Chapter 8

Fault Tolerance

November, 26th

Introduction

Dependable systems

Faults

- Systems that can tolerate faults (fault tolerant)
  - Provide a service despite faults
- **Transient** faults
  - Occur once and disappear
- **Intermittent** faults
  - Occur, then vanish, then occur again, and so on
- **Permanent** faults
  - Continue to exist until the faulty component is replaced

Dependable systems

Metrics

- **Availability**
  - A highly available system is mostly likely working at any given time
- **Reliability**
  - A highly reliable system will most likely continue to work
- **Safety**
  - A safe system does not do anything wrong (although it may stop)
- **Maintainability**
  - How easily a failed system can be repaired

Dependable systems

Availability vs Reliability

- A system that goes down for one millisecond every hour
  - Highly available: up 99.9999% of the time
  - Unreliable: the system crashes often
- A system that never crashes but down for two weeks every year
  - Low availability: up 96% of the time
  - Reliable: the system almost never crashes

Failure models

- **Crash failure**
  - A server halts but was working properly until it stopped
  - For example, an operating system that hangs
- **Omission failure**
  - Receive omission: a server does not receive a request
  - Send omission: server has done some work but cannot send a message (e.g., transmission buffer overflow)
- **Timing failure**
  - Response lies outside a specified real-time interval

Failure models

- **Response failure**
  - The server's response is incorrect (e.g., a search engine that returns Web pages unrelated to search terms)
- **State transition failure**
  - A request that makes the server react unexpectedly
  - For example, the server receives a request it cannot recognize
- **Arbitrary failure or Byzantine failure**
  - Most difficult failures to handle
  - Malicious server (hacked server)
  - Software bugs
    - silently omitted in recent years
    - originally tackled via n-version programming
    - until n-version programming was shown to fail in practice

Failure models

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

Different types of failures.

Failure masking by redundancy

Redundancy is the best way to hide failures:

- **Information** redundancy
  - Extra bits added to allow recovery (e.g., Hamming code)
- **Time** redundancy
  - An action is performed multiple times (e.g., transactions)
- **Physical** redundancy
  - Extra equipment or processes are added to tolerate failures
  - Possible in hardware (e.g., disk) and software (e.g., process)

