The University of Texas at Arlington
Department of Computer Science and Engineering
CSE@UTA

CSE 5306
Distributed Systems

Fault Tolerance

Failure in Distributed Systems

- Partial failure
  - happens when one component of a distributed system fails
  - often leaves other components unaffected
- A failure in non-distributed system often leads to the failure of entire system
- Fault tolerance
  - The system can automatically recover from partial failures without seriously affecting the overall performance
  - i.e., the system continues to operate in an acceptable way and tolerate faults while repairs are being made

Basic Concepts

- Being fault tolerant is strongly related to
  - Dependable systems
- Dependability implies the following:
  - Availability
    - A system is ready to be used immediately
  - Reliability
    - A system can run continuously without failure
  - Safety
    - When a system temporarily fails, nothing catastrophic happens
  - Maintainability
    - A failed system can be easily repaired
- Faults
  - Transient faults, intermittent faults, permanent faults

Failure Models

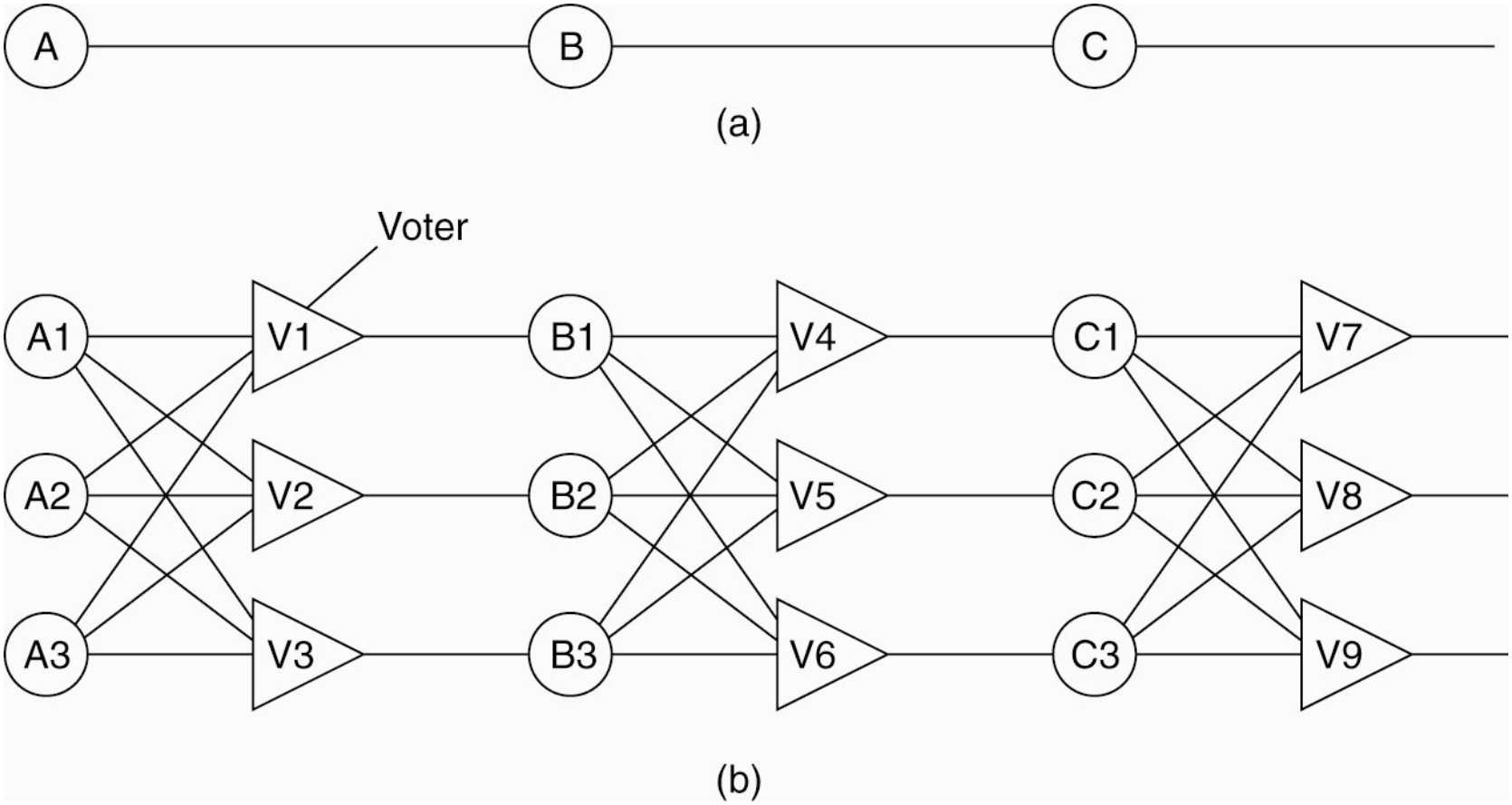
Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response 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

Different types of failures.

Failure Masking by Redundancy

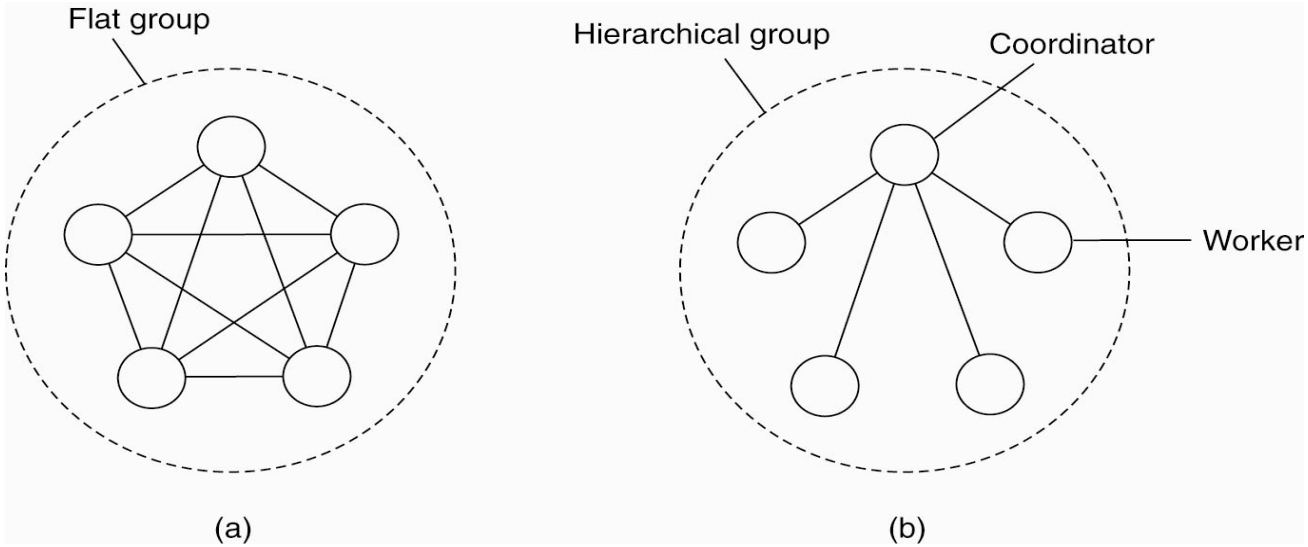
- Redundancy is the key technique for achieving fault tolerance
  - Information redundancy
    - Extra bits are added to be able to recover from errors
  - Time redundancy
    - The same action is performed multiple times to handle transient or intermittent faults
  - Physical redundancy
    - Extra equipment or processes are added to tolerate malfunctioning components

Example: Triple Modular Redundancy



Process Resilience

- Protection agains process failure
  - achieved by replicating processes into groups
  - a message to this group should be received by all members
    - thus, if one process fails, others can take over
- Internal structure of process groups
  - flat groups v.s. hierarchical groups



Failure Masking and Replication

- A key question is: how much replication is needed to achieve fault tolerance
- A system is said to be k fault tolerant if
  - it can survive faults in k components and still meet its specification
- If the components (e.g., processes) fail silently, then having k+1 replicas is enough
- If the processes exhibit Byzantine (arbitrary) failures, a minimum of 2k+1 replicas are needed

Agreement in Faulty Systems

- The processes in a process group needs to reach an agreement in many cases
  - It is easy and straightforward when communication and processes are all perfect.
  - However, when they are not, we have problem
- The goal is to have all non-faulty process reach consensus in a finite number of steps
- Different solutions may be needed, depending on:
  - Synchronous versus asynchronous systems.
  - Communication delay is bounded or not.
  - Message delivery is ordered or not.
  - Message transmission is done through unicast or multicast.

