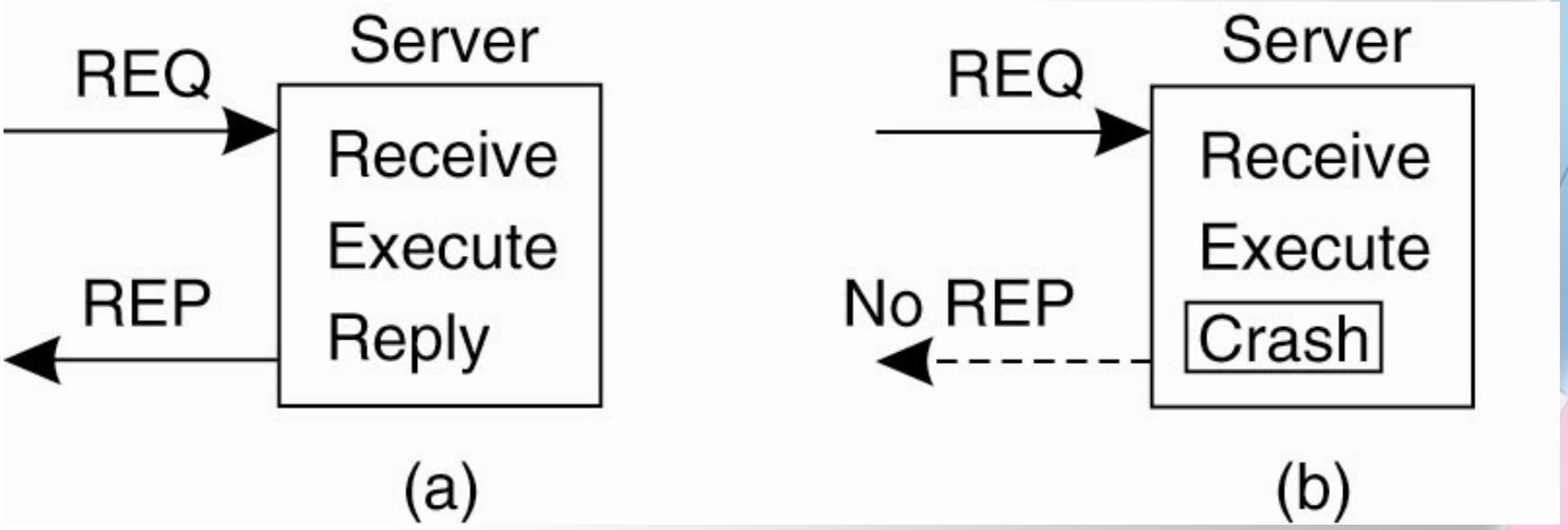
1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

RPC communication: Solutions

1: Raise exception: just report back to client (not really a solution, no transparency!)

2: Just resend message

Server Crashes



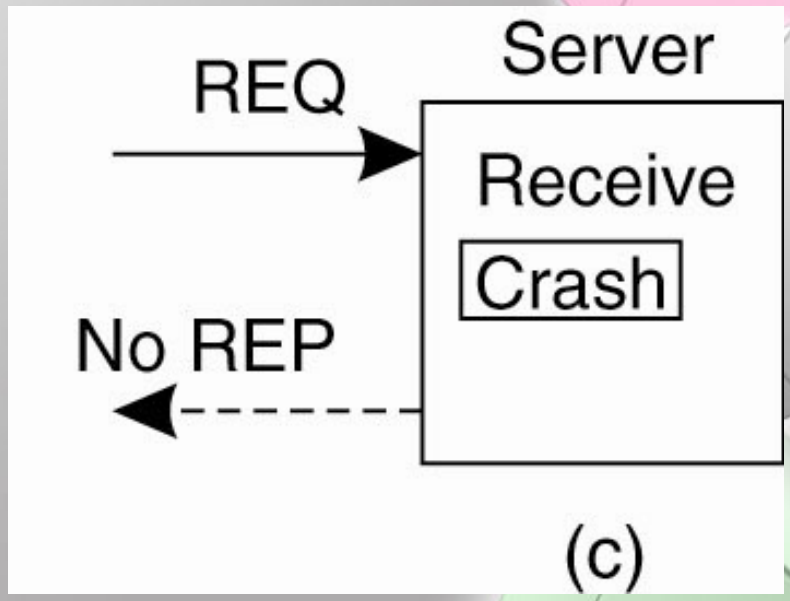
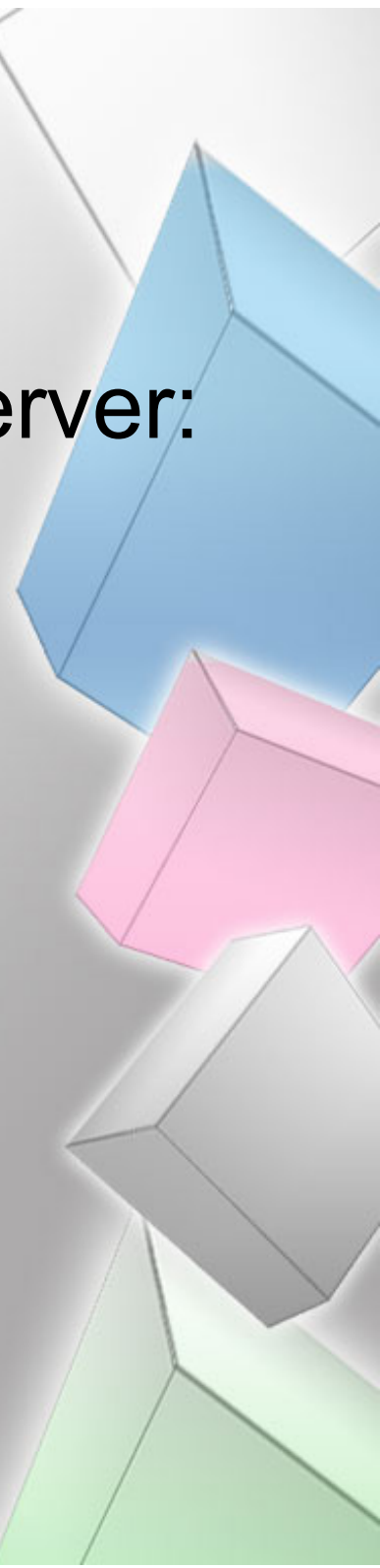