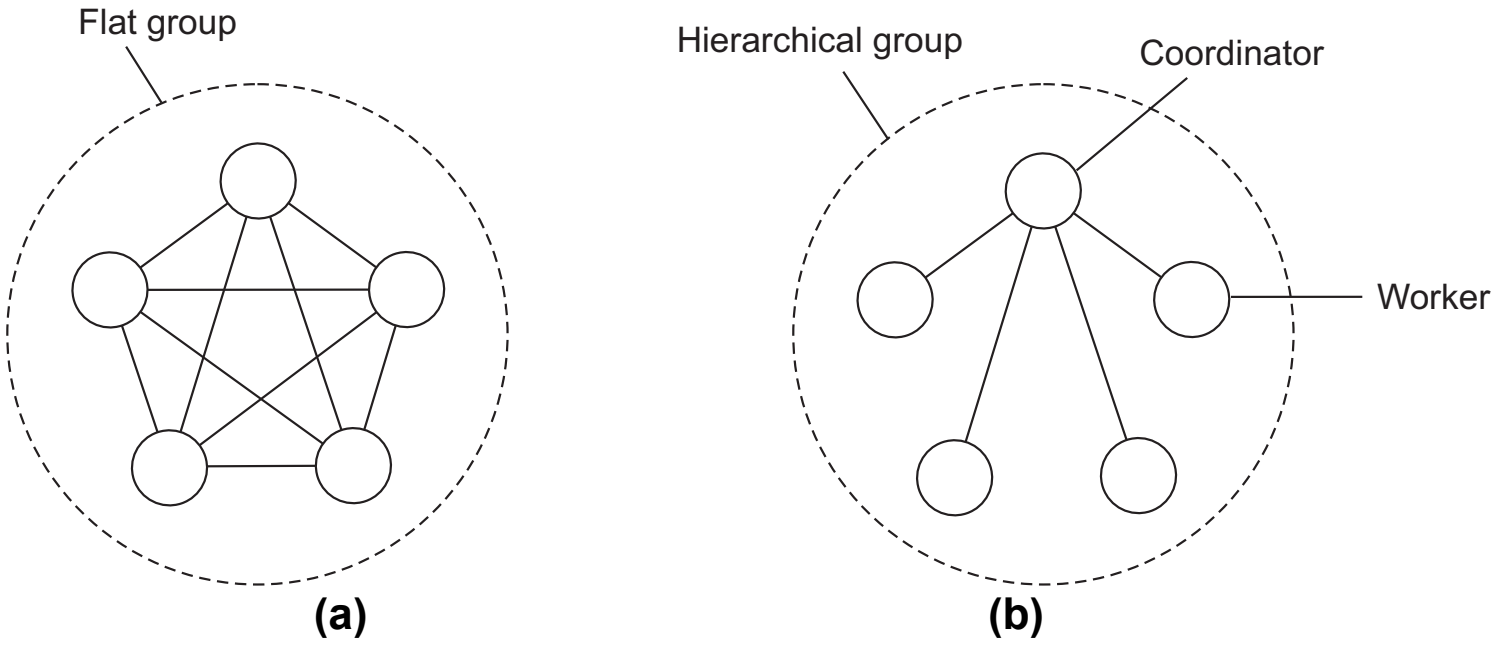
Process resilience

Design issues

- Key idea is organize identical processes into a group
- When a message is sent to the group, all processes receive it
- Abstracts collection of processes as a single entity
- Dynamic groups
  - A process can join and leave a group
  - A process can be the member of several groups at the same time

Flat groups versus hierarchical groups

- Internal structure of a group
- Defines how work is done by group members



(a) Communication in a flat group.
(b) Communication in a simple hierarchical group.

Group membership

How to create and delete groups, add/remove members?

- **Group server**: easiest approach but not fault tolerant
- **Distributed group membership**:
  - Group state at each group member
  - Mechanism needed to detect member failures

Failure masking and replication

How to organize processes inside a group?

- Primary-based replication
- Active replication

How many replicas in a k-fault tolerant system?

- System can tolerate up to k failures
- Silent failures: $k + 1$ replicas are enough
- Byzantine failures: $2k + 1$
- In asynchronous systems $2k + 1$ and $3k + 1$ respectively

Agreement in faulty systems

- Agreement is a fundamental problem in distributed systems, e.g.
  - electing a coordinator
  - deciding whether to commit or not a transaction
  - totally ordering messages
- Easy to implement under “ideal conditions”
  - Absence of process failures
  - Absence of message losses

Consensus in faulty systems

Prerequisite

In a fault-tolerant process group, each nonfaulty process executes the same commands, and in the same order, as every other nonfaulty process

Reformulation

Nonfaulty group members need to reach consensus on which command to execute next

Flooding-based consensus

System model:

