Fault Tolerance

Dealing successfully with *partial failure* within a Distributed System.

Key technique: ***Redundancy***.

Basic Concepts

Fault Tolerance is closely related to the notion of "Dependability". In Distributed Systems, this is characterized under a number of headings:

- *Availability* – the system is ready to be used immediately.
- *Reliability* – the system can run continuously without failure.
- *Safety* – if a system fails, nothing catastrophic will happen.
- *Maintainability* – when a system fails, it can be repaired easily and quickly (and, sometimes, without its users noticing the failure).

But, What Is “Failure”?

Definition:

- A system is said to “fail” when it *cannot meet* its promises.
- A failure is brought about by the *existence* of “errors” in the system.
- The *cause* of an error is a “fault”.

Types of Fault

There are 3 main types of ‘fault’:

- *Transient Fault* – appears once, then disappears.
- *Intermittent Fault* – occurs, vanishes, reappears; but: follows no real pattern (worst kind).
- *Permanent Fault* – once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally.

Classification of Failure Models

Different types of failures, with brief descriptions.

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 outgoing messages.
Timing failure	A server's response lies outside the specified time interval.
Response failure *Value failure* *State transition failure*	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

Strategy: hide the occurrence of failure from other processes using ***redundancy***:

- *Information Redundancy* – add extra bits to allow for error detection/recovery (e.g., Hamming codes).
- *Time Redundancy* – perform operation and, if needs be, perform it again (e.g. how transactions work - BEGIN/END/COMMIT/ABORT).
- *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

DS Fault Tolerance Topics

1. Process Resilience
2. Reliable Client/Server Communications
3. Reliable Group Communciation
4. Distributed COMMIT
5. Recovery Strategies