Figure 8-7. A server in client-server communication.
(a) The normal case.
(b) Crash after execution.
(c) Crash before execution.

Server Crashes

Three events that can happen at the server:

- Send the completion message (M),
- Print the text (P),
- Crash (C).



Server Crashes

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: solutions

Problem

We need to decide on what we expect from the server

- At-least-once-semantics: The server guarantees it will carry out an operation at least once, no matter what.
- At-most-once-semantics: The server guarantees it will carry out an operation at most once.

Server Crashes

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 8-8. Different combinations of client and server strategies in the presence of server crashes.

Server response is lost

RPC communication: Solutions

4. Server response is lost

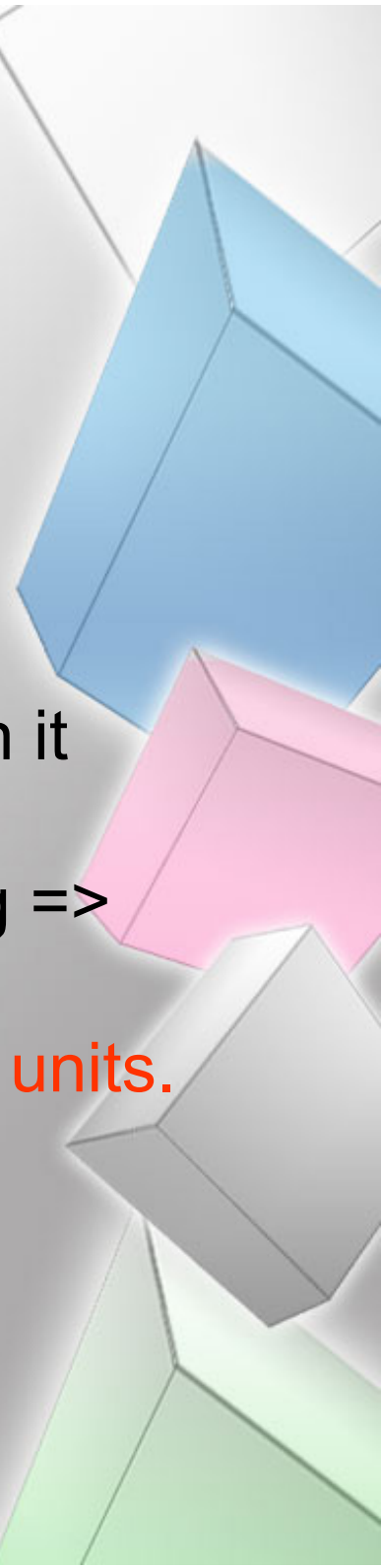
- Detecting lost replies can be hard, because it can also be that the server had crashed.
- You don’t know whether the server has carried out the operation
- Solution: send the request again,
  - we can try to make the operations idempotent: repeatable without any harm done if it happened to be carried out before.

Client Crashes

RPC communication: Solutions

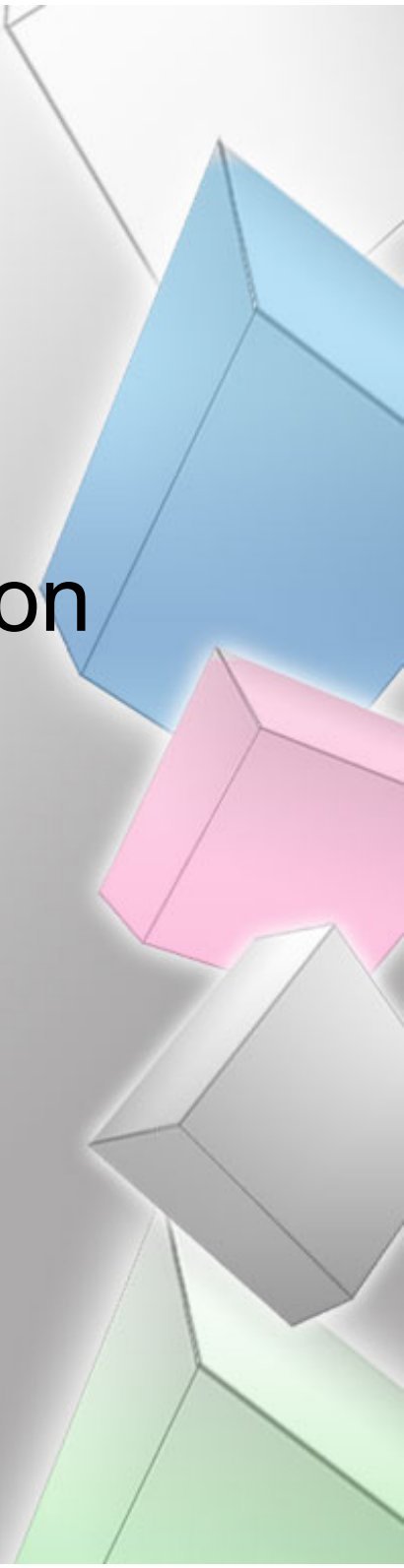
5. Client crashes

- Problem: The server is doing work and holding resources for nothing (called doing an orphan computation).
  - Orphan is killed (or rolled back) by client when it reboots
  - Broadcast new epoch number when rebooting => servers kill orphans
  - Require computations to complete in a T time units. Old ones are simply removed.



