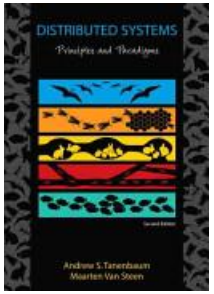
Chapter 8: FAULT TOLERANCE II

Continue to operate even when something goes wrong!



Thanks to the authors of the textbook [**TS**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.

Turgay Korkmaz
korkmaz@cs.utsa.edu

Chapter 8: FAULT TOLERANCE

- INTRODUCTION TO FAULT TOLERANCE
  - Basic Concepts, Failure Models
- PROCESS RESILIENCE
  - Design Issues, Failure Masking and Replication
  - Agreement in Faulty Systems, Failure Detection
- RELIABLE CLIENT-SERVER COMMUNICATION
  - Point-to-Point Communication, RPC Semantics -- SELF-STUDY
- RELIABLE GROUP COMMUNICATION
  - Basic Reliable-Multicasting Schemes, Scalability
  - Atomic Multicast
- DISTRIBUTED COMMIT
  - Two-Phase Commit, Three-Phase Commit
- RECOVERY
  - Introduction
  - Checkpointing
  - Message Logging
  - Recovery-Oriented Computing

Objectives

- To understand failures and their implications
- To learn about how to deal with failures
-

In addition to faulty processes, we need to consider communication failures…

RELIABLE COMMUNICATION

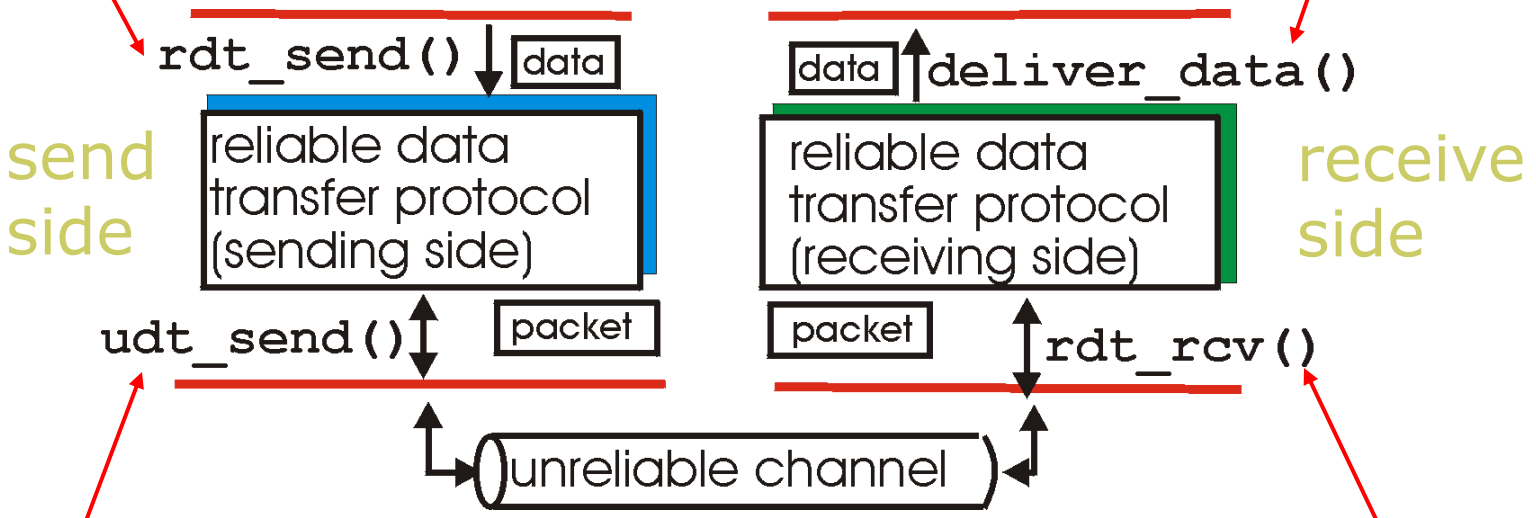
Reliable Communication

- Previous models equally apply to communication channels, too
  - Crash connection is lost
  - Omission lost or corrupted msg
  - Timing response outside the expected time frame
  - Arbitrary (both non- and malicious) duplicate packets
- How can we mask the above errors to provide Reliable Data Transfer (RDT)
- In practice, most techniques focus on crash and omission faults
  - TCP tries to hide omission, but it cannot hide crash
  - To hide crash, middleware tries to re-establish connections

Reliable data transfer: getting started

`rdt_send()`: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

`deliver_data()`: called by **`rdt`** to deliver data to upper



`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

`rdt_rcv()`: called when packet arrives on rcv-side of channel

From Computer Networking by Kurose and Ross.