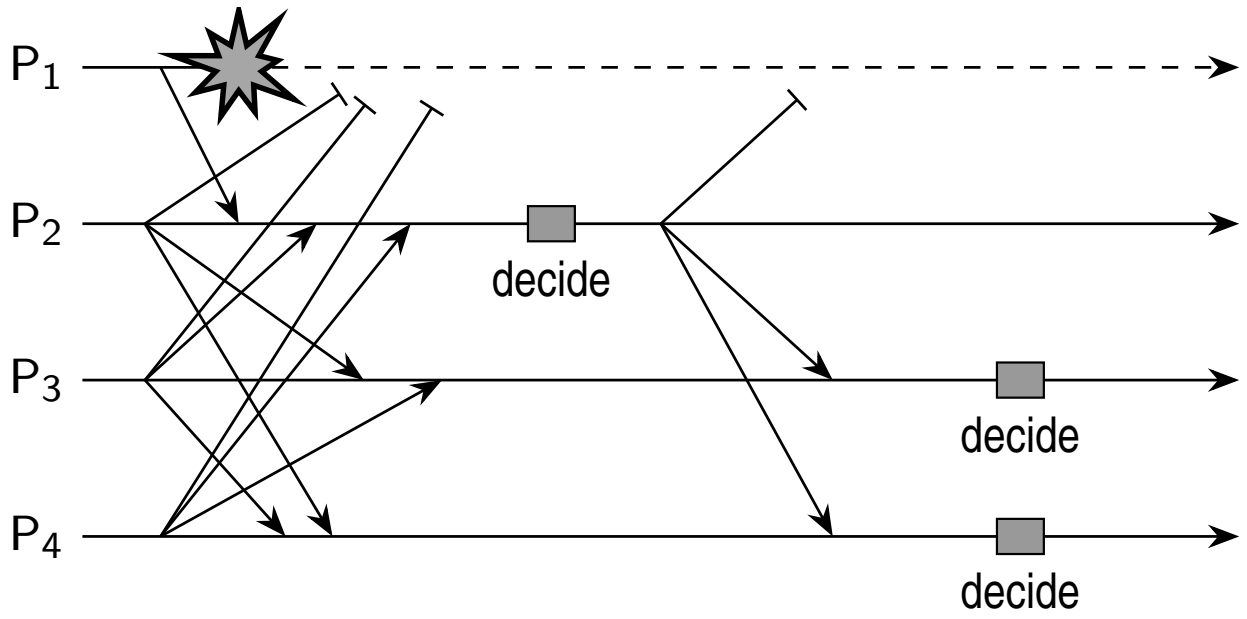
- A process group $P = \{P_1, ..., P_n\}$
- Fail-*stop* failure semantics, i.e., with reliable failure detection
- A client contacts a P_i requesting it to execute a command
- Every P_i maintains a list of proposed commands

Basic algorithm (based on rounds)

- In round r, P_i multicasts its known set of commands $C_{r,i}$ to all others
- At the end of r, each P_i merges all received commands into a new $C_{r+1,i}$
- Next command $\texttt{cmd}_i$ selected through a globally shared, deterministic function: $\texttt{cmd}_i \leftarrow \texttt{select}(C_{r+1,i})$

Flooding-based consensus

Example



Failure detection

- Fundamental in many fault-tolerant distributed systems
- Basic mechanism
  - Message exchanges (heartbeats)
  - Timeouts (due to no message reception)
- How to handle failure detection in large systems?
- Network failures versus process failures

Reliable client-server communication

Point-to-point communication

- How to mask omission, timing, and arbitrary failures?
- TCP masks omission failures
  - Lost messages handled with acknowledgments
  - What about crash failures of connections?

RPC semantics in the presence of failures

Different classes of failures that can occur in RPC systems:

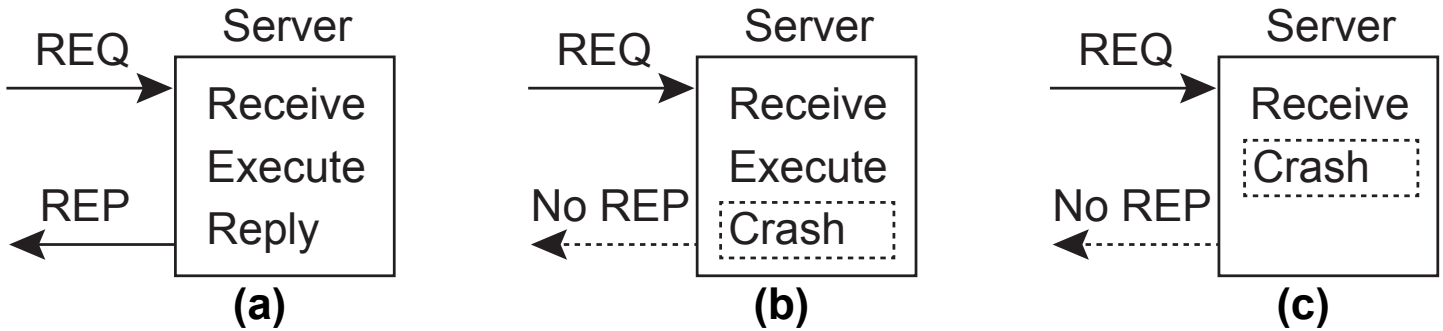
- The client is unable to locate the server
- The request message from the client to the server is lost
- The server crashes after receiving a request
- The reply message from the server to the client is lost
- The client crashes after sending a request