Byzantine Generals Problem (1)

- One paper
  - "The Byzantine Generals Problem", by Lamport, Shostak, Pease, In ACM Transactions on Programing Languages and Systems, July 1982
- Settings
  - Several divisions of the Byzantine army are camped outside an enemy city,
    - each division commanded by its own general.
  - After observing the enemy, they must decide upon a common plan of action
  - However, some generals may be traitors
    - trying to prevent the loyal generals from reaching agreement

Byzantine Generals Problem (2)

- Must guarantee that
  - All loyal generals decide upon the same plan of action.
  - A small number of traitors cannot cause the loyal generals to adopt a bad plan.
- A straightforward approach: simple majority voting
  - However, traitors may give different values to others
- More specifically
  - If the i-th general is loyal, then the value that he sends must be used by every loyal general as the value of v(i).

Byzantine Generals Problem (3)

- More precisely, we have:
  - a commanding general must send an order to his n-1 lieutenant generals such that
    - IC1. All loyal lieutenants obey the same order.
    - IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Impossibility Results

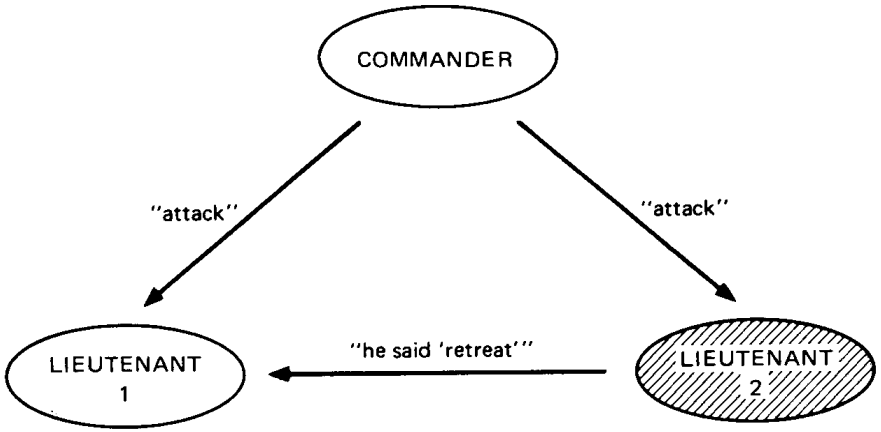


Fig. 1. Lieutenant 2 a traitor.

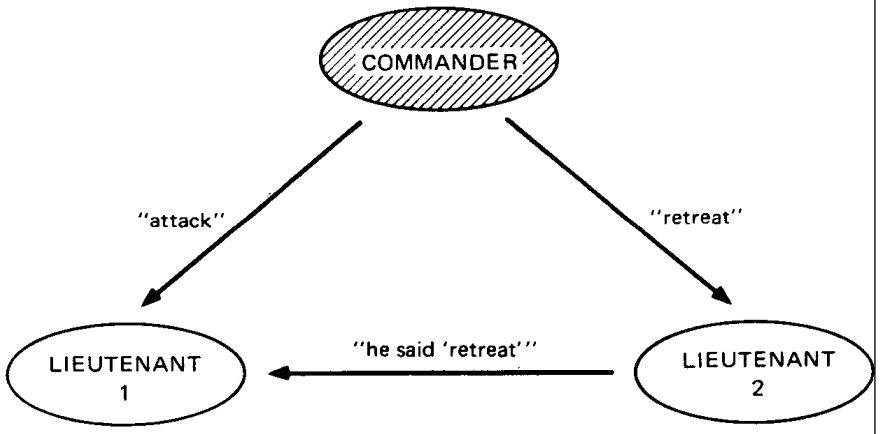
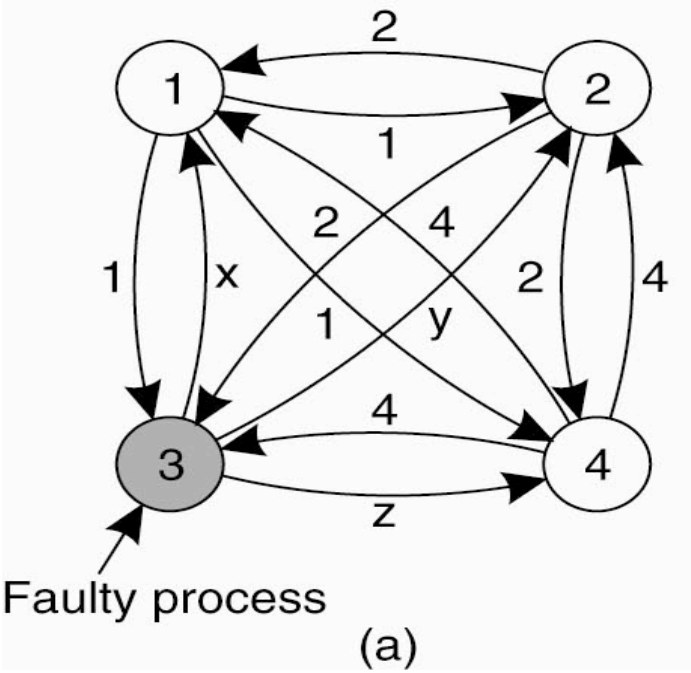


Fig. 2. The commander a traitor.

Lieutenant 1 sees the same information in two different scenarios

Byzantine Agreement Problem (1)

- The problem: reaching an agreement given
  - three non-faulty processes
  - one faulty process
- Assume
  - processes are synchronous
  - messages are unicast while preserving ordering
  - communication delay is bounded



Each process sends their value to the others.

Byzantine Agreement Problem (2)

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)

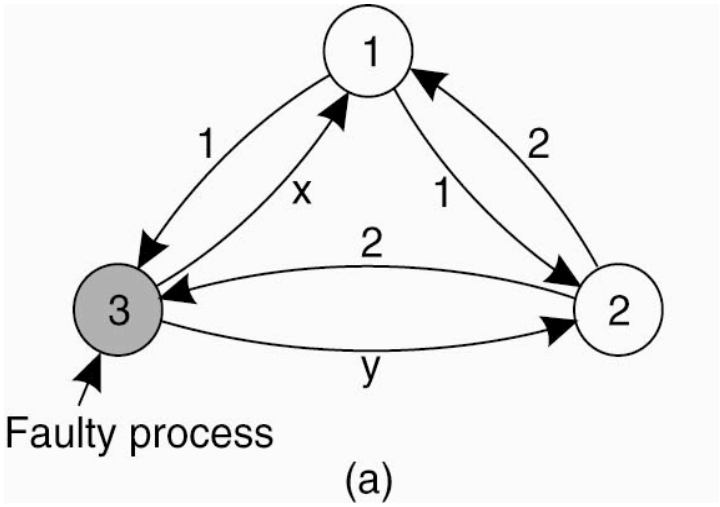
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The Byzantine agreement problem for three nonfaulty and one faulty process. (b) The vectors that each process assembles based on (a). (c) The vectors that each process receives in step 3.