General mechanisms for RDT

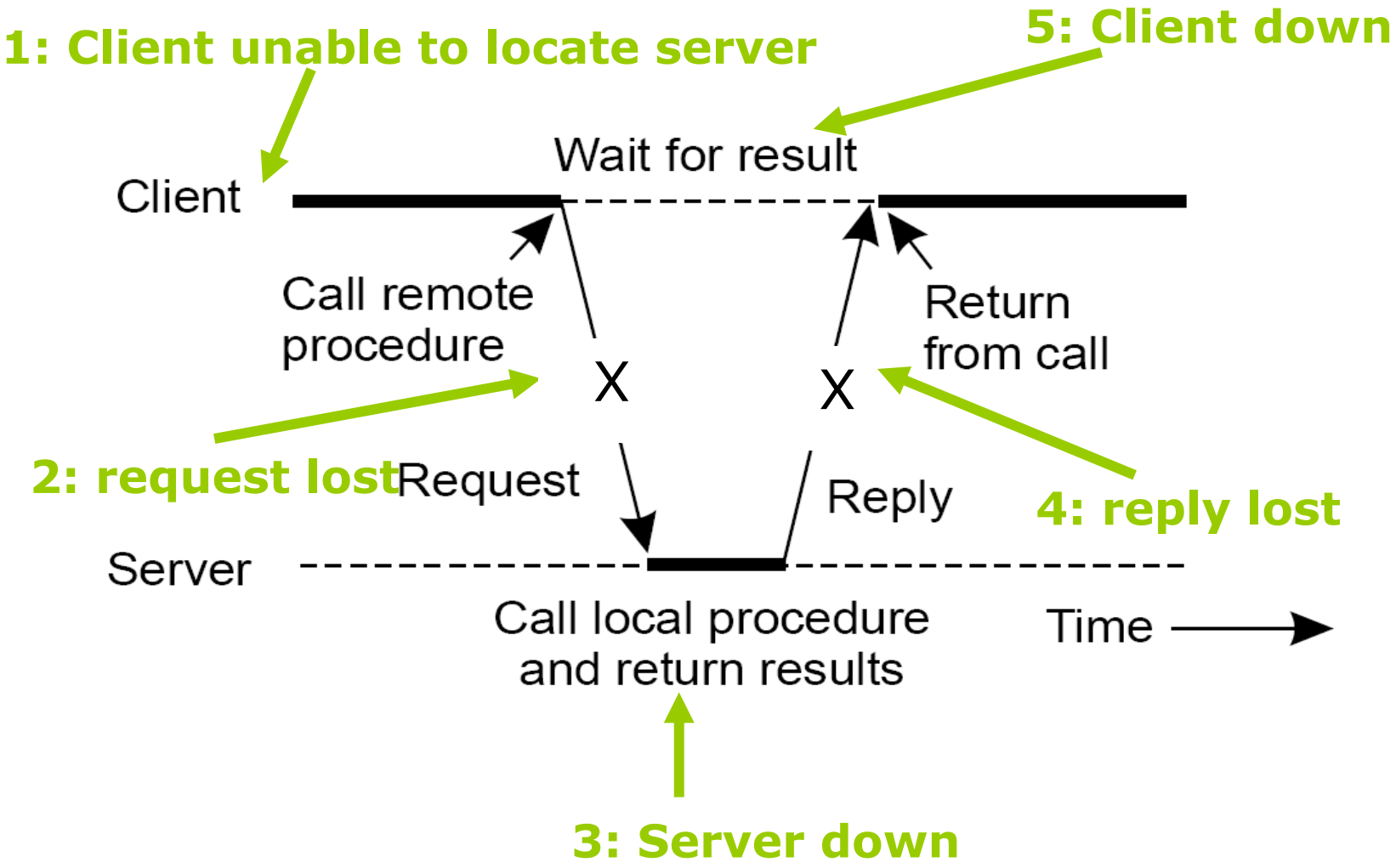
- Error detection
  - Checksum or CRC to detect bit errors
- Receiver feedback: control msgs (ACK,NAK)
- Timeout to detect packet loss
- Retransmissions
  - but can't just retransmit: possible duplicate
  - add *sequence number* to each pkt
- Error correction
  - Add so much information redundancy that corrupted packets can be automatically *corrected; CRC codes*

What may go wrong?

What to do when there is a failure?

RPC SEMANTICS WITH FAILURES

What may go wrong during RPC?



What to do?

RPC Semantics with Failures

1: Client unable to locate server

- Relatively simple – just report back to client (exception)
- But having to write exception handling destroys transparency

2: Request lost

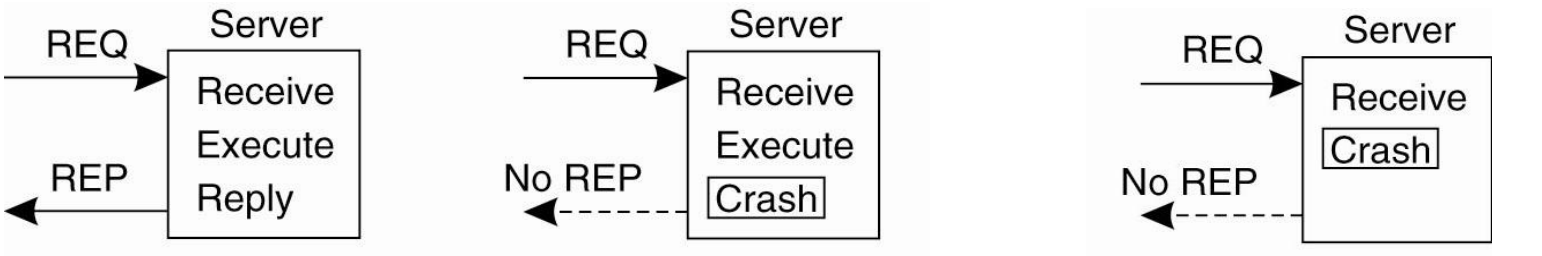
- Just resend message upon timeout
- How to set timeout value?
- Use sequence numbers to detect duplicate requests

What to do?

RPC Semantics with Failures (cont'd)

3: Server down

- Client does not know which is which?



(a) The normal case. (b) Crash after execution. (c) Crash before execution.

- What should we do or expect from server?
  - Ideally, exactly once (but it is not easy to realize)
  - At-least-once-semantics: The server guarantees that it will carry out an operation at least once, no matter what
  - At-most-once-semantics: The server guarantees that it will carry out an operation at most once
  - Guarantee nothing (perform rpc 0 to ∞ times)

What to do?

RPC Semantics with Failures (cont'd)

4: Reply 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
- Try to structure all the operations as idempotent
  - **idempotent**: repeatable without any harm done if it happened to be carried out before
- But some are not idempotent (e.g., money transfer):
  - client assigns a sequence number to each request and server keep tracks of these request
  - Server refuses to perform the same request a second time
  - Server stores the results from first time and send it back to the client (but how long?)

What to do?

RPC Semantics with Failures (cont'd)

5: Client down

- The server is doing work and holding resources for nothing (orphan computation)
  - Waste CPU cycles
  - Lock files or other valuable resources
- To do deal with orphan computation
  - **Extermination**: client stub logs its requests, and upon reboot, explicitly kills orphans
  - **Re-incarnation**: Broadcast new epoch number when recovering ⇒ servers kill orphans
  - **Gentle Re-incarnation** : server tries to locate the owner before it kills orphans
  - **Expiration**: Require computations to complete in a *T* time units. Old ones are simply removed

Example: Server Crashes (1)

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

- ___ M ____ P ____
- ___ P ____ M ____

Example: Server Crashes (2)

- 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.

Example: Server Crashes (3)

- Different combinations of client and server strategies in the presence of 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