RPC semantics in the presence of failures

- The client is unable to locate the server
  - All servers are down or client and server don’t agree on interface
  - Possibly handled by an exception at client, but...
  - No longer fully transparent
- The request message from the client to the server is lost
  - Client stub starts a timer and retransmits if no response
  - Server must handle retransmissions to avoid duplicate work

RPC semantics in the presence of failures

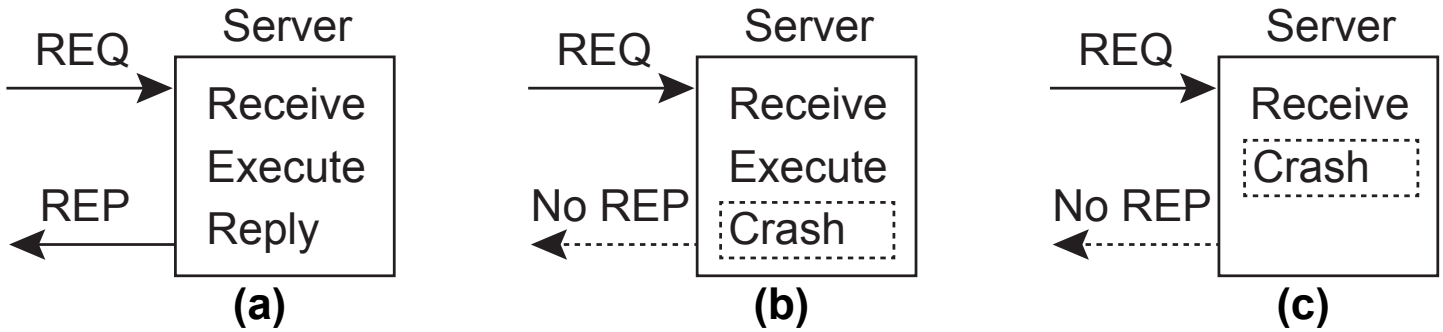
- The server crashes after receiving a request



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

RPC semantics in the presence of failures

- The server crashes after receiving a request



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

- Variations of RPC semantics:
  - **At-most-once semantics**:
    Give up immediately and report an error
  - **At-least-once semantics**:
    Keep trying until a response is received (e.g., after server reboot)
  - **Exactly-once semantics**:
    Ideal, but not always possible to implement

RPC semantics in the presence of failures

- The reply message from the server to the client is lost
  - resend request if no response is received after some time
  - the problem is that client does not know what the problem is...
  - not so much a problem with idempotent operations
  - but what about nonidempotent requests?