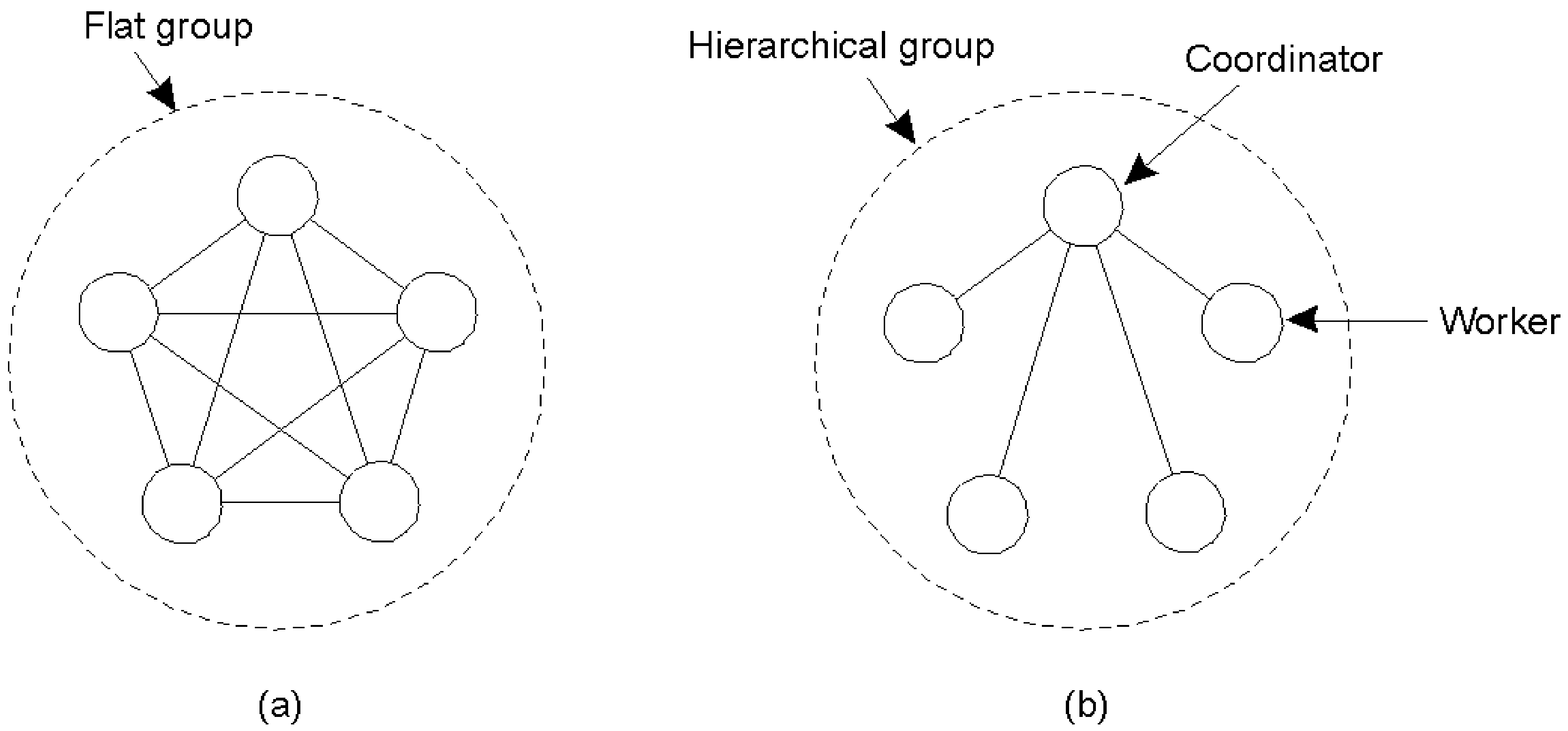
1. Process Resilience

- Processes can be made fault tolerant by arranging to have a **group of processes**, with each member of the group being *identical*.

- A message sent to the group is delivered to all of the “copies” of the process (group members), and then *only one* of them performs the required service.

- If one of the processes fails, it is assumed that one of the others will still be able to function (and service any pending request or operation).

Flat vs. Hierarchical Groups



(a) Communication in a flat group: all processes are equal, decisions are made collectively.

NB: no single point-of-failure; however, decision making is complicated as consensus is required.

(b) Communication in a simple hierarchical group: one of processes is elected coordinator, which selects another process (a worker) to perform the operation.

NB: single point-of failure; however, decisions are easily and quickly made by the coordinator without first having to get consensus.

Failure Masking and Replication

- By organizing a *fault tolerant group of processes*, we can protect a single vulnerable process.

- There are two approaches for arranging the replication of the group:

1. Primary (backup) Protocols.
2. Replicated-Write Protocols.

Goal of Agreement Algorithms

- To have all *non-faulty* processed reach concensus on some issue (quickly).
- The **two-army problem**

Even with non-faulty processes, agreement between even two processes is not possible in the face of **unreliable communication**.

History Lesson: The Byzantine Empire

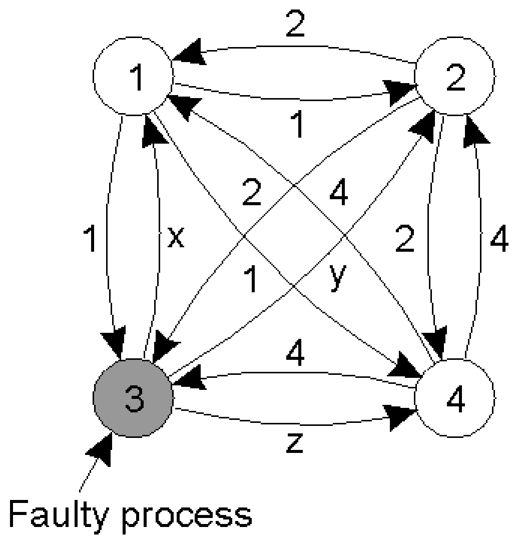
- *Time*: 330-1453 AD
- *Place*: Balkans (Modern Turkey)

- *Scenario*: endless conspiracies, intrigue, and untruthfullness were alleged to be common practice in the ruling circles of the day (*sounds strangely familiar…*).
  - Typical for intentionally wrong and malicious activity to occur among the ruling group.
  - Similar occurrence can surface in a DS - known as '**byzantine failure**'.