Byzantine Agreement Problem (3)

- In a system with k faulty processes, an agreement can be achieved only if
  - 2k+1 correctly functioning processes are present, for a total of 3k+1 processes



(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)

BGP with Signed Messages

- Signed messages
  - Cryptographic guarantee
  - You cannot deny what you have said
- Generals only accept messages that have been correctly signed
- A traitor can be easily detected if he gives two different messages
- Only need k+2 generals to deal with k traitors
  - Note that the problem doesn't make sense for fewer than k+2 generals

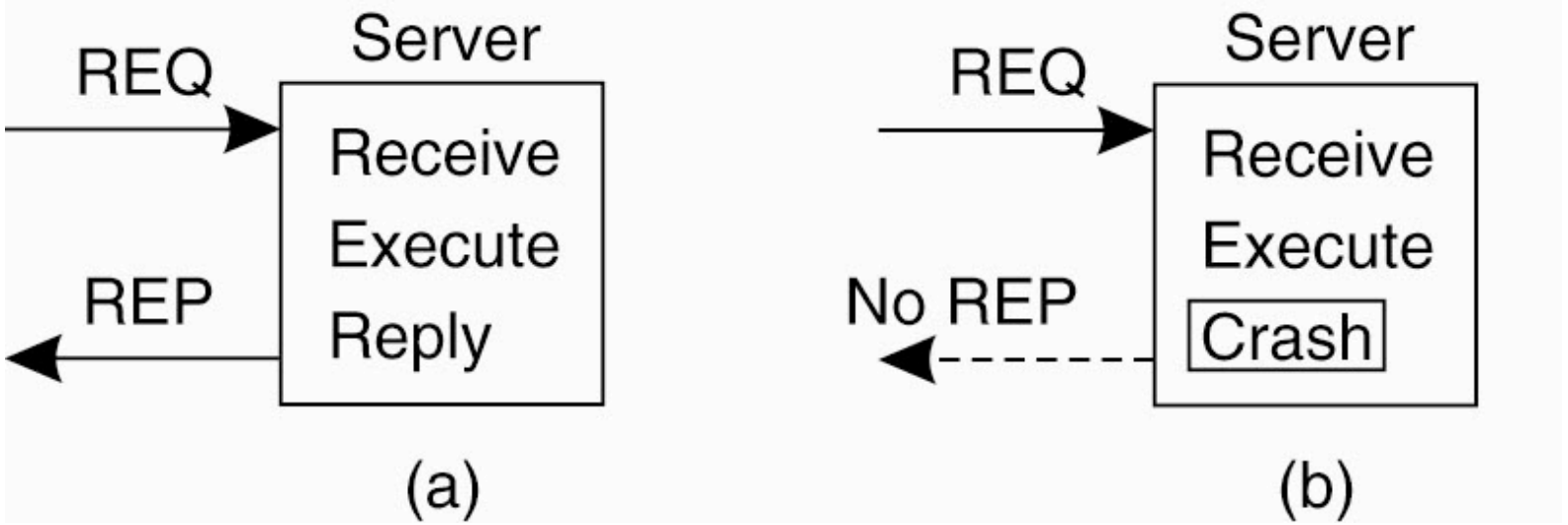
Failure Detection

- It is critical to detect faulty components
  - so that we can do proper recovery
- A common approach is to active ping processes with a time-out mechanism
  - Faulty if no response within a given time limit
  - Can be a side-effect of regular message exchanging
- The problem with the "ping" approach
  - It is hard to determine if no response is due to failure or just communication failure

Reliable Client-Server Communication

- In addition to process failures, another important class of failure is communication failures
- Point-to-point communication
  - Reliability can be achieved by protocols such as TCP
  - However, TCP itself may fail, and the distributed system will need to mask such TCP crash failure
- Remote procedure call (RPC): transparency is the challenge
  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 Crash

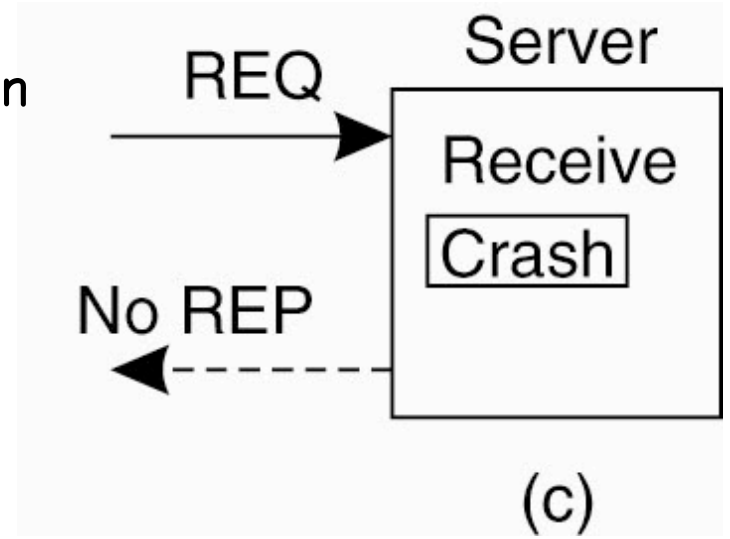


Three possibilities for a server in client-server communication:

(a) The normal case

(b) Crash after execution

(c) Crash before execution



Recovery from Server Crashes

- The challenge is that
  - A client does not know if server crashes before execution or crashes after execution
  - Two situations should be handled differently
- Three schools of thought for client OS
  - at least once semantics
  - at most once semantics
  - to guarantee nothing
- Ideally, we like exactly once semantics
  - But in general, there is no way to arrange this

Example: Printing Text (1)

- Assume the client
  - request the server to print some text
  - got ACK when the request is delivered
- Two strategies at the server
  - Send a completion message right before it tells the printer
  - Send a completion message after text has been printed
- The server crashes and then recover and announce to all clients that he is up and running again
  - The question is what the client should do?
  - The client does not know if its request will be actually carried out by the server

Example: Printing Text (2)

- Four strategies at the client
  - Never reissue a request: text may not be printed
  - Always reissue a request: text may be printed twice
  - Reissue a request only if it did not receive the acknowledgement of its request
  - Reissue a request only if it has received the acknowledgement of its request
- Three events that could happen at the server
  - Send the completion message (M), Print the text (P), and Crash (C)
  - Six different orderings: MPC, MC(P),PMC,PC(M),C(PM),C(MP)

Example: Printing Text (3)