RPC semantics in the presence of failures

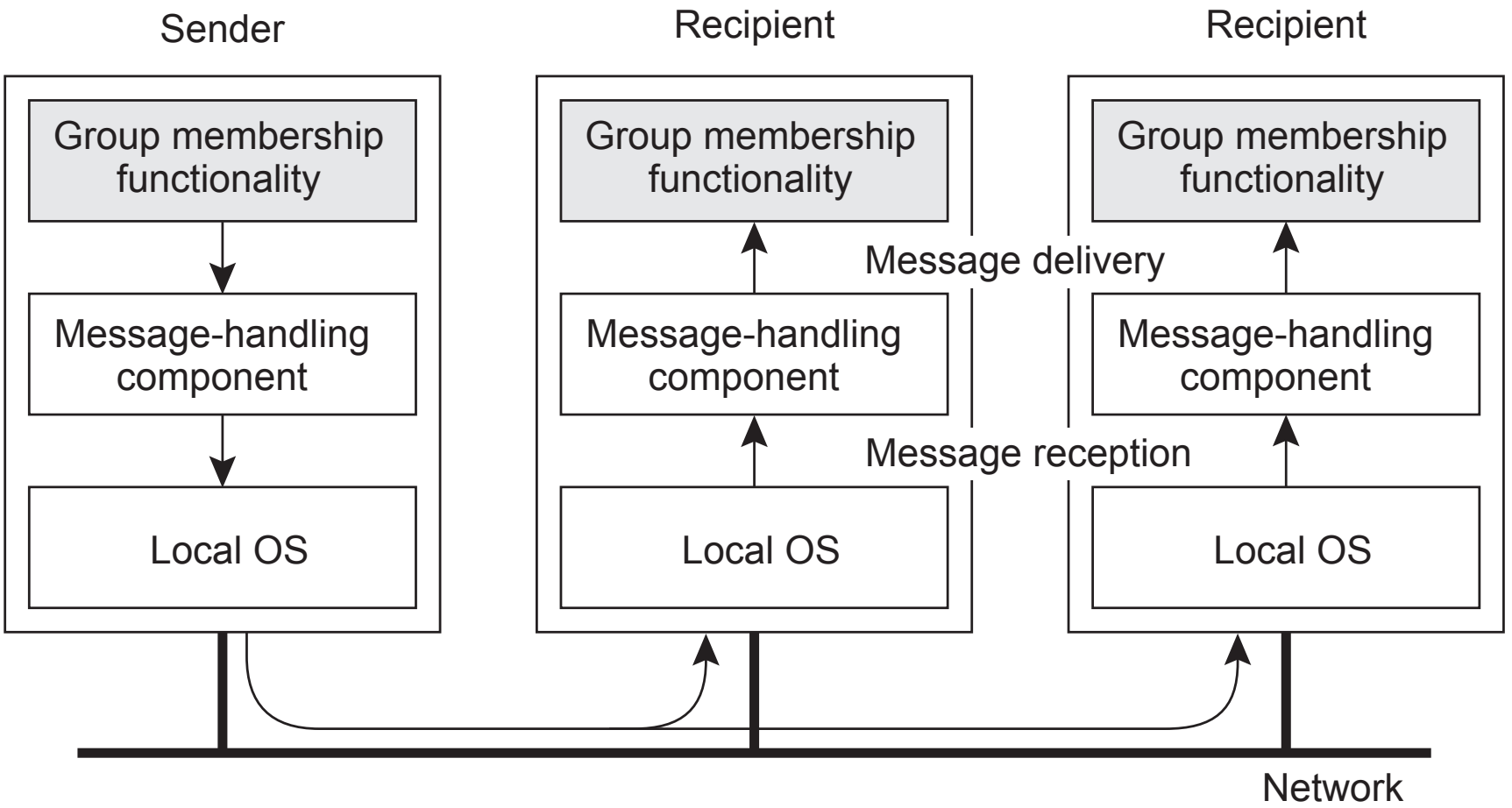
- The reply message from the server to the client is lost
  - resend request if no response is received after some time
  - the problem is that client does not know what the problem is...
  - not so much a problem with idempotent operations
  - but what about nonidempotent requests?
- The client crashes after sending a request
  - “Orphan” computation
  - Problematic for a few reasons
    - it may block resources (e.g., locked files)
    - the recovered client may do an RPC again and receive the answer of its previous call
  - Remediation:
    - **orphan extermination**: log before RPC to remove orphan on recovery... expensive
    - **reincarnation**: recovered client broadcasts new “epoch” and previous computations are killed
    - **expiration**: server checks with client and eliminates orphan computation