- *Question*: how do we deal with such **malicious group members** within a distributed system?

Agreement in Faulty Systems (1)

How does a process group deal with a **faulty member**?



(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 and 1 traitor.

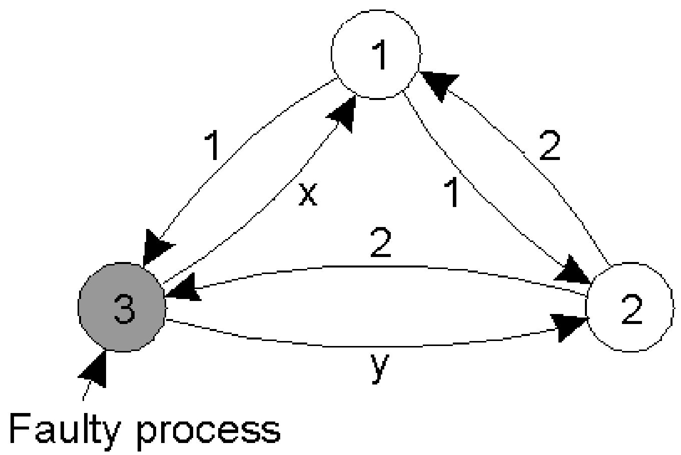
a) The generals **announce** their troop **strengths** (in units of 1 kilosoldiers) to the other members of the group by sending a message.

b) The vectors that each general assembles based on (a), each general knows their own strength. They then **send** their **vectors** to all the other generals.

c) The vectors that each general receives in step 3. It is clear to all that General 3 is the traitor. In each ‘column’, the **majority** value is assumed to be correct.

Agreement in Faulty Systems (2)

Warning: the algorithm does not always work!



(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)

- The same algorithm as in previous slide, except now with 2 loyal generals and 1 traitor. NB: It is **no** longer possible to determine the **majority** value in each column, and the algorithm has failed to produce agreement.
- It has been shown that for the algorithm to work properly, ***more*** than **2/3** of the processes have to be working correctly. That is: if there are **M faulty** processes, we need **2M + 1** functioning processes to reach **agreement**.

2 .Reliable C/S Communications

- A communication channel may also exhibit crash, omission, timing, and/or arbitrary failures.
- In practice, the focus is on masking *crash* and *omission* failures.
  - ***e.g.*** point-to-point TCP masks omission failures by guarding against lost messages using ACKs and retransmissions. However, it performs poorly when a crash occurs (although a DS may try to mask a TCP crash by automatically re-establishing the lost connection).