RELIABLE GROUP COMMUNICATION

Reliable Multicasting

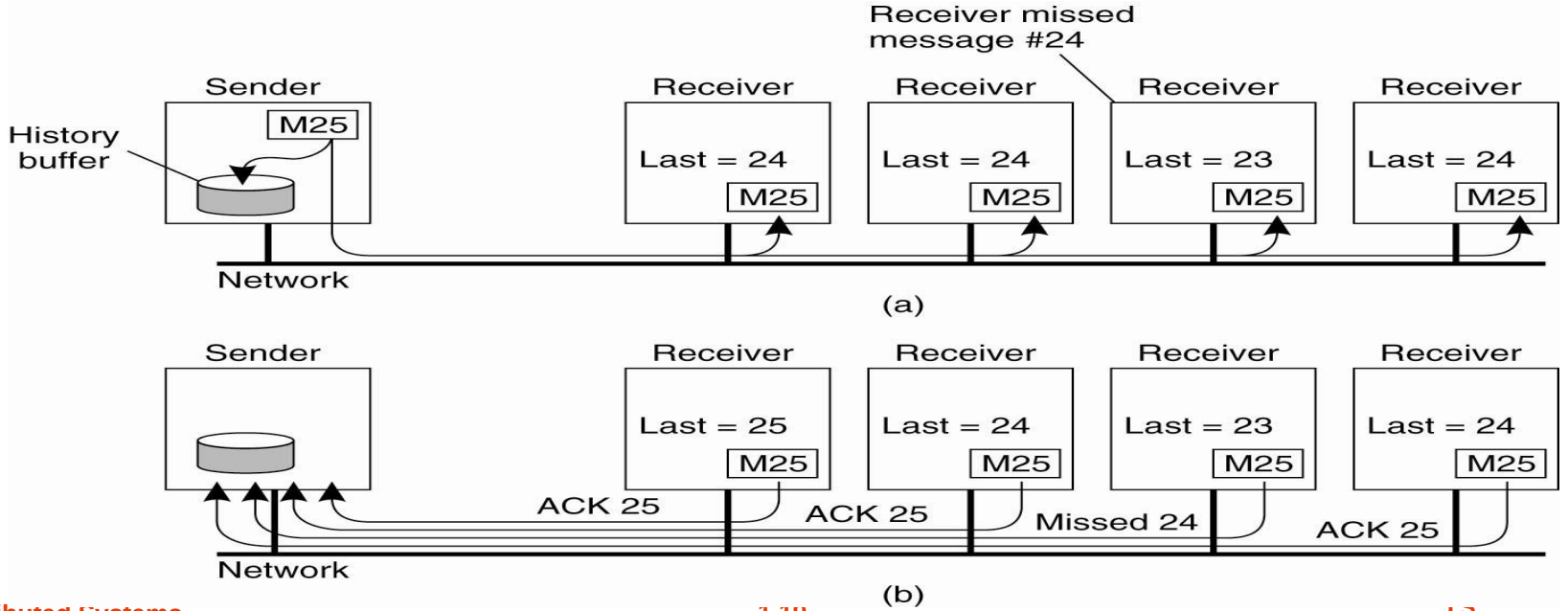
- **Model:** We have a **multicast channel** *c* with two (possibly overlapping) groups
  - **The sender group** *SND*(*c*) of processes that *submit* messages to channel *c*
  - **The receiver group** *RCV*(*c*) of processes that can receive messages from channel *c*
- **Basic Reliable Multicast:**

  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

- Let the sender broadcast the messages to channel *c* and log them
  - If *P* sends message *m*, *m* is stored in a **history buffer**
  - Each receiver acknowledges the receipt of *m*, or requests retransmission from P when noticing msg lost
  - Sender *P* removes *m* from history buffer when everyone has acknowledged receipt

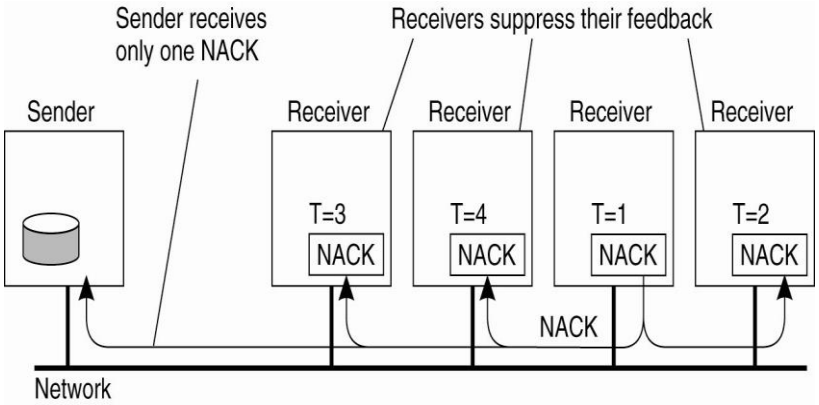


Basic Reliable-Multicasting Improvements

Basic scheme is not scalable

Improvements:

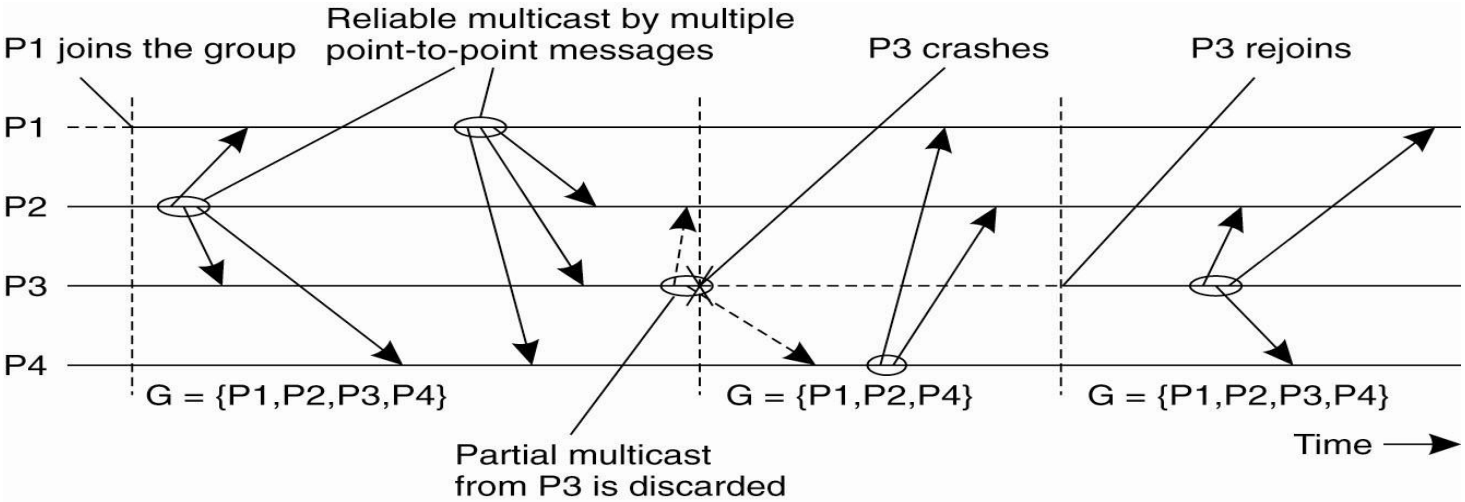
- Piggyback ACKs
- Just send Neg ACK



- Use point-to-point reliable channels for re-transmission
  - Sender may keep all sent msg in buffer (worst case)
- Different schemes are proposed to reduce number of feedbacks
  - **Feedback suppression**: report only missing msg and multicast neg-ACK to all so they will not generate neg-ACK if they miss the same msg

Atomic Multicast

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



- A message is delivered only to the nonfaulty members of the current group.
- All members should agree on the current group membership → Virtually synchronous multicast.