Fault tolerance

- Concepts
- Process Resilience
- Reliable Client-Server Communication
- Reliable Group Communication
- Recovery



Reliable multicasting

- **Basic model**

We have a multicast channel *c* with two (possibly overlapping) groups:

  - The sender group *SND(c)* of processes that submit messages to channel *c*
  - The receiver group *RCV(c)* of processes that can receive messages from channel *c*

- Simple reliability: If process $P \in RCV(c)$ at the time message *m* was submitted to *c*, and *P* does not leave *RCV(c)*, *m* should be delivered to *P*
- Atomic multicast: How can we ensure that a message *m* submitted to channel *c* is delivered to process $P \in RCV(c)$ only if *m* is delivered to all members of *RCV(c)*?

Basic Reliable-Multicasting Schemes

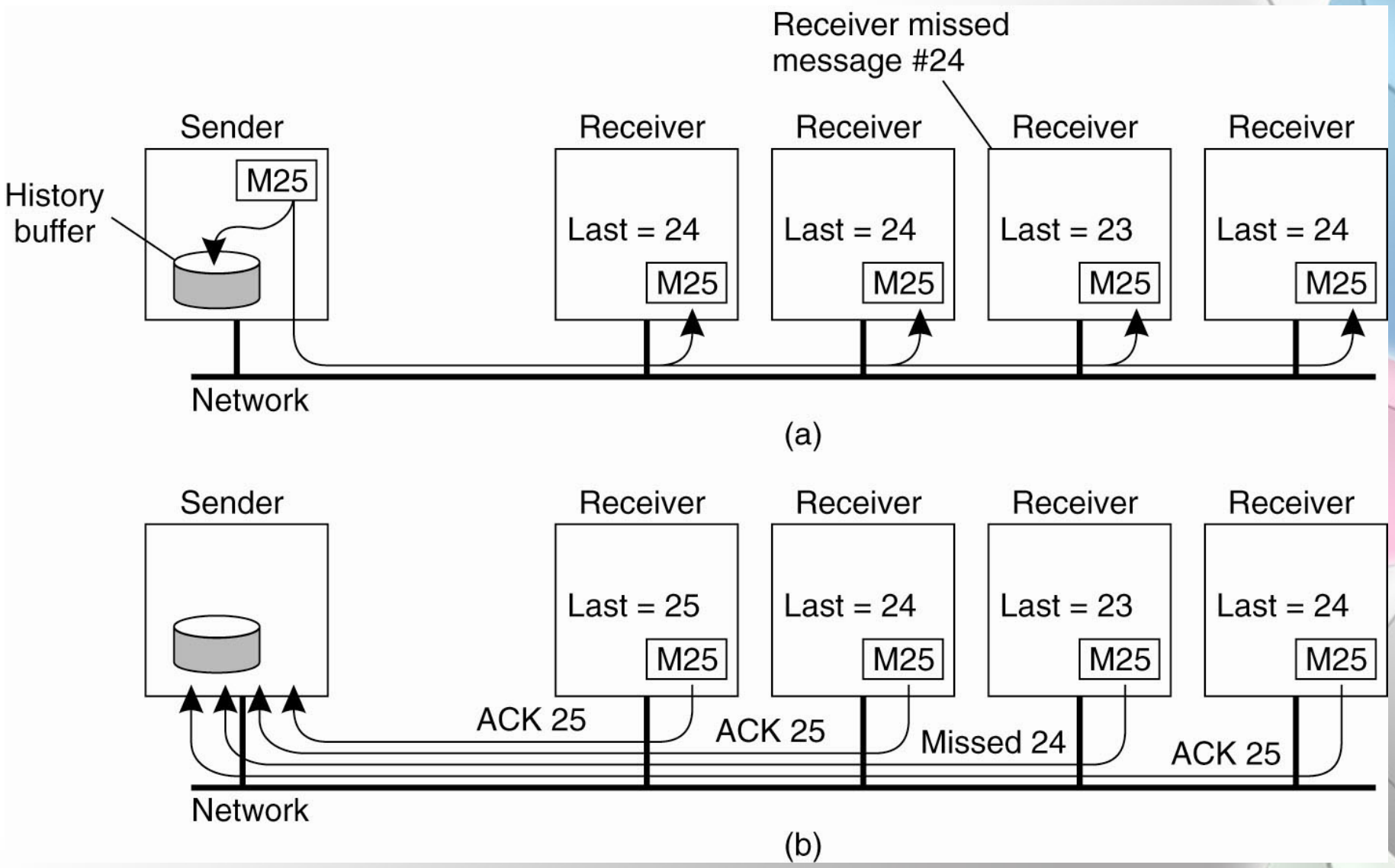


Figure 8-9.Rreceivers are known and assumed not to fail. (a) Message transmission with sequence numbering. (b) Reporting feedback.