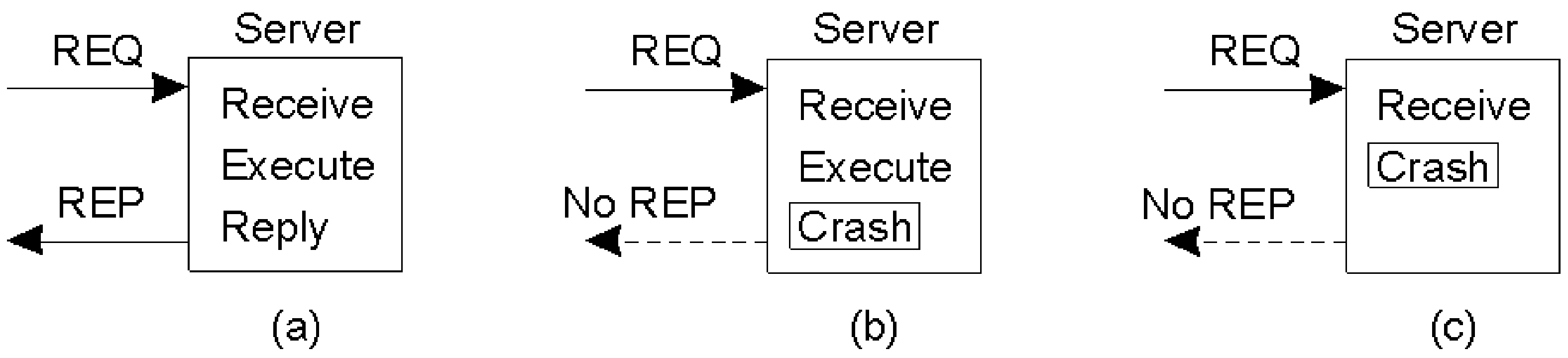
Example: RPC Semantics and Failures

- The RPC mechanism works well as long as both the client and server function perfectly

- Five classes of RPC failure can be identified:

1. *Client cannot locate the server*, so no request can be sent.
2. *Client's request to the server is lost*, so no response is returned by the server to the waiting client.
3. *Server crashes after receiving the request*, and the service request is left acknowledged, but undone.
4. *Server's reply is lost on its way to the client*, the service has completed, but the results never arrive at the client
5. *Client crashes after sending its request*, and the server sends a reply to a newly-restarted client that may not be expecting it.

The Five Classes of Failure (1)



- A server in client-server communication:

a) The normal case.

b) Crash *after* service execution.

c) Crash *before* service execution.

The Five Classes of Failure (2)

- An appropriate **exception handling** mechanism can deal with a missing server.
  - However, such technologies tend to be language-specific and non-transparent (which is a big DS ‘no-no’).

- Dealing with lost request messages can be dealt with easily using **timeouts**.
  - If no ACK arrives in time, the message is resent.
  - Of course, the server needs to be able to deal with the possibility of duplicate requests.

The Five Classes of Failure (3)

- Server crashes are dealt with by implementing one of three possible implementation philosophies:
  - ***At least once semantics***: a guarantee is given that the RPC occurred at least once, but (also) possibly more than once.
  - ***At most once semantics***: a guarantee is given that the RPC occurred at most once, but possibly not at all.
  - ***No semantics***: nothing is guaranteed, and client and servers take their chances!

- It has proved difficult to provide *exactly once semantics*.

The Five Classes of Failure (4)

- Lost replies are difficult to deal with.
  - *Why* was there no reply?
  - Is the server *dead*, *slow*, or did the reply just go *missing*?

- A request that can be repeated any number of times without any nasty side-effects is said to be *idempotent* (e.g. a read of a static web-page is said to be idempotent).

- *Nonidempotent* requests (e.g. electronic transfer of funds) are a little harder to deal with.
  - Common solution is to employ *unique sequence numbers*.
  - Another technique is the inclusion of additional bits in a retransmission to identify it as such to the server.

The Five Classes of Failure (5)

- When a client crashes, and when an ‘old’ reply arrives, such a reply is known as an *orphan*.

- Four orphan solutions have been proposed:
  - *extermination* (the orphan is simply killed-off),
  - *reincarnation* (each client session has an *epoch* associated with it, making orphans easy to spot),
  - *gentle reincarnation* (when a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed), and,
  - *expiration* (if the RPC cannot be completed within a stardard amount of time, it is assumed to have expired).