Atomic Multicast

Why is this important?

- Consider a replicated database
  - All replicas need to get updates in the same order and all must get them or not at all.
- If we have just reliable multicast support,
  - Then the replicas that are down will miss some updates and cause inconsistency
- But if we have atomic multicast support, then
  - Either all replicas perform the same updates or none at all, so all replicas will be consistent
  - Faulty process can be taken out of group, so non-faulty ones can continue to provide consistent replication
  - When faulty ones come back, they first try to join the group and sync themselves with the rest of the group

Have an operation to be performed by each member of a process group or none at all.

Atomic multicast is an example of this more general problem

DISTRIBUTED COMMIT

One-Phase Commit Protocol

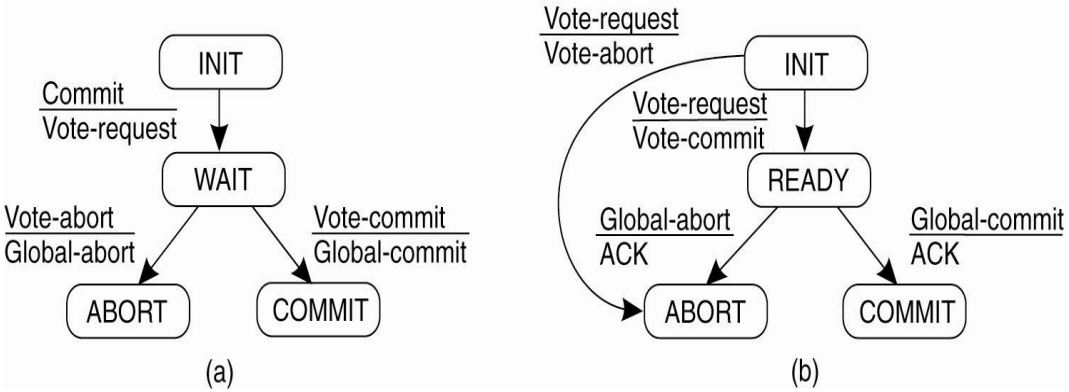
- Establish distributed commit by means of a coordinator
  - Simply tell all processes to (or not to) locally perform an operation
  - + simple
  - - but if one did not perform the operation, there is no way to tell this to the coordinator
- Accordingly, two-, three-phase protocols are introduced

Two-Phase Commit (1)

Assume there is no failure

The client who initiated the computation acts as **coordinator**; processes required to commit are the **participants**

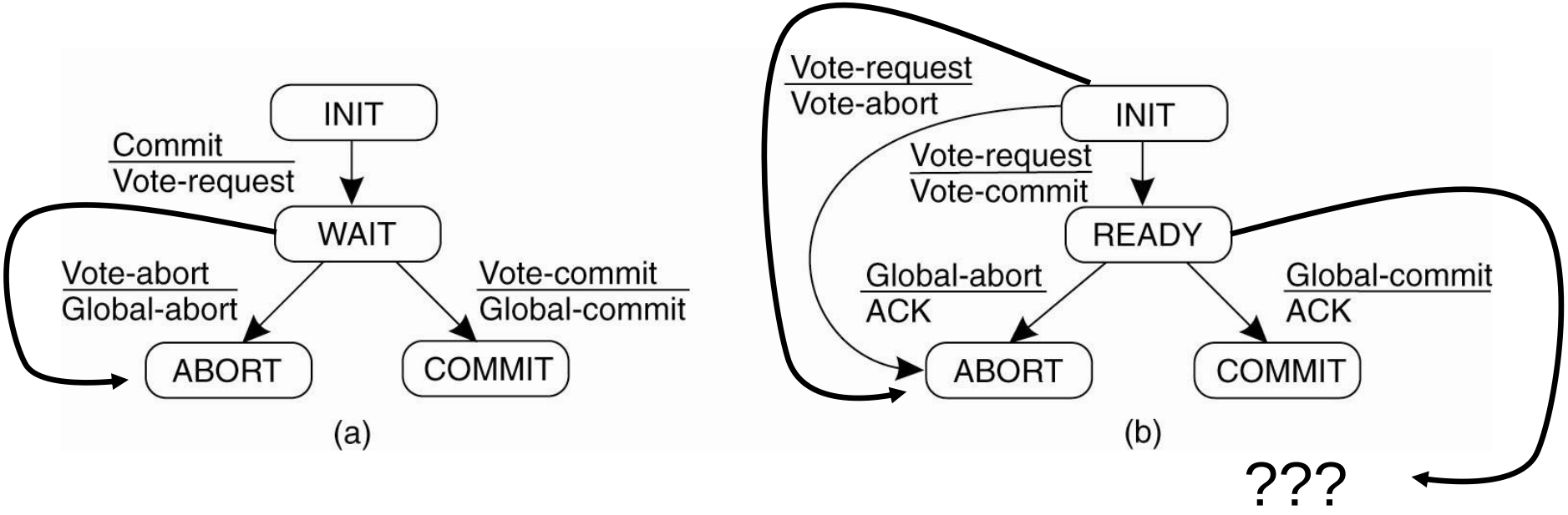
- Phase 1a: Coordinator sends vote-request to participants (also called a pre-write)
- Phase 1b: When participant receives vote-request it returns either vote-commit or vote-abort to coordinator. If it sends vote-abort, it aborts its local computation
- Phase 2a: Coordinator collects all votes; if all are vote-commit, it sends global-commit to all participants, otherwise it sends global-abort
- Phase 2b: Each participant waits for global-commit or global-abort and handles accordingly



Two-Phase Commit (2)

Problems arise when there is failure

- Coordinator (a) and participants (b) may wait one another forever…
- Introduce timeouts



Two-Phase Commit (3)

Problems arise when there is failure

- Simplest sol: Wait until the coordinator recovers!
- Better sol: Check state of other participants Q → no need to log coordinator's decision.

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 are in READY?