Reliable-Multicasting: Feedback suppression

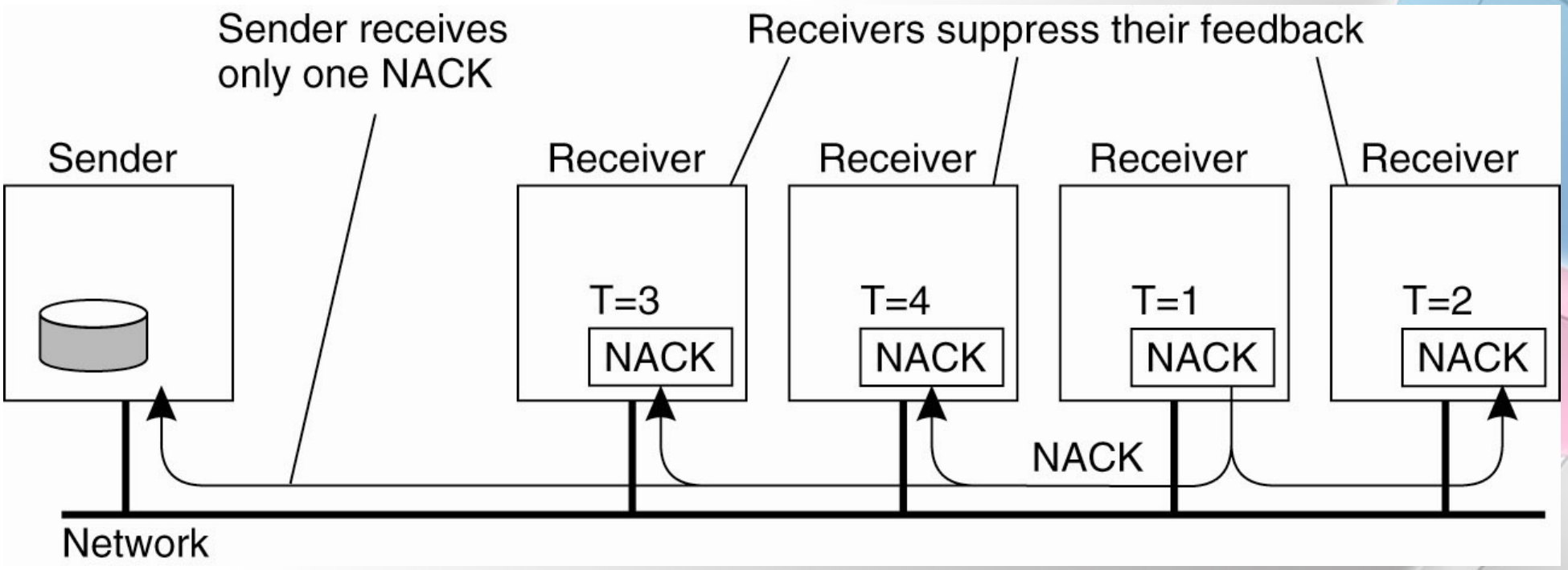


Figure 8-10. Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.