Reliable group communication

Basic reliable-multicasting schemes

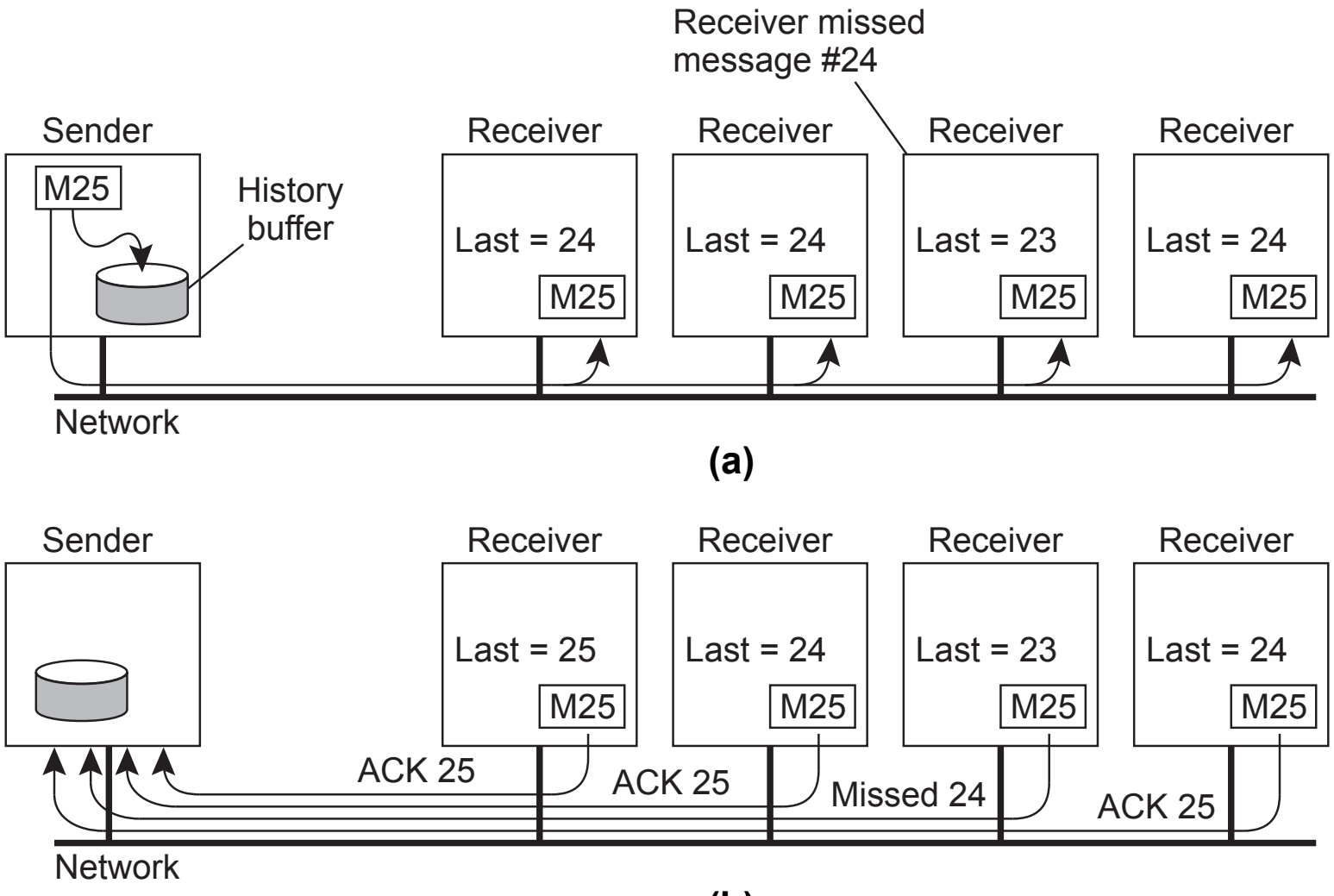
- Important mechanisms to implement replication
- Usually not provided by transport layers
- Intuitively, a message sent to a group must be received by all members of the group
- What happens if processes join the group during multicast?
- What happens if sending process fails during multicast?

Receiving versus delivering a message



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

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

- Problems with previous multicasting scheme
  - If there are N receivers, sender must send N messages
  - Sender may be swamped with ACKs
- Negative acknowledgments
  - Only sent by receivers when missing a message
  - No guarantee that **feedback implosion** will not happen
  - How to garbage collect the sender's history buffer?