Two-Phase Commit (4)

Problems arise when there is failure

- Actions for participant crashes in state S, and recovers to S
  - **Initial** state: No problem: participant was unaware of protocol
  - **Ready** state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make → log the coordinator's decision
  - **Abort** state: Merely make entry into abort state idempotent, e.g., removing the workspace of results
  - **Commit** state: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Two-Phase Commit (5)

Problems arise when there is failure

Actions by coordinator:

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send VOTE_ABORT to coordinator;
}
```

Actions for handling decision requests: /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}
```

(b)

Two-Phase Commit (6)

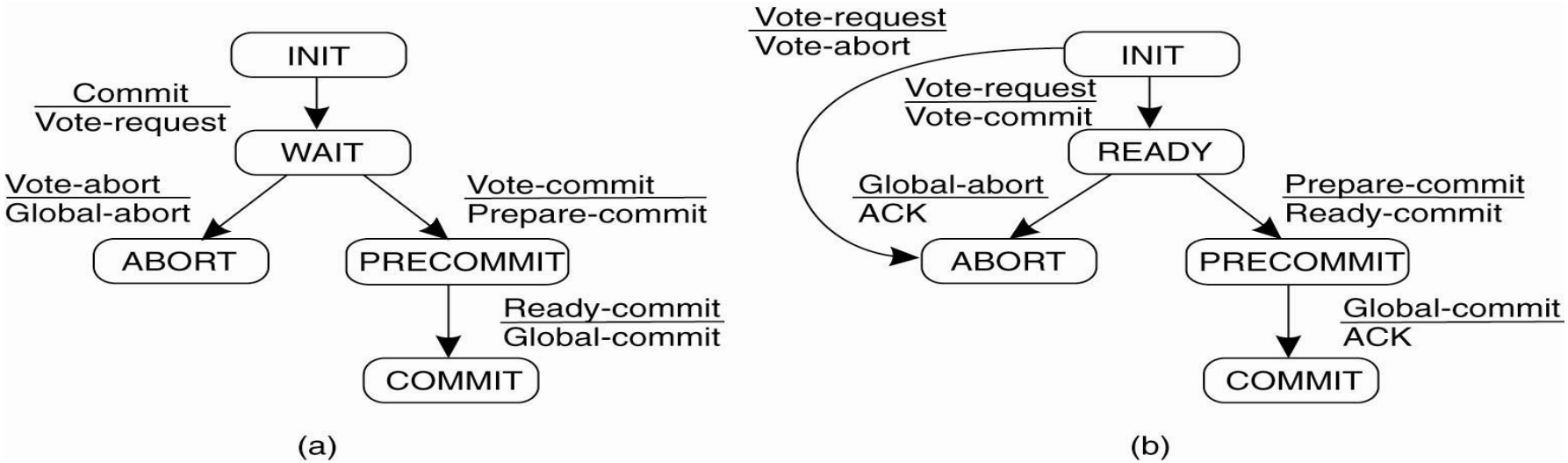
Problems arise when there is failure

- If coordinator fails, participants may not reach a final decision…
- If all participants are in the READY state, the protocol blocks.
- Apparently, the coordinator is failing.
- Participants need to be blocked until the coordinator recovers…
- To avoid blocking (in case of fail-stop),
  - Let a participant multicasts a received msg
  - Use three-phase commit ….

Three-Phase Commit (1)

Model (Again: the client acts as coordinator)

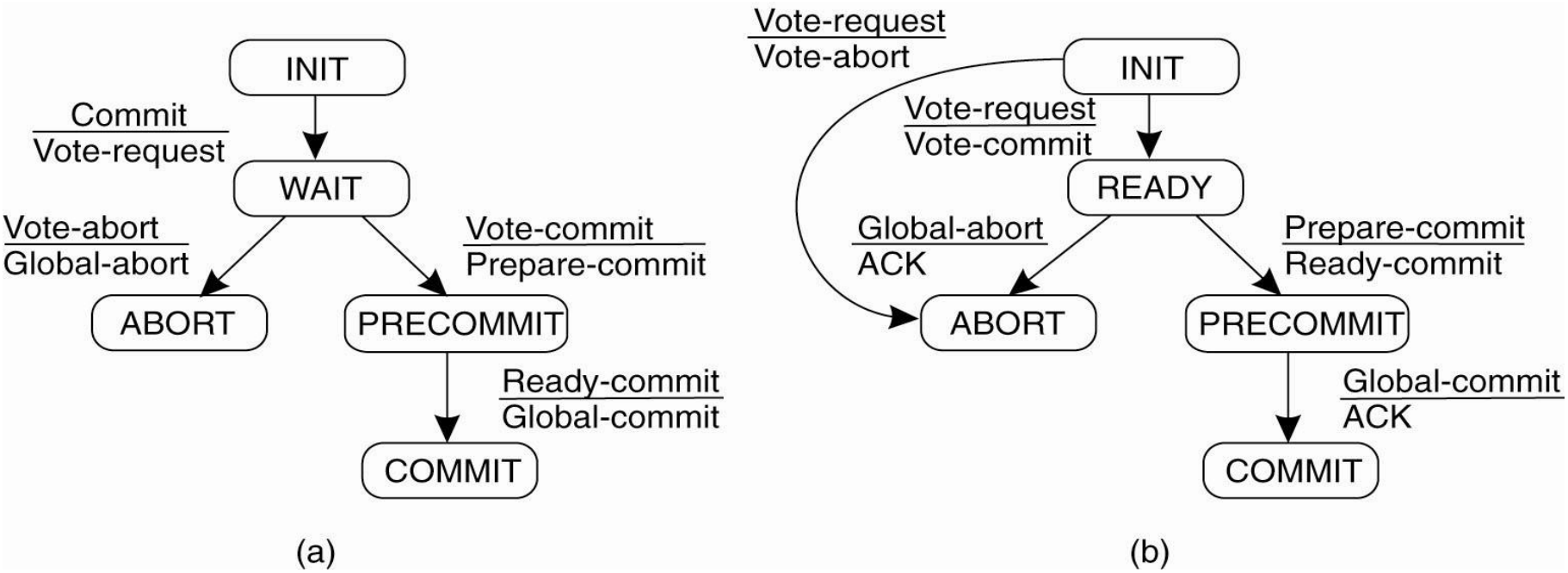
- Phase 1a: Coordinator sends vote-request to participants
- Phase 1b: When participant receives vote-request it returns either vote-commit or vote-abort to coordinator. If it sends vote-abort, it aborts its local computation
- Phase 2a: Coordinator collects all votes; if all are vote-commit, it sends prepare-commit to all participants, otherwise it sends global-abort, and halts
- Phase 2b: Each participant waits for prepare-commit, or waits for global-abort after which it halts
- Phase 3a: (Prepare to commit) Coordinator waits until all participants have sent ready-commit, and then sends global-commit to all
- Phase 3b: (Prepare to commit) Participant waits for global-commit



Three-Phase Commit (2)

To make the protocol non-blocking, the states of the coordinator and each participant satisfy the following two conditions:

1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.