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

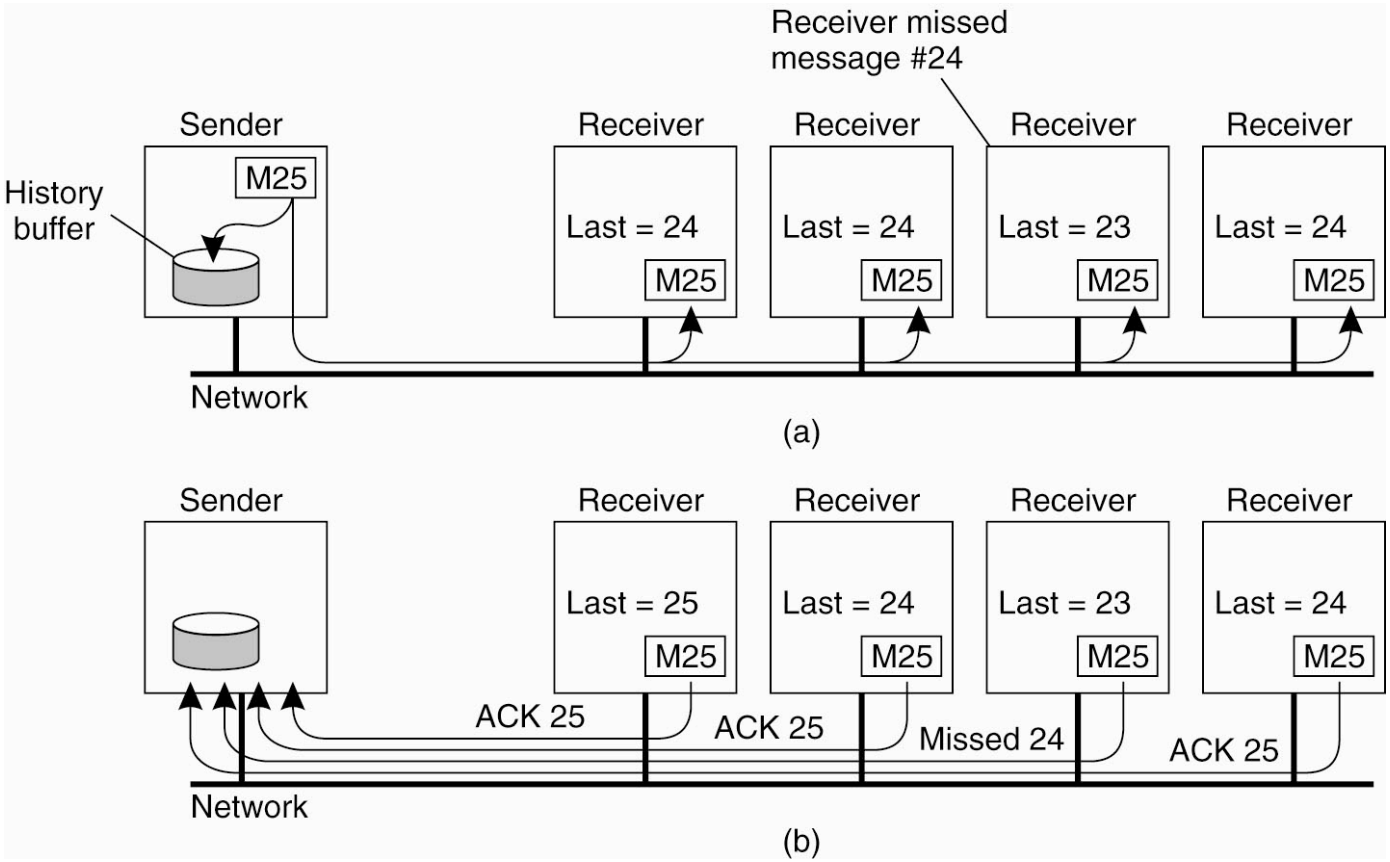
OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

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

Lost Reply Message

- A common solution is to set a timer
  - if the timer expires, send the request again
- However, the client cannot tell why there was no reply
  - The request gets lost in the channel? the reply gets lost in the channel? or the server is just slow?
- If the request is idempotent, then we can always reissue a request with no harm
  - We can structure requests in an idempotent way
  - However, this is not always true, e.g., transfer money
- Other possible solutions
  - Ask the server to keep a sequence number
  - Use a bit in the message indicating if it is the original request

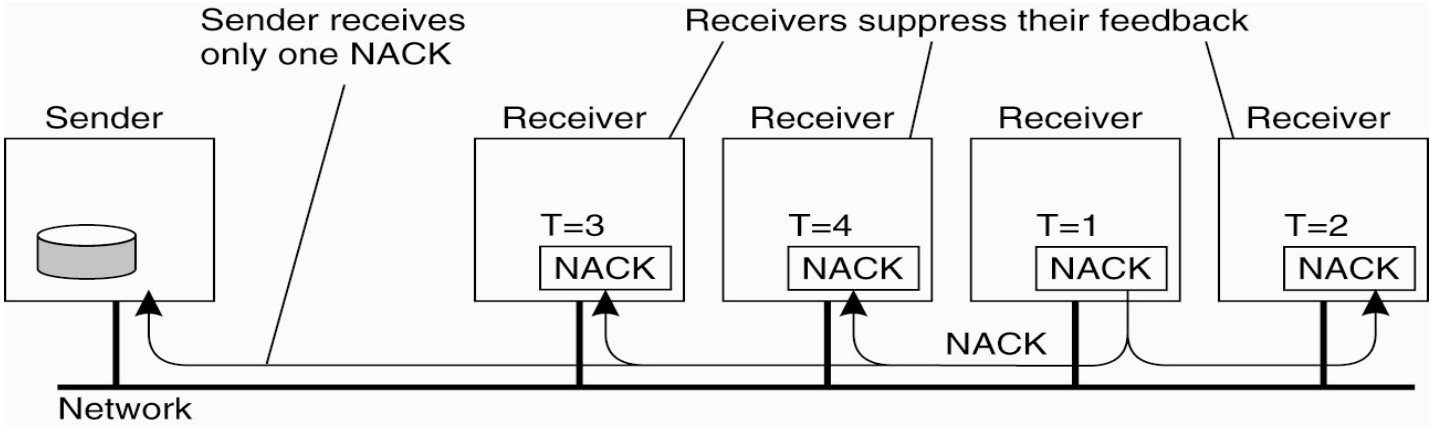
Basic Reliable-Multicasting Schemes



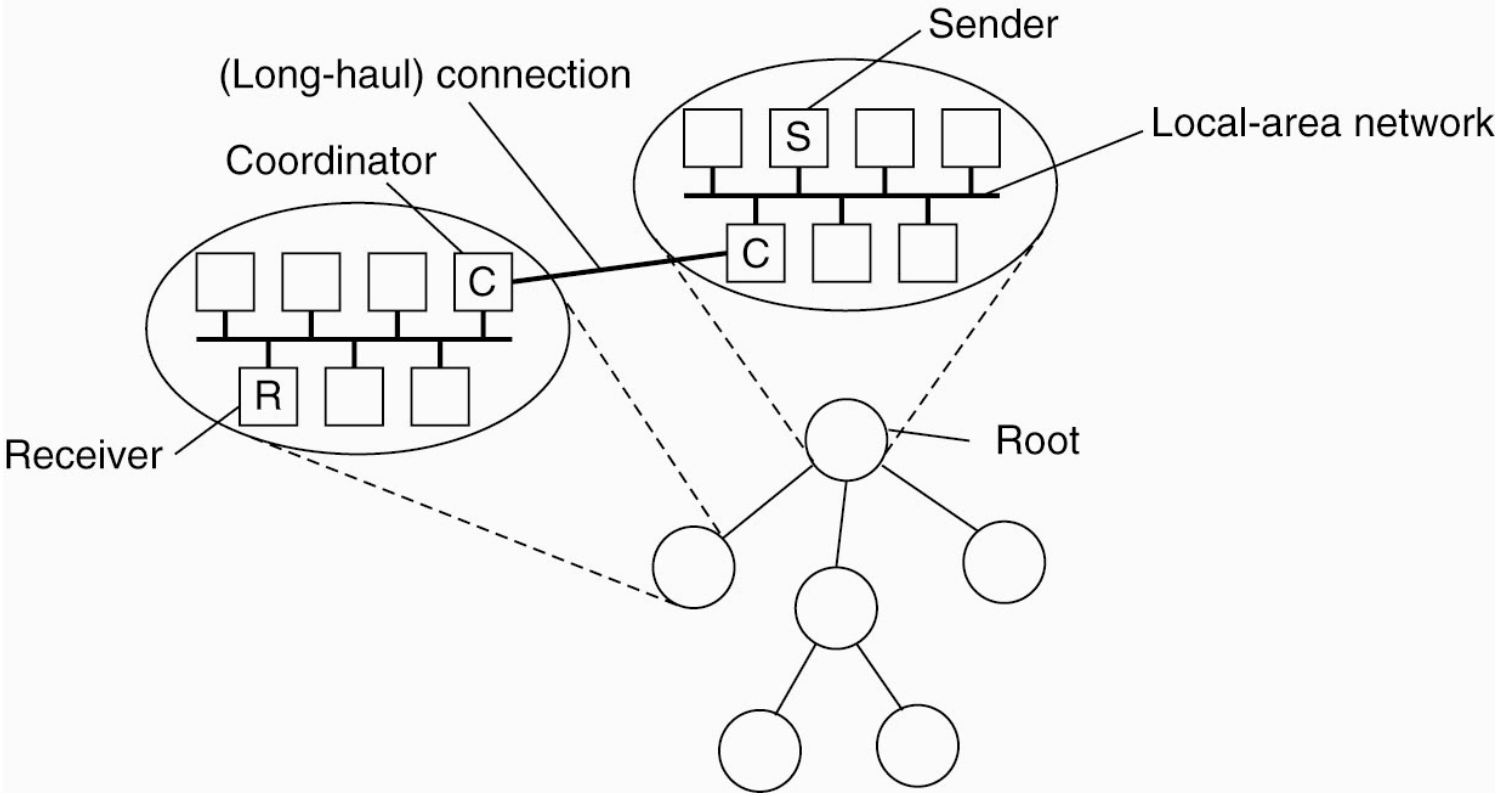
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail. (a) Message transmission. (b) Reporting feedback.