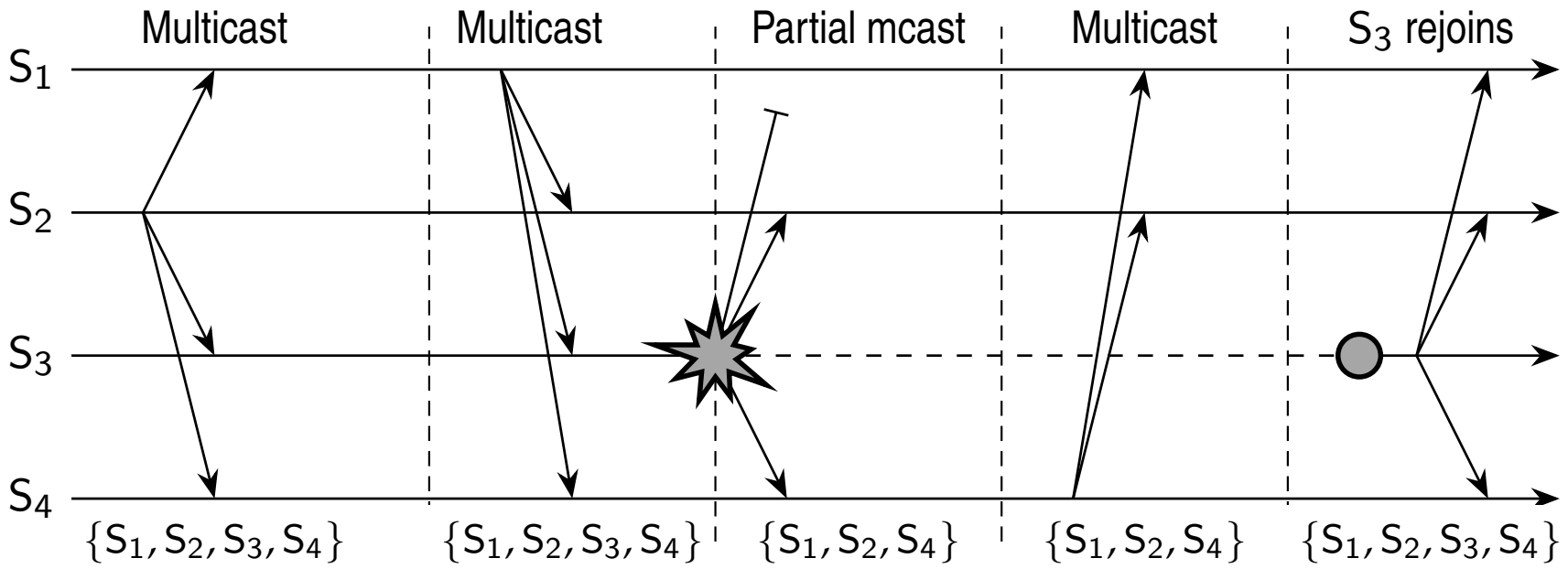
Atomic (total order) multicast

- The basis for state machine replication
- Messages are delivered to all non-faulty processes
- Messages are delivered in the **same order**
- Atomic multicast ensures that non-faulty processes maintain a consistent view of the system (within a group) and forces reconciliation when a replica recovers and rejoins the group

Virtual synchrony

- Process groups and group membership
  - Defining reliable multicast accurately in the presence of failures
- Processes in the same view must agree on which messages are delivered in that view
- Example
  - Message m is multicast to a group and at “the same time” a process joins the group, leading to a **view change** message vc
  - Virtual synchrony ensures that m and vc are delivered in the same order by all group members

Virtual synchrony



The principle of virtual synchronous multicast.

Message ordering

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

Message ordering

Reliable, unordered multicast

No guarantees are given concerning message ordering

Event order	Process P_1	Process P_2	Process P_3
1	sends m_1	receives m_1	receives m_2
2	sends m_2	receives m_2	receives m_1

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

Message ordering

Reliable, FIFO-ordered multicast

Incoming messages from the same process delivered in the order in which they were sent

Event order	Process P_1	Process P_2	Process P_3	Process P_4
1	sends m_1	receives m_1	receives m_3	sends m_3
2	sends m_2	receives m_3	receives m_1	sends m_4
3		receives m_2	receives m_2	
4		receives m_4	receives m_4	

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

Message ordering

Reliable causally- and totally-ordered multicast

- Reliable, causally-ordered multicast
  - Delivers messages so that potential causality between different messages is preserved
  - Usually broken down into two properties
    - *FIFO order*: as in FIFO-ordered multicast
    - *Local order*: if process delivers message m_1 before multicasting m_2, no process delivers m_2 before m_1

Message ordering

Reliable causally- and totally-ordered multicast

- Reliable, causally-ordered multicast
  - Delivers messages so that potential causality between different messages is preserved
  - Usually broken down into two properties
    - *FIFO order*: as in FIFO-ordered multicast
    - *Local order*: if process delivers message m_1 before multicasting m_2, no process delivers m_2 before m_1
- Reliable, totally ordered multicast
  - All messages are delivered in the same order

Message ordering

Multicast	Basic message ordering	TO 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.