Bring the system in an error-free state…

FAULT RECOVERY

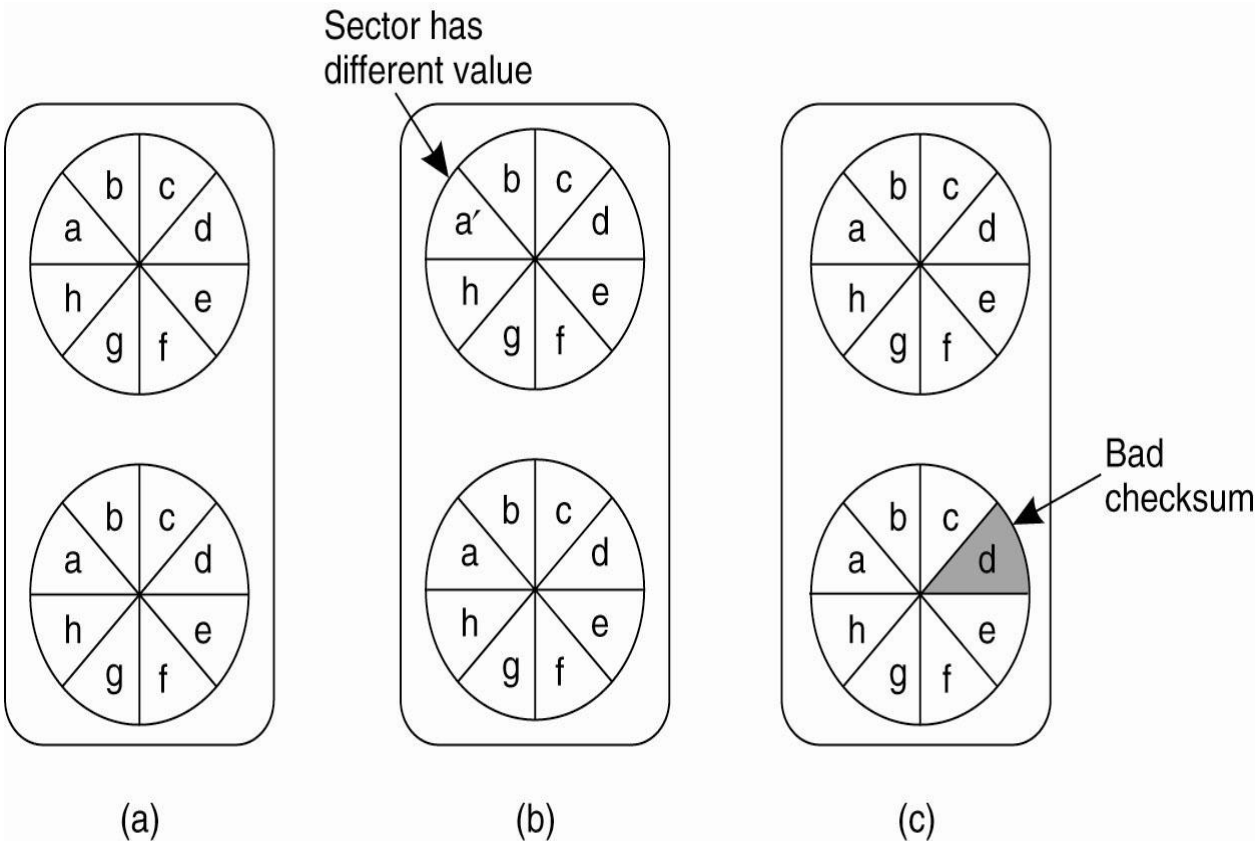
Fault Recovery

- **Backward** recovery
  - Bring the system back into a **previous** error-free state
  - E.g., packet retransmission
- **Forward** recovery
  - Find a new **future** state from which system can continue operation
  - E.g., Error-correction codes
- **In Practice:**

  Use backward error recovery, requiring that we establish recovery points (checkpoints)

Stable Storage

Designed to survive anything?



After a crash

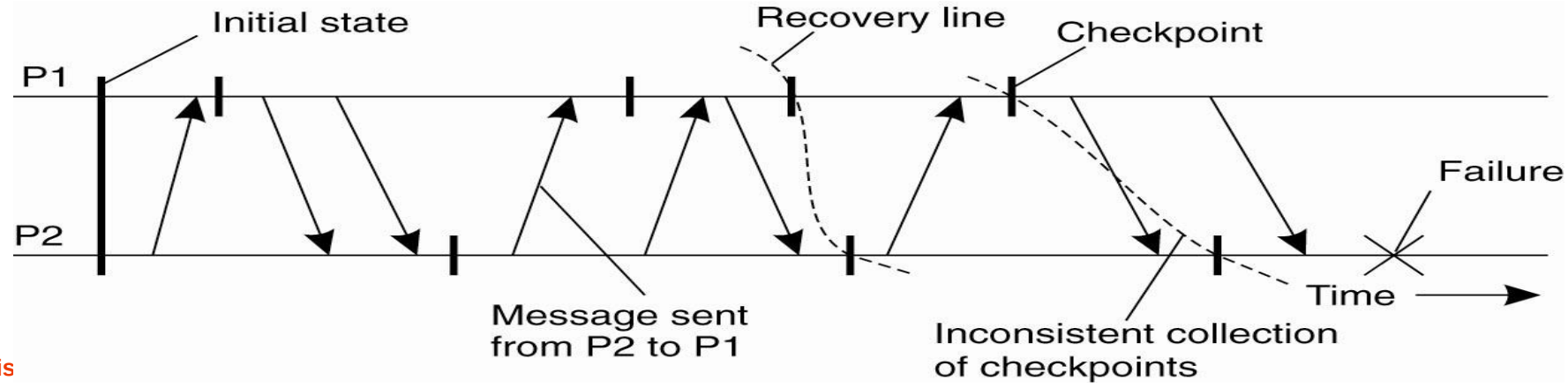
- If both disks are identical: you're in good shape.
- If one is bad, but the other is okay (checksums): choose the good one.
- If both seem okay, but are different: choose the main disk.
- If both aren't good: you're not in a good shape.

Main idea: **replicate** all data on **at least two disks**, and keep one copy "correct" at all times

What if both fail? Probability?