Scalability in Reliable Multicasting

- The basic scheme discussed has some limitations
  - If there are N receivers, the sender must be prepared to receive N ACKs
    - Let's only send NACKs (still no hard guarantee)
  - The sender has to keep old messages
    - Let's set a limit on the buffer (no retransmission for very old messages)
- Nonhierarchical feedback control
  - Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.



Hierarchical Feedback Control



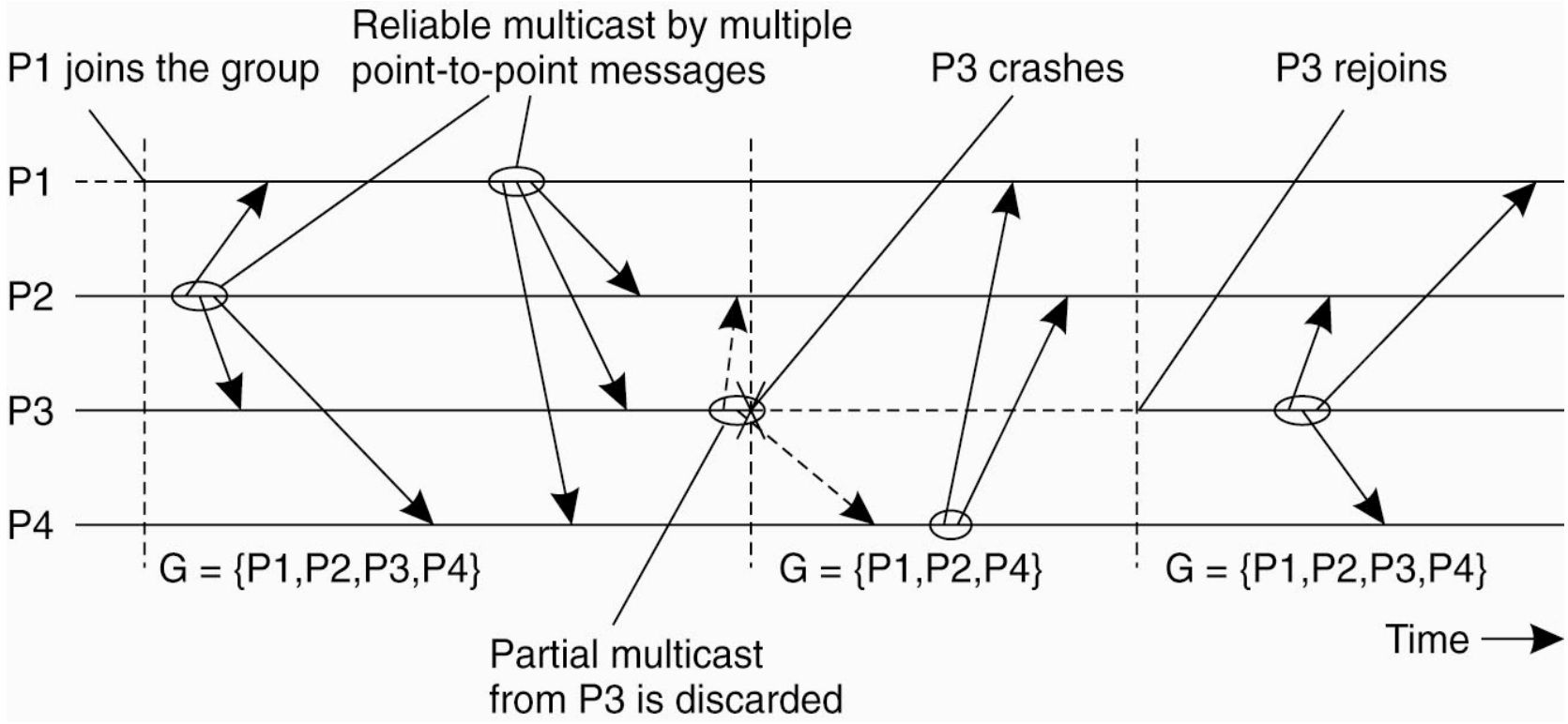
The essence of hierarchical reliable multicasting. Each local coordinator forwards the message to its children and later handles retransmission requests.

Atomic Multicast

- Consider a replicated database system constructed on top of a distributed system, we require that
  - An update should be either performed at all replicas or none at all
  - All updates should be done in the same order in all replicas
- The atomic multicast problem
  - A message is delivered to either all processes or to none
    - Virtually synchronous
  - Messages are delivered in the same order to all processes
    - Message ordering

Virtual Synchrony

- The principle of virtual synchronous multicast
  - No multicast can pass the view-change barrier