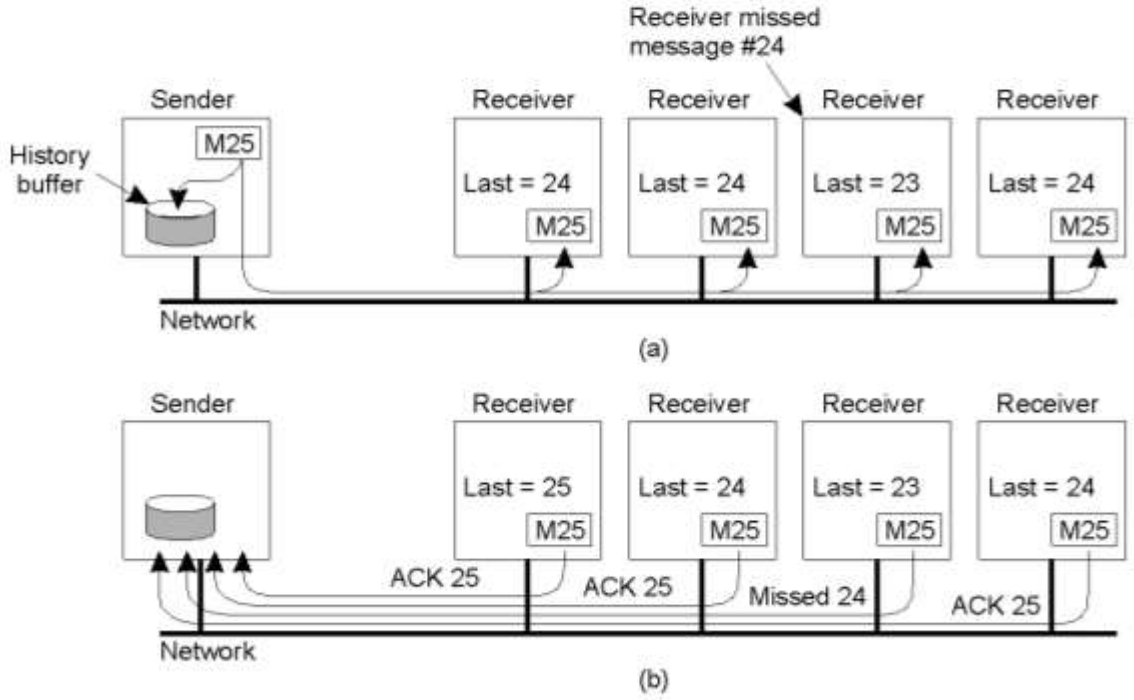
- In practice, however, none of these methods are desirable for dealing with orphans. Research continues…

3. Reliable Group Communication

- Reliable multicast services guarantee that all messages are delivered to all members of a process group.
  - Sounds simple, but is surprisingly *tricky* (multicasting services tend to be *inherently* unreliable).

- For a small group, multiple and reliable point-to-point channels will do the job;
  - However, such a solution *scales poorly* as the group membership grows.
  - What happens if a process *joins* the group during communication?
  - Worse, what happens if the sender of the multiple, reliable point-to-point channels *crashes* half way through sending the messages?

Basic Reliable-Multicasting Schemes



- **Sequencer**: simple solution to reliable multicasting when *all receivers are known* and are assumed *not to fail*. The sending process assigns a **sequence number** to outgoing messages (making it easy to spot when message missing).

  (a) Message transmission: note that the third receiver is expecting 24.

  (b) Reporting feedback: the third receiver informs the sender.
- But, how long does the sender keep its *history-buffer* populated?
- Also, such schemes **perform poorly** as the group grows… *too many* ACKs.