Hierarchical Feedback Control

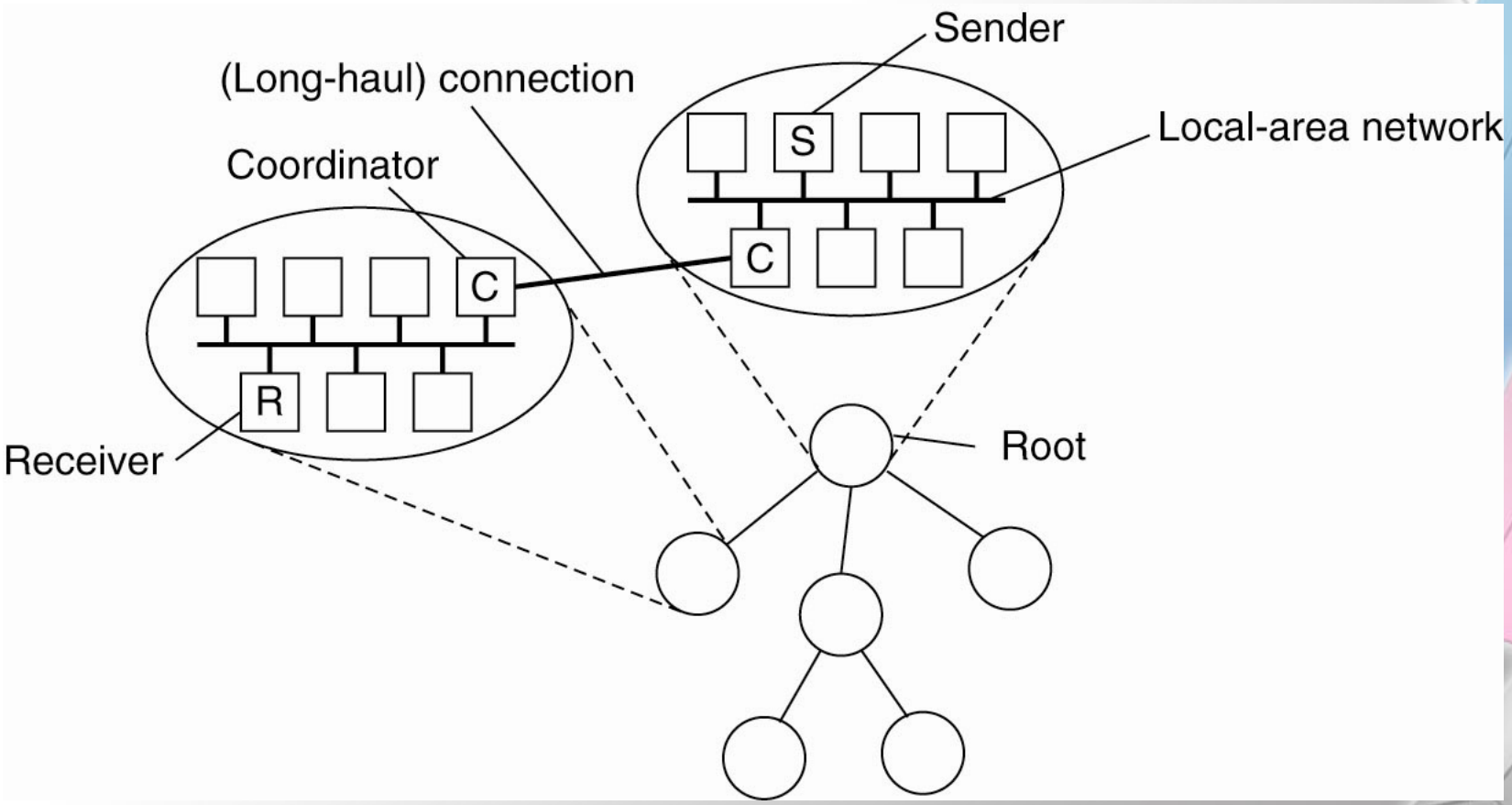


Figure 8-11. Hierarchical reliable multicasting: Each local coordinator forwards the message to its children and later handles retransmission requests.

Atomic Multicast

Idea

Formulate reliable multicasting in the presence of process failures in terms of process groups and changes to group membership.

- What is often needed in a distributed system is the guarantee that a message is delivered to either all processes or to none at all.
- In addition, it is generally also required that all messages are delivered in the same order to all processes.
- This is also known as the atomic multicast problem.

Virtual Synchrony (2)

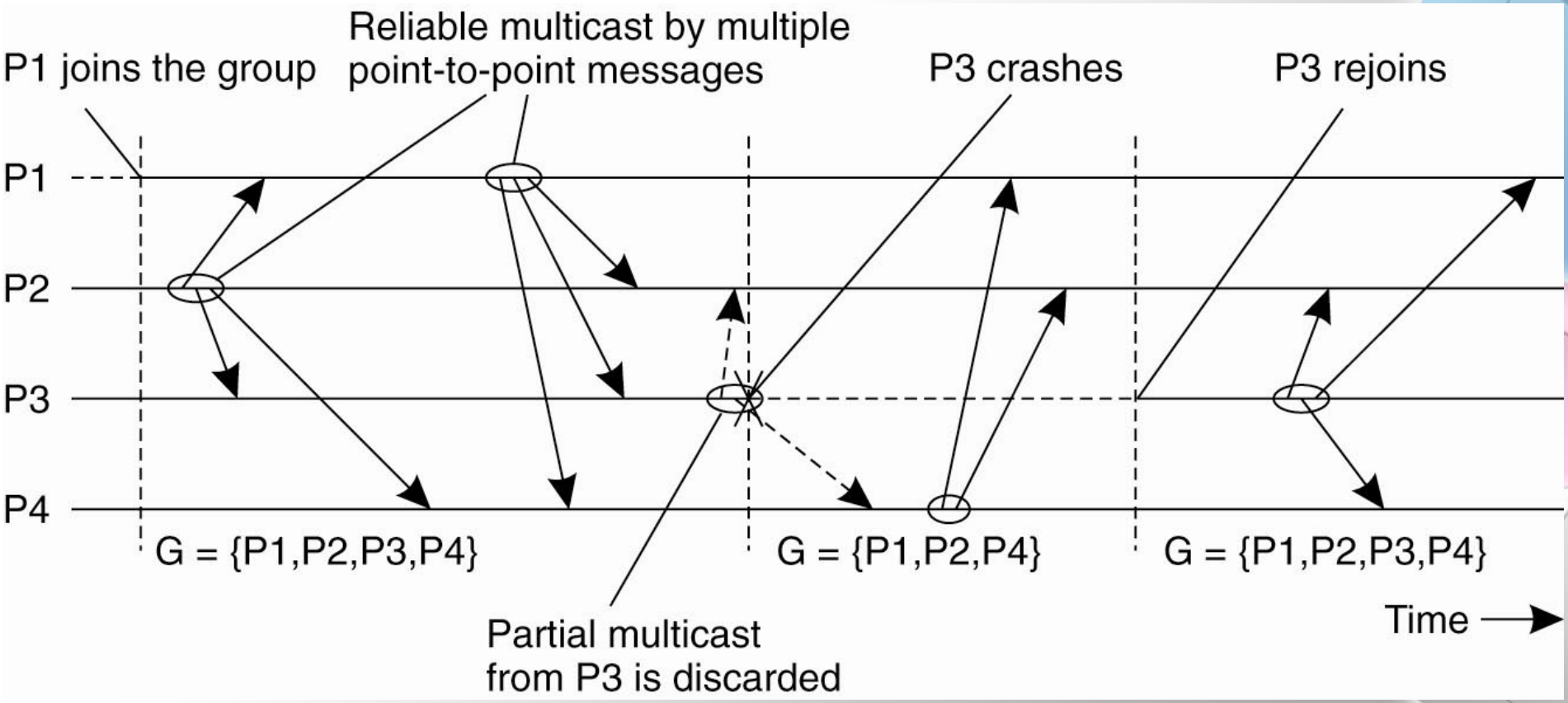
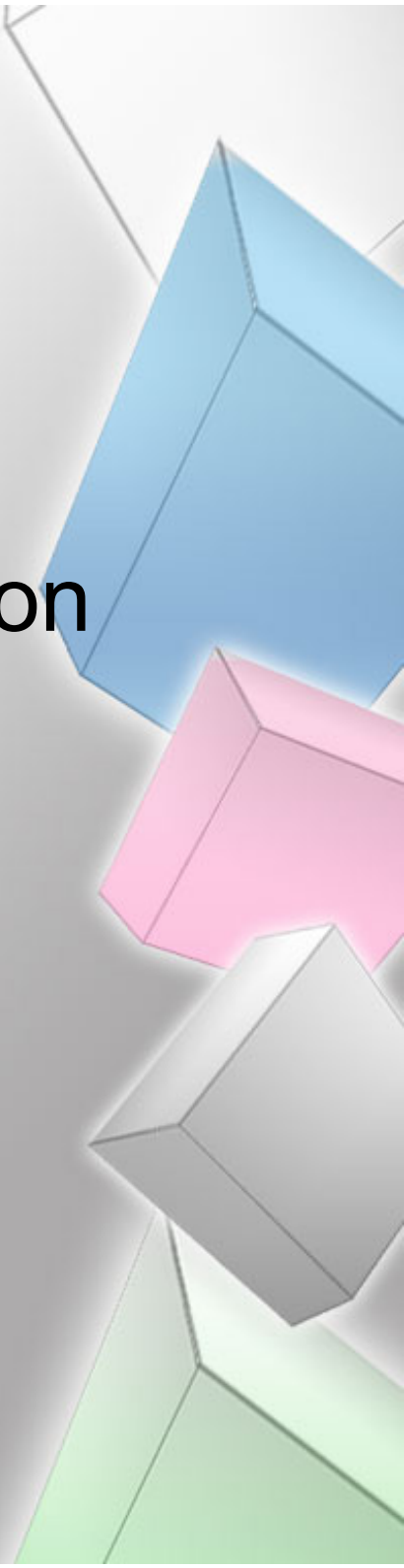