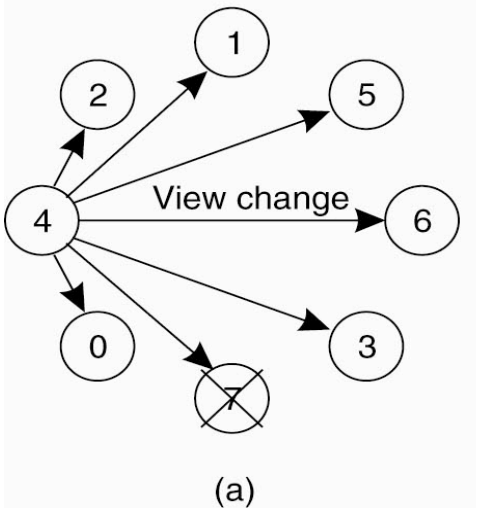


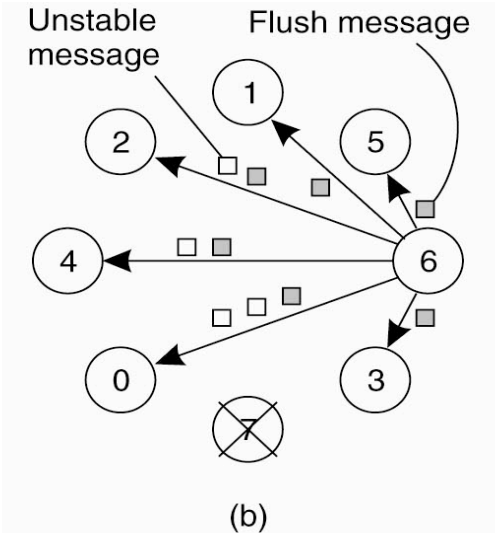
Message Ordering

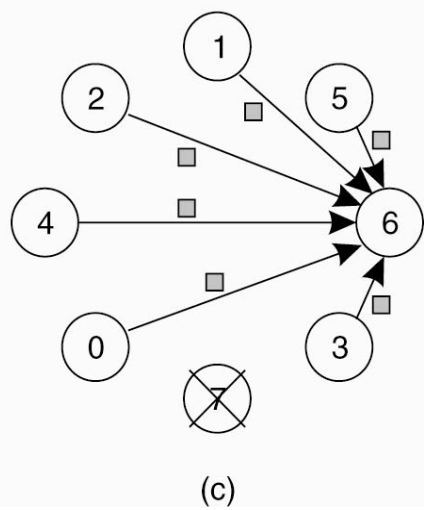
- Virtual Synchrony does not address the ordering of multicast
- There are four different cases
  - Unordered multicasts
    - Receivers may receive messages in a different order
  - FIFO-ordered multicasts
    - The messages from the same sender should be received in the same order as they are sent
  - Causally-ordered multicasts
    - If a message m_1 causally precedes m_2, then m_1 should be always received before m_1 at any receiver, even if the senders are different
  - Totally-ordered multicasts
    - Messages are delivered to all receivers in the same order
    - They may not be FIFO-ordered or causally-ordered

Implementing Virtual Synchrony

- What we will discuss is the implementation in Isis
  - a fault-tolerant distributed system that is used in industry for many years
- Assume point-to-point communication is reliable
- The task is to deliver all unstable messages before view changes
  - m is stable if one knows for sure that it has been received by all members



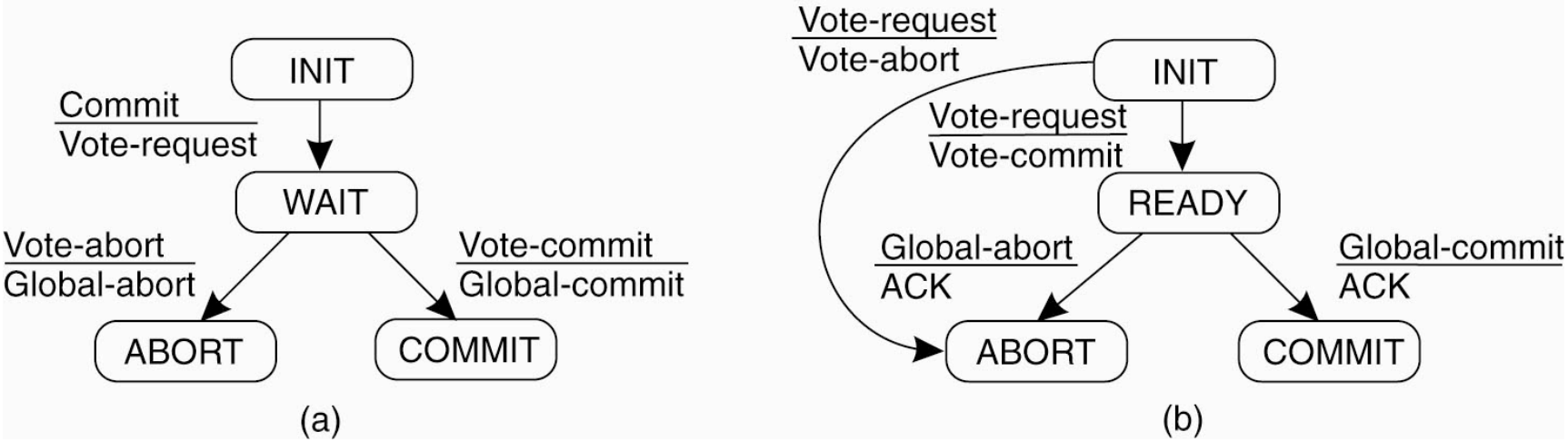




Distributed Commit

- Requires an operation being performed by all process in the group or none at all
  - Atomic multicasting is an example of this general problem (operation = delivery of a message)
- It is often achieved by means of an coordinator
  - One-phase commit protocol
    - The coordinators tells everyone what to do
    - No feedback when a member may fail to perform
  - Two-phase commit protocol
    - Cannot efficiently handle the failure of coordinator
  - Three-phase commit protocol

Two-Phase Commit



(a) The finite state machine for the coordinator

(b) The finite state machine for a participant

Handling Failures

- Both coordinator or participants may fail
  - Timeout mechanisms is often applied, and
  - Each saves its state to persistent storage (in a fault-tolerant way)
- If a participant is in INIT state
  - abort if no request from coordinator within a given time limit
- If the coordinator is in WAIT state
  - abort if not all votes are collected within a given time limit
- If a participant is in READY state
  - We cannot simply decide to abort since
    - A GLOBAL_COMMIT or GLOBAL_ABORT may have been issued
  - Let everybody block until coordinator recovers (Not the best option)
  - Contact other participants for more informed decision

Actions to Take in READY State

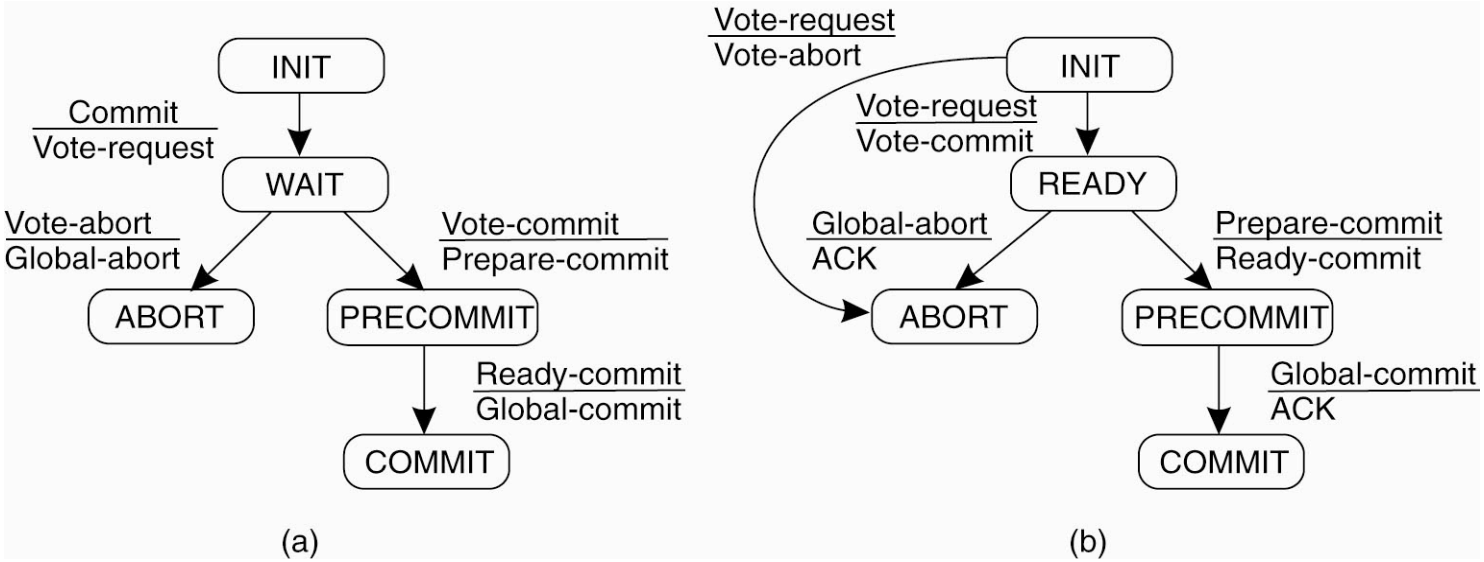
State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

what if all live participants are in READY state and one crashed with unknown state?