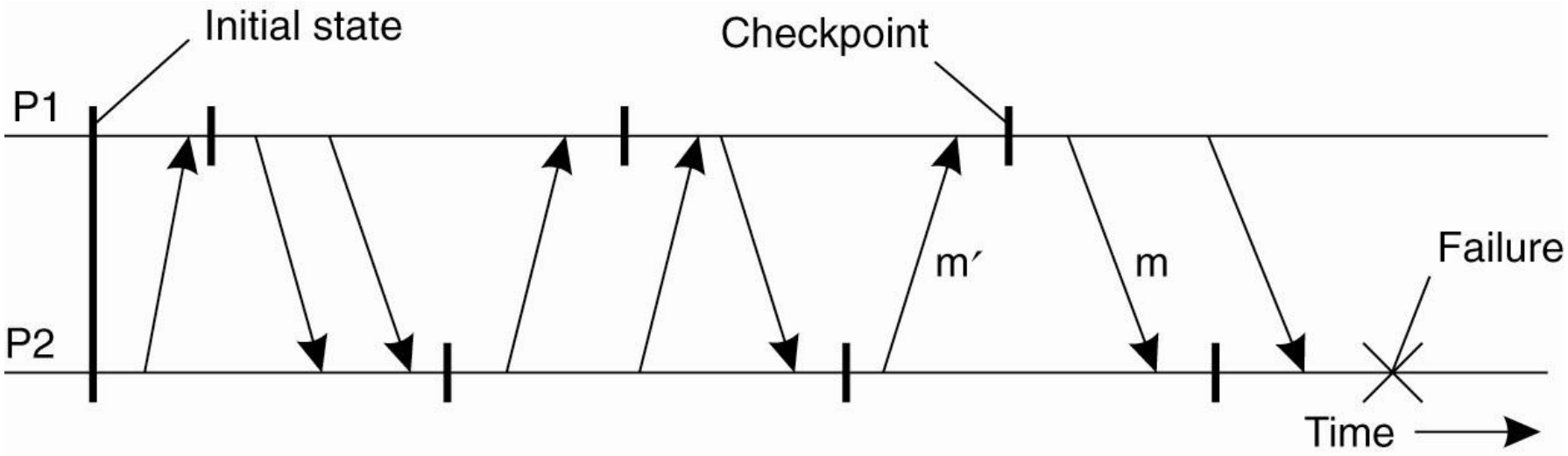
Recovery in Distributed Systems

- Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a **consistent state** from where to recover
- For this, each process saves its state time to time to a local stable storage (called **checkpoint**)
- In case of failure, get the most recent **consistent global state** or **recovery line**
  - If P has recorded the receipt of a msg, then there should be Q recorded the sending of this msg



Independent Checkpointing

- Each process independently takes snapshot!
- Easy, but it might be hard to find a recovery line
- Cascaded rollback may lead to domino effect
  - If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time



Independent Checkpointing

- Each process independently takes checkpoints
  - Let $CP[i](m)$ denote m^{th} checkpoint of process Pi and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
  - When process Pi sends a message in interval $INT[i](m)$, it piggybacks (i,m)
  - When process Pj receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
  - The dependency $INT[i](m) \rightarrow INT\ [j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$
- If process Pi rolls back to $CP[i](m)$, Pj must roll back to $CP[j](n)$.
- **Risk**: cascaded rollback to system startup

Coordinated Checkpointing

- Each process takes a checkpoint after a **globally coordinated** action
- **Simple solution:** two-phase blocking protocol
  - A coordinator multicasts a *checkpoint request* message
  - When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
  - When all checkpoints have been confirmed at the coordinator, the latter broadcasts a *checkpoint done* message to allow all processes to continue
- **Observation:** consider processes that depend on coordinator, and ignore the rest → **incremental**

Message Logging

- Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint
  - store messages in a **log** ⇒ **replay** your (communication) behavior from the most recent checkpoint
- **Assumption:**

  Assume a **piecewise deterministic** execution model:
  - The execution of each process can be considered as a sequence of state intervals
  - Each state interval starts with a nondeterministic event (e.g., message receipt)
  - Execution in a state interval is deterministic

  If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

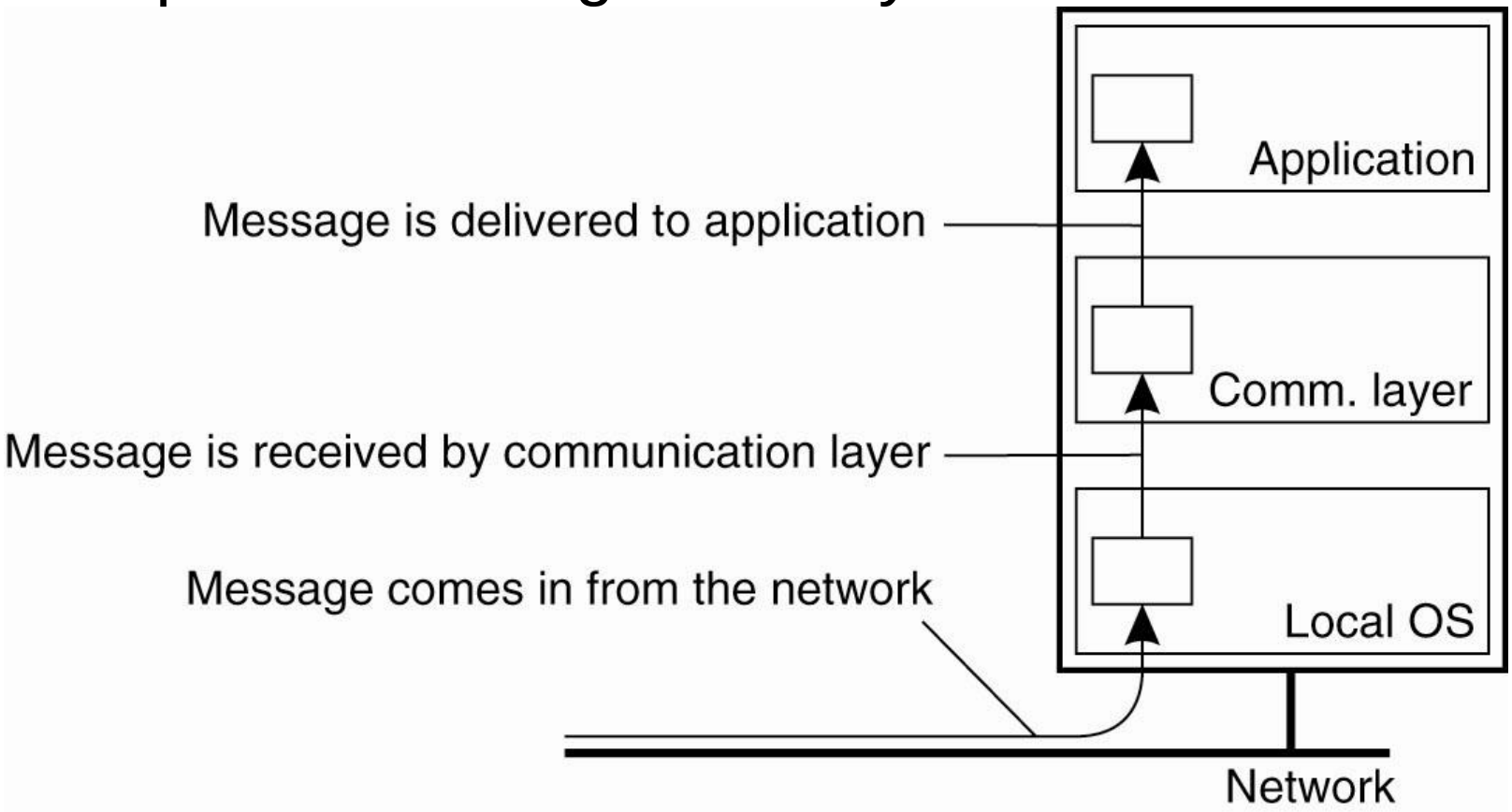
EXTRAS

Summary

- Terminology: fault, error and failures
- Fault management and failure models
- Fault tolerance (agreement) with redundancy
  - Level of redundancy vs. failure models
- Fault recovery techniques
- Checkpointing and stable storage
- Recovery in distributed systems:
  - Consistent checkpointing
  - Message logging

Virtual Synchrony (1)

- The logical organization of a distributed system to distinguish between message receipt and message delivery.