Figure 8-13. The principle of virtual synchronous multicast.

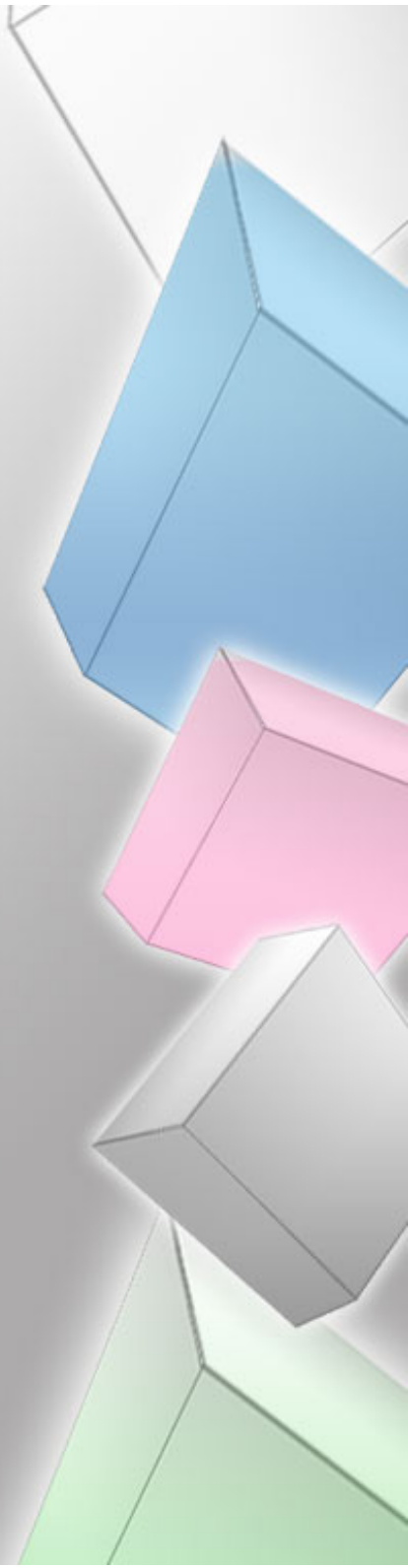
Fault tolerance

- Concepts
- Process Resilience
- Reliable Client-Server Communication
- Reliable Group Communication
- Recovery