Actions taken by a participant P when residing in state READY and having contacted another participant Q.

Three-Phase Commit

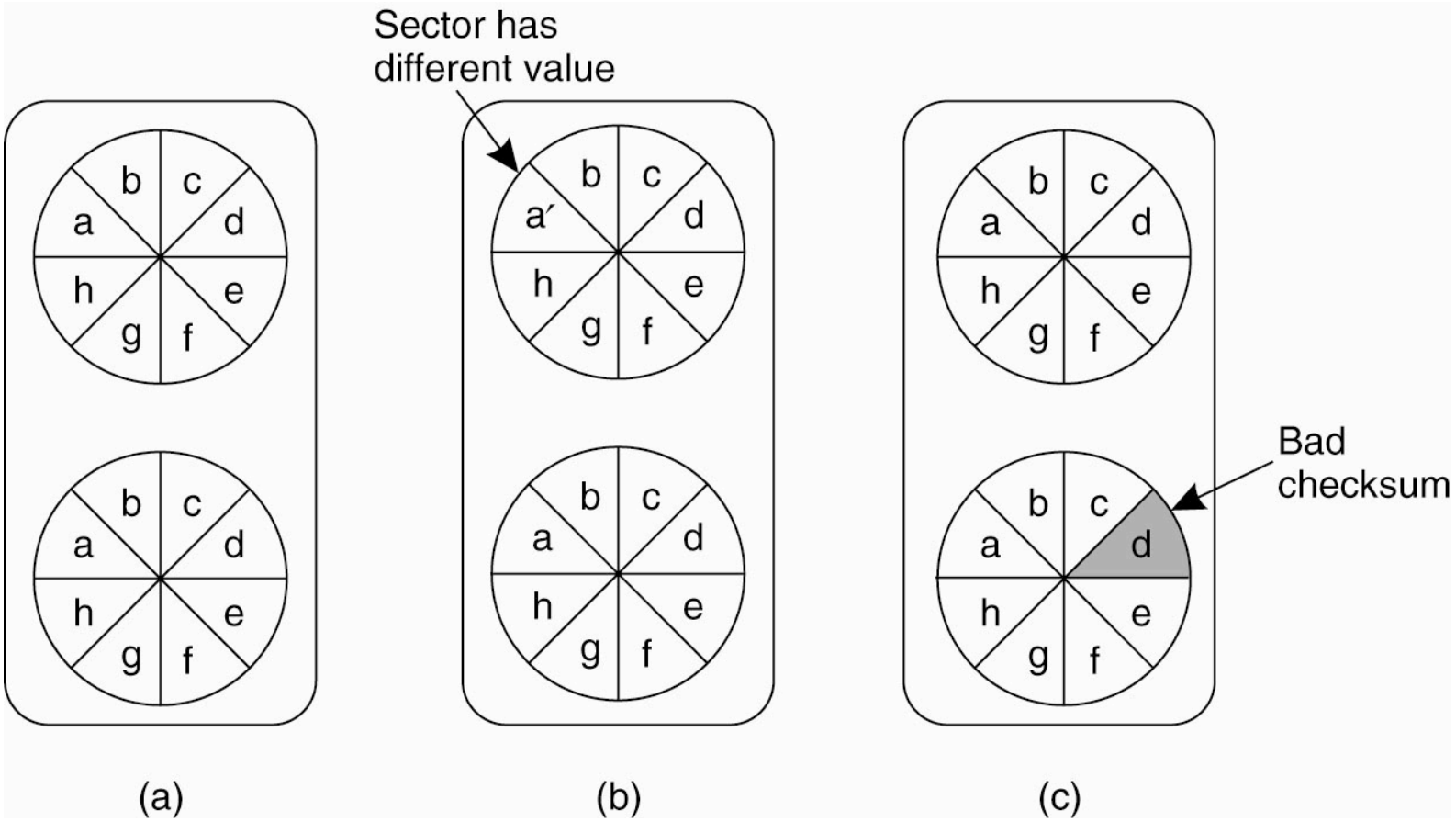
- **Two-phase commit is a blocking commit protocol**
  - When all participants are in READY state, no decision can be made until coordinator recovers



(a) The finite state machine for the coordinator

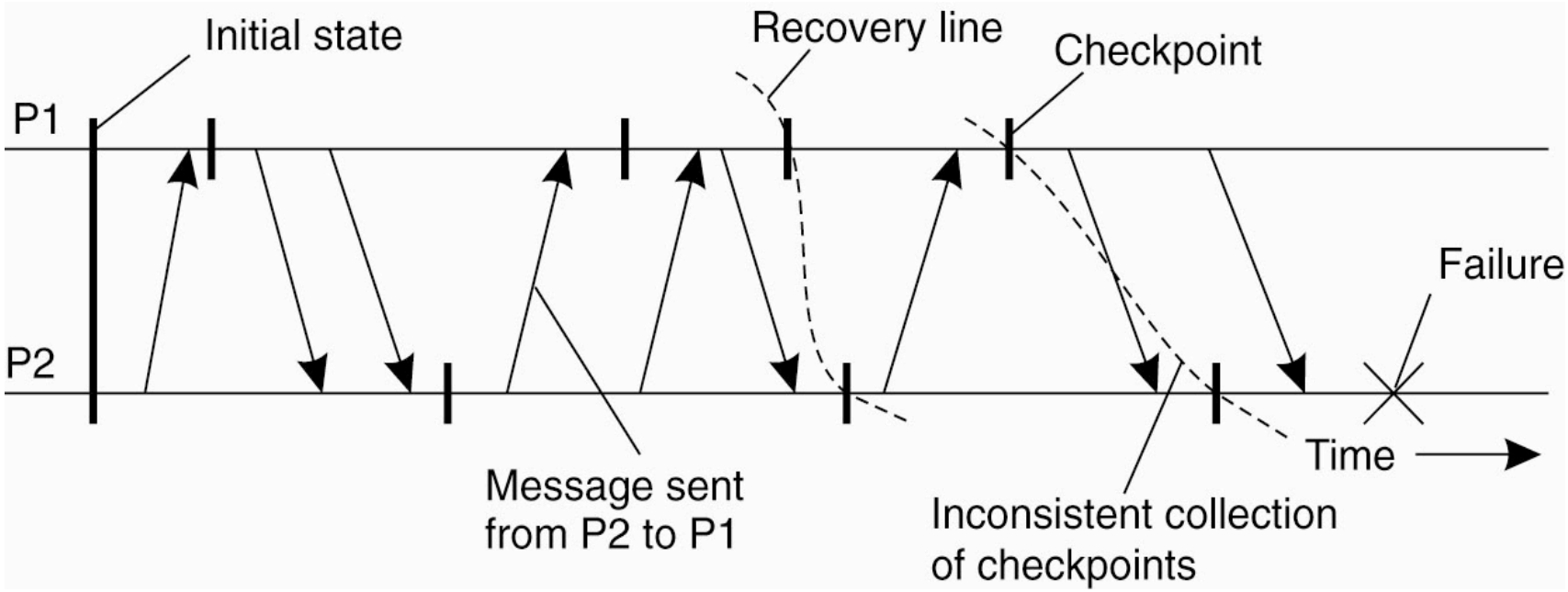
(b) The finite state machine for a participant