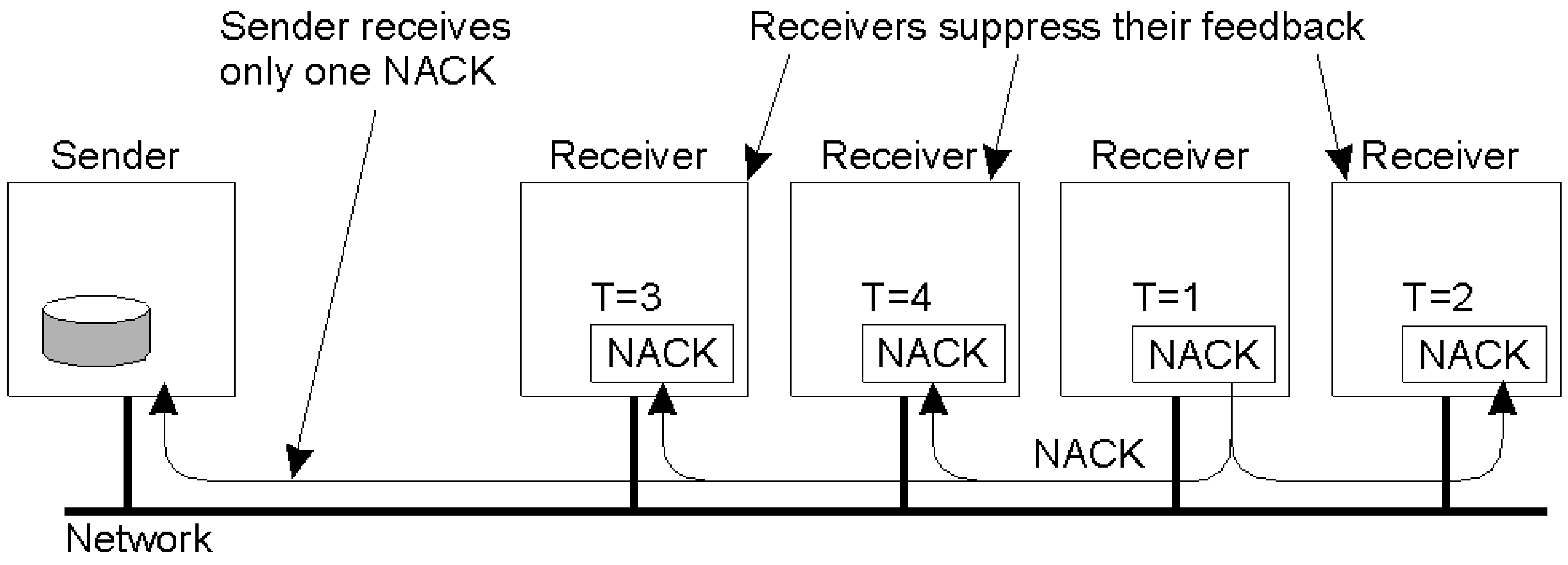
SRM: Scalable Reliable Multicasting

- Receivers *never* ACKs successful delivery.
  - **Only missing messages are reported.**
  - NACKs are multicast to all group members.

- This allows other members to suppress their feedback, if necessary.