Recovery

- Checkpointing
- Message Logging



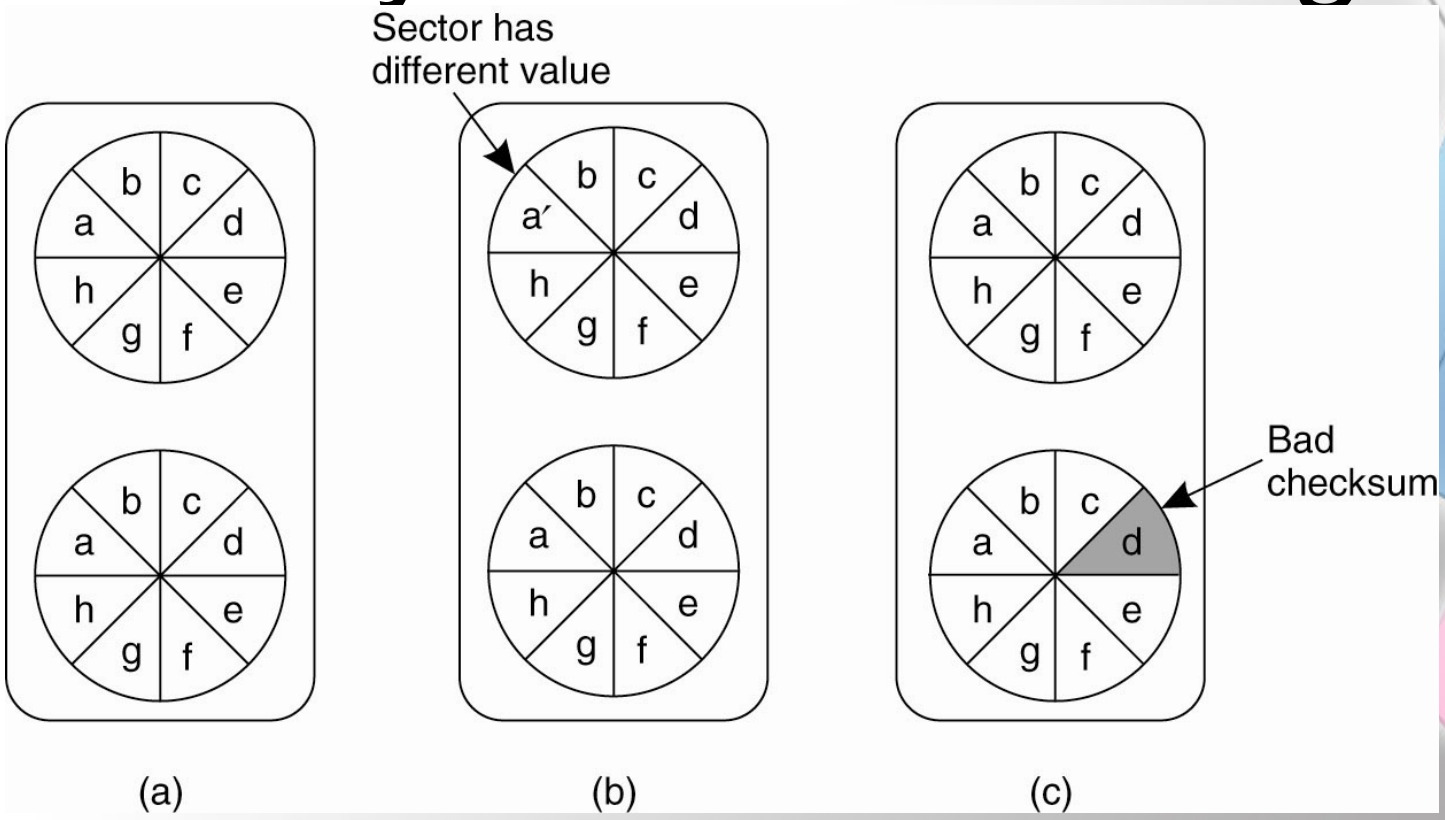
Types of Recovery

- In backward recovery, the main issue is to bring the system from its present erroneous state back into a previously correct state. To do so, it will be necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong.
  - Each time (part of) the system's present state is recorded, a checkpoint is said to be made.
- Another form of error recovery is forward recovery. In this case, when the system has entered an erroneous state, instead of moving back to a previous, checkpointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute.
  - The main problem with forward error recovery mechanisms is that it has to be known in advance which errors may occur.

Erasure Correction

- In this approach, a missing packet is constructed from other, successfully delivered packets.
- For example, in an *(n,k)* block erasure code, a set of *k source packets* is encoded into a set of *n encoded packets,* such that *any* set of *k* encoded packets is enough to reconstruct the original *k* source packets.
  - Typical values are k =16 or k=*32,* and $k<11\leq 2k$.
- If not enough packets have yet been delivered, the sender will have to continue transmitting packets until a previously lost packet can be constructed.
- Erasure correction is a typical example of a forward error recovery approach.

Recovery – Stable Storage



After a crash

If both disks are identical: you're in good shape.

If one is bad, but the other is okay (checksums): choose the good one.

If both seem okay, but are different: choose the main disk.