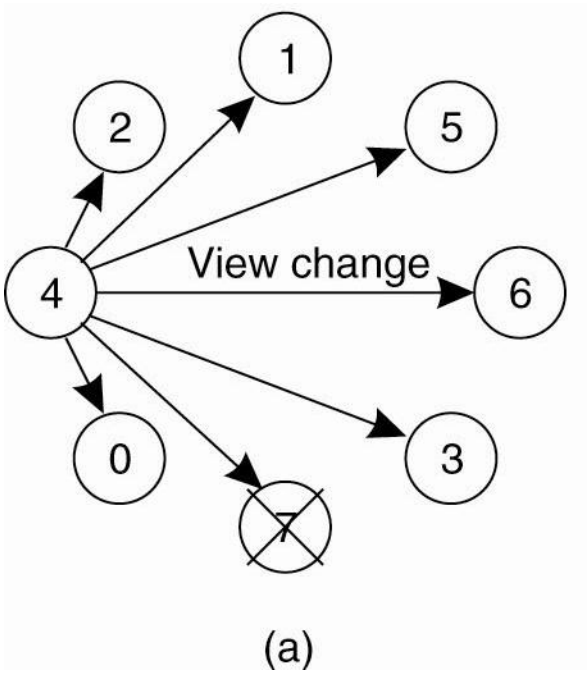


Implementing Virtual Synchrony (1)

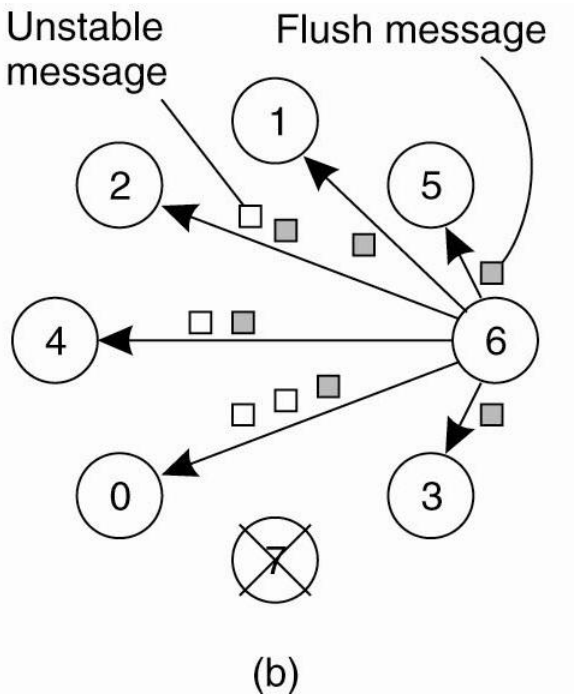
Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

- Six different versions of virtually synchronous reliable multicasting.

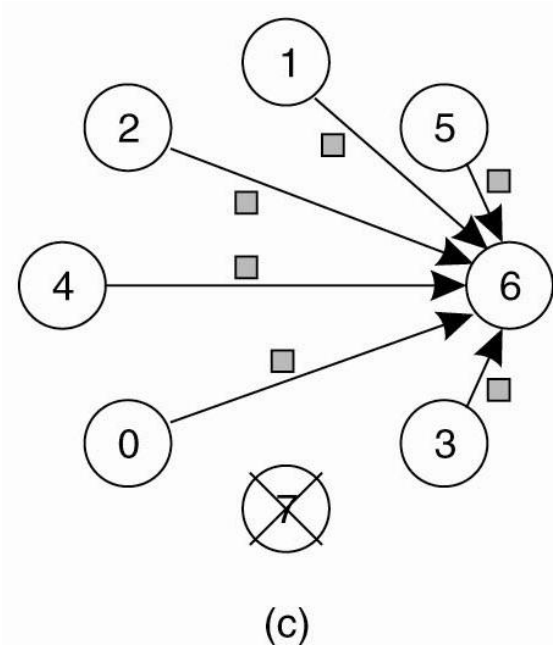
Implementing Virtual Synchrony (2)



(a)



(b)



(c)

- (a) Process 4 notices that process 7 has crashed and sends a view change.
- (b) Process 6 sends out all its unstable messages, followed by a flush message.
- (c) Process 6 installs the new view when it has received a flush message from everyone else.

Message Ordering (1)

- Four different orderings are distinguished:
  - Unordered multicasts
  - FIFO-ordered multicasts
  - Causally-ordered multicasts
  - Totally-ordered multicasts

Message Ordering (2)

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

- Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

Message Ordering (3)

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

- Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

Message Logging Schemes

- HDR[m]: message m's header contains its source, destination, sequence number, and delivery number
  - A message *m* is **stable** if HDR[m] cannot be lost (e.g., because it has been written to stable storage)
  - The header contains all information for resending a message and delivering it in the correct order (assume data is reproduced by the application)
- DEP[m]: set of processes to which message *m* has been delivered, as well as any message that causally depends on delivery of *m*
- COPY[m]: set of processes that have a copy of HDR[m] in their volatile memory

Message Logging Schemes (cont.)

- **Orphan**: If *C* is a collection of crashed processes, then $Q \notin C$ is an orphan if there is a message *m* such that $Q \in DEP[m]$ and $COPY[m] \subseteq C$
- If for each message *m*, $DEP[m] \subseteq COPY[m]$ → no orphans;

- **Pessimistic protocol:** for each *non-stable* message *m*, there is at most one process dependent on *m*, that is $|DEP[m]| \leq 1$
  - An unstable message in a pessimistic protocol *must* be made stable before sending a next message

Message Logging Schemes (cont.)

- **Optimistic protocol:** for each unstable message *m*, we ensure that if COPY[m] $\subseteq C$, then eventually also DEP[m] $\subseteq C$, where C denotes a set of processes that have been marked as faulty;
  - To guarantee that DEP[m] $\subseteq C$, we generally rollback each orphan process Q until $Q \notin$ DEP[m]
  - More complicated to implement