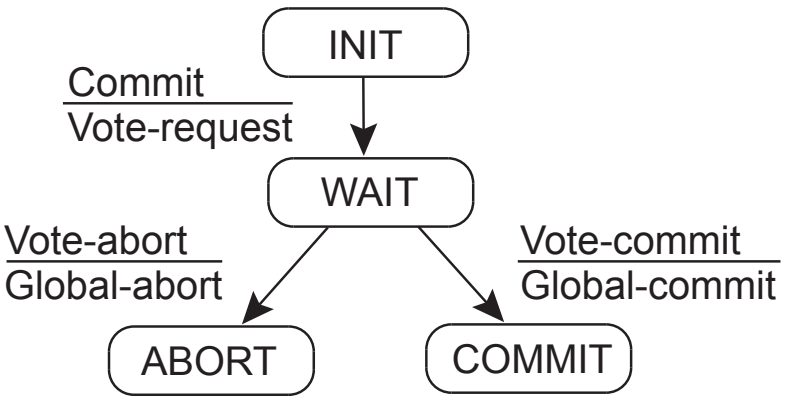
Distributed Commit

Introduction

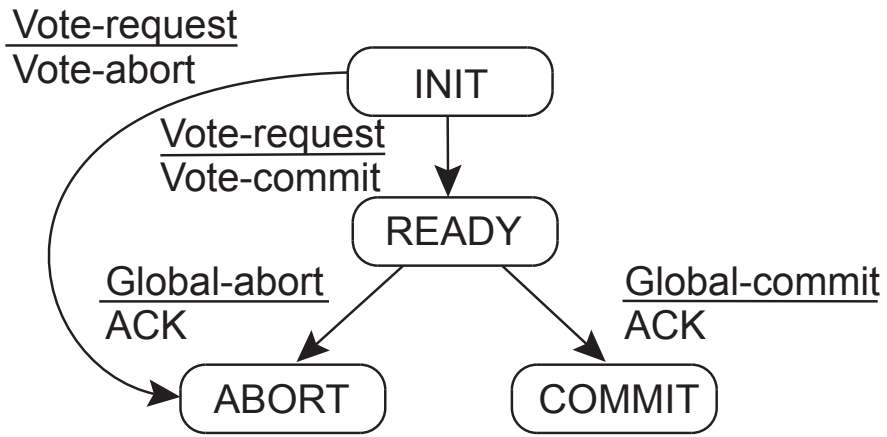
- How to ensure that an operation is performed by all members (participants) of a group or none at all?
- One-phase commit protocol does not work!
  - What happens if one participant cannot perform operation?
- More sophisticated protocols
  - Two-phase commit protocol (2PC)
  - Three-phase commit protocol (3PC)

Two-phase commit

- The coordinator sends `Vote-request` message to all participants
- When a participant receives `Vote-request`, it returns either a `Vote-commit` message (i.e., it is prepared to commit its part) or a `Vote-abort` message (otherwise)
- The coordinator collects all votes. If all voted to commit, it sends a `Global-commit` message to all; otherwise it sends a `Global-abort` message
- Each participant that voted to commit waits for the response from the coordinator and acts accordingly



(a)



(b)

Two-phase commit

In the presence of failures

- The protocol may block if the coordinator crashes
- To avoid blocking on participants, timeouts must be used to give up waiting for a message that will never arrive
- States in which a process is blocked waiting for a message
  - A participant in the `INIT` state waiting for a `Vote-request`
  - The coordinator in the `WAIT` state waiting for participant votes
  - A participant in the `READY` state waiting for the result

Two-phase commit

Two-phase commit is blocking

A participant will block until the coordinator recovers from a failure

- All participants have received and processed `Vote-request`
- The coordinator crashes before sending the result
- Participants cannot cooperatively decide on what to do
- Why???