If both aren't good: you're not in a good shape ☺

Checkpointing

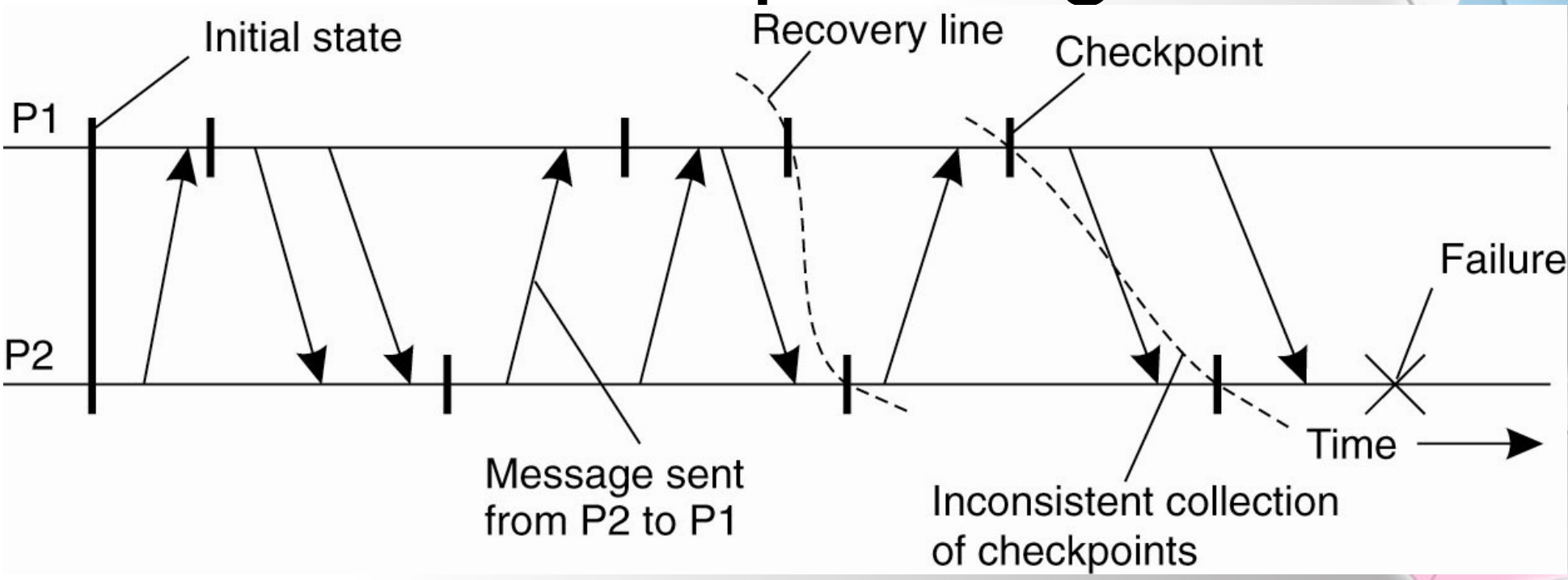


Figure 8-24. A recovery line.

We need to record a consistent global state, also called a distributed snapshot. In a distributed snapshot, if a process *P* has recorded the receipt of a message, then there should also be a process *Q* that has recorded the sending of that message. After all, it must have come from somewhere.

Independent Checkpointing

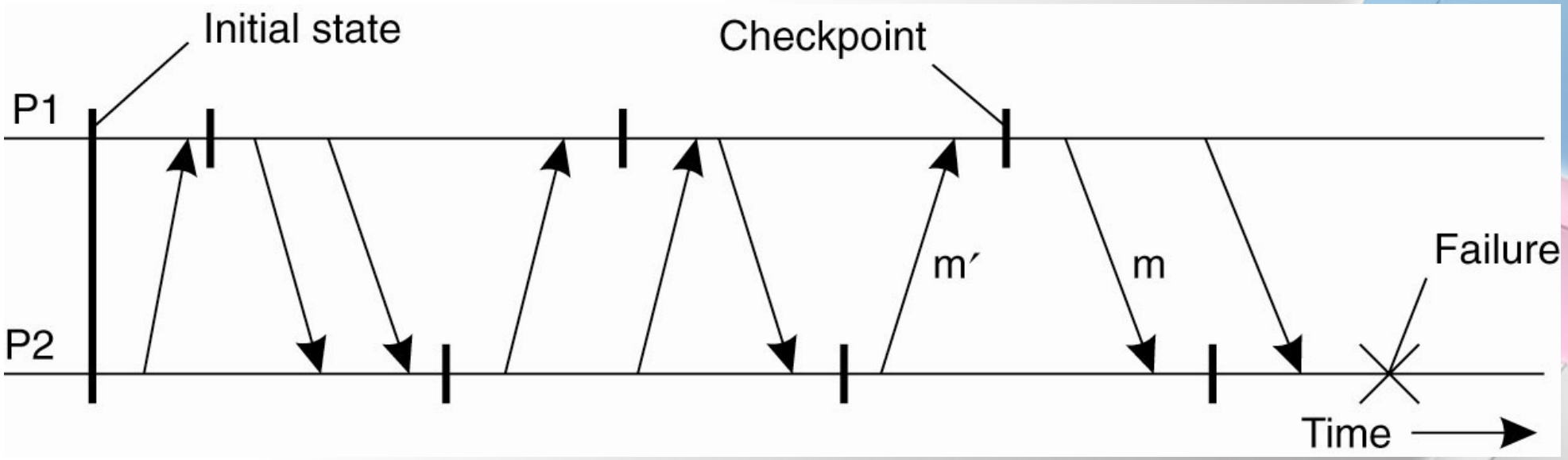


Figure 8-25. The domino effect.

Coordinated Checkpointing

- In coordinated checkpointing all processes synchronize to jointly write their state to local stable storage.
  - The main advantage of coordinated checkpointing is that the saved state is automatically globally consistent, so that cascaded rollbacks leading to the domino effect are avoided.
- A solution is to use a two-phase blocking protocol.
  - A coordinator first multicasts a *CHECKPOINT _REQUEST* message to all processes.
  - When a process receives such a message, it takes a local checkpoint, queues any subsequent message handed to it by the application it is executing, and acknowledges to the coordinator that it has taken a checkpoint.
- When the coordinator has received an acknowledgment from all processes, it multicasts a *CHECKPOINT_DONE* message to allow the (blocked) processes to continue.

End of Lesson 8

- Readings
  - Distributed Systems, Chapter 8, except 8.4.3 (page 351-355), 8.5, 8.6.3 and 8.6.4.