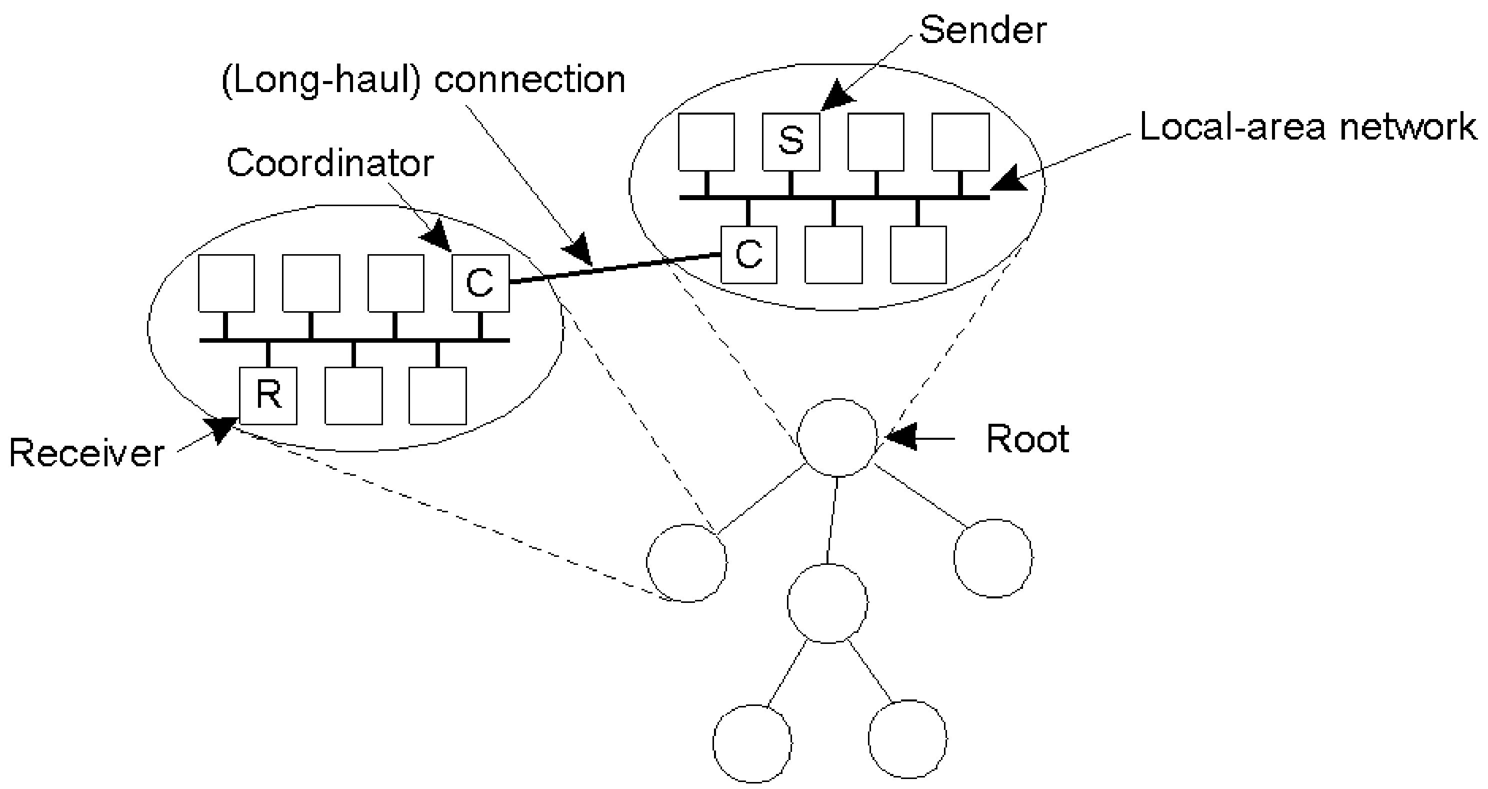
- To avoid “retransmission clashes”, each member is required to wait a random delay prior to NACKing.

Nonhierarchical Feedback Control



- ***Feedback Suppression***: reduce number of feedback messages to the sender (cf. *Scalable Reliable Multicasting Protocol*).
- Successful delivery is never ACK, only missing messages are reported (NACK), which are multicast to all group members. If another process is about to NACK, this feedback is suppressed as a result of the first multicast NACK (only a **single** NACK is delivered to the sender).

Hierarchical Feedback Control



- **Hierarchical** reliable multicasting is another solution, the main characteristic being that it supports the creation of **very large groups**.
- **Sub-groups** within the entire group are created, with each *local coordinator* forwarding messages to its children.
- A **local coordinator** handles retransmission requests *locally*, using any appropriate multicasting method for small groups.

Atomic Multicasting

- There often exists a requirement where the system needs to ensure that **all processes** get the message, **or none** of them get it.

- An additional requirement is that all messages arrive at all processes in **sequential order**.

- This is known as "**atomic multicast problem**".

4. Distributed COMMIT

General Goal:

- *We want an operation to be performed by all group members, or none at all.*
  - In case of atomic multicasting, the operation is the delivery of the message.

- There are 3 types of “commit protocol”:
  - single-phase
  - two-phase
  - three-phase commit.

Commit Protocols

- **One-Phase Commit Protocol**:
  - An elected co-ordinator tells all the other processes to perform the operation in question.
  - But, what if a process cannot perform the operation?
    - There is no way to tell the coordinator!