Recovery - Stable Storage



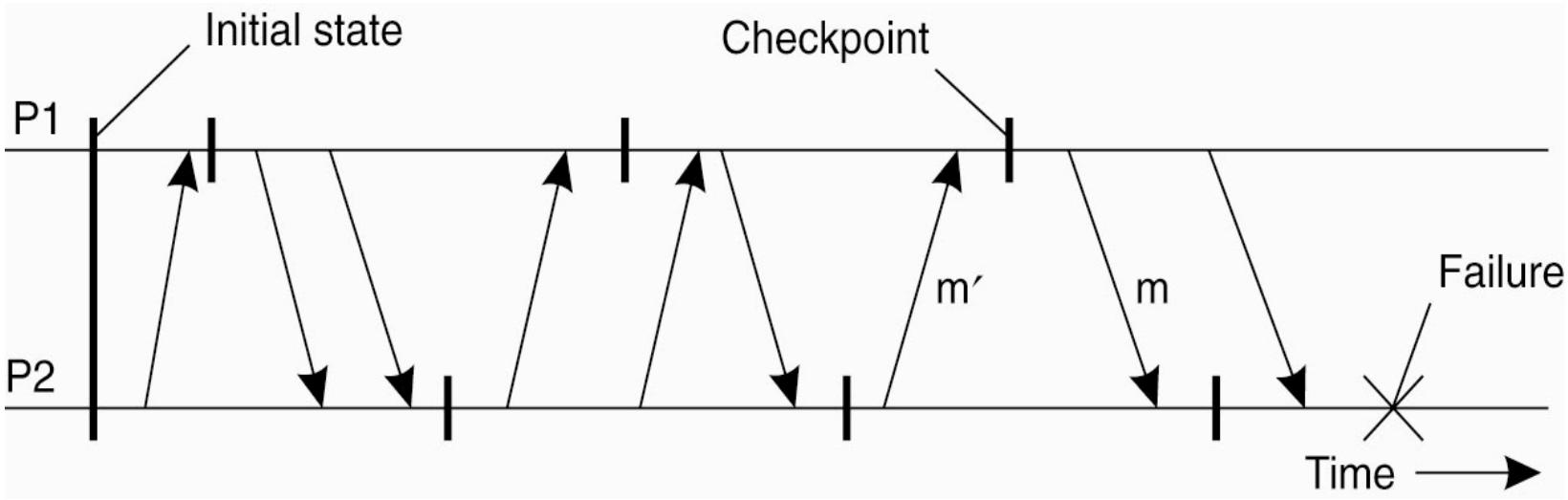
(a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

CheckPointing



A recovery line.

Independent Checkpointing



The domino effect.

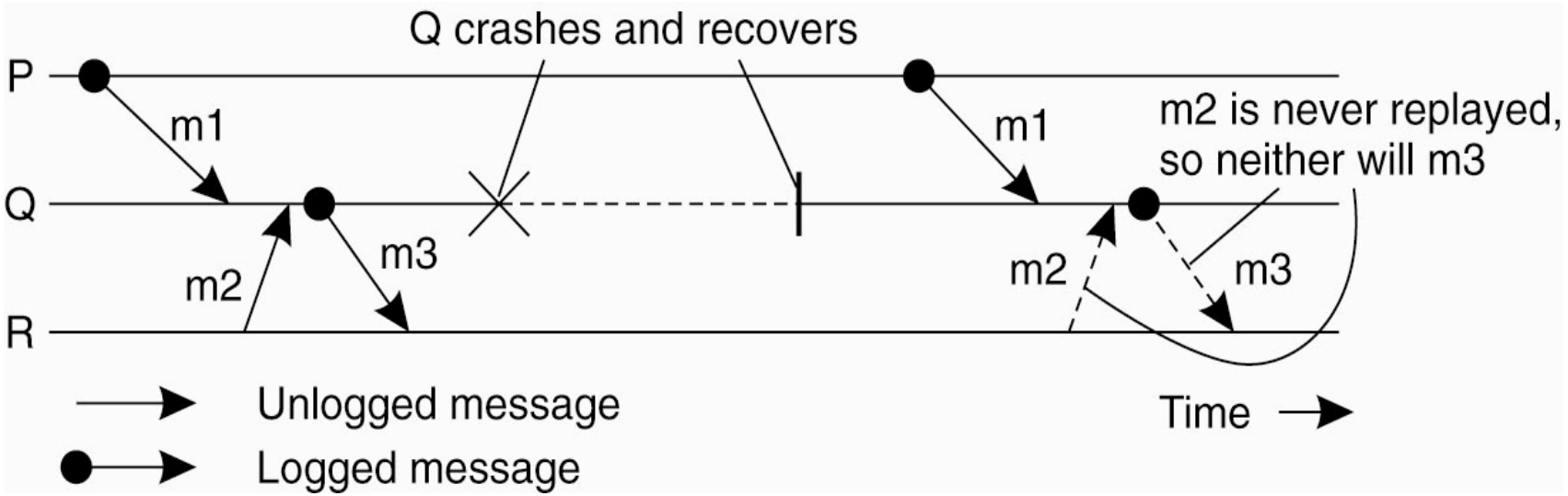
Coordinated Checkpointing

- Synchronize the checkpointing in all processes
  - the saved state is automatically globally consistent
- Achieved by using a two-phase blocking protocol
  1. The coordinator multicasts a request to do checkpoint
  2. Upon receiving such request, a process queues any subsequent message and notify the coordinator that it has take a checkpoint
  3. When the coordinator receives all notifications, it multicasts a CHECKPOINT_DONE message
  4. Everyone moves forward after seeing CHECKPOINT_DONE

Message Logging

- Checkpointing is expensive,
  - it is thus important to reduce the number of checkpointing
- The main intuition is
  - if we can replay all the transmissions since the last checkpoint, we can reach a globally consistent state
  - I.e., trade off communication with frequent checkpointing
- The challenge of message logging is how to deal with the orphan process
  - i.e., the process survived the crash, but is in an inconsistent state with the crashed process after recovery

Orphan Process - An Example



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

Exact Meaning of Orphan Process

- A message m is said to be stable if
  - it can no longer be lost, e.g., it has been written to stable storage
- DEP(m): include processes that depend on the delivery of m
  - i.e., the processes to which m has been delivered
  - If m' causally depends on m, then DEP(m') $\subset$ DEP(m)
- COPY(m): include processes that have a copy of m, but m has not been written to stable storage (m is not stable)
  - If all these process crashes, we can never replay m
- Orphan process Q can then be precisely defined as
  - There exists m such that Q$\in$DEP(m) but everyone in COPY(m) has crashed, i.e., it depends on m but m can no longer be replayed

Handling Orphan Process

- Our objective can thus be explained as
  - To ensure that if each process in COPY(m) crashes, then no surviving process left in DEP(m), i.e., DEP(m)⊆COPY(m)
- Thus, whenever a process becomes dependent on m, it should keep a copy of m
  - This is hard since it may be too late when you realize the you are dependent on m
- Pessimistic logging protocols: ensure that
  - each nonstable message is delivered to at most one process, i.e., there is at most one process dependent on a nonstable message
- Optimistic logging protocols
  - Any orphan process is rolled back so that it is not in DEP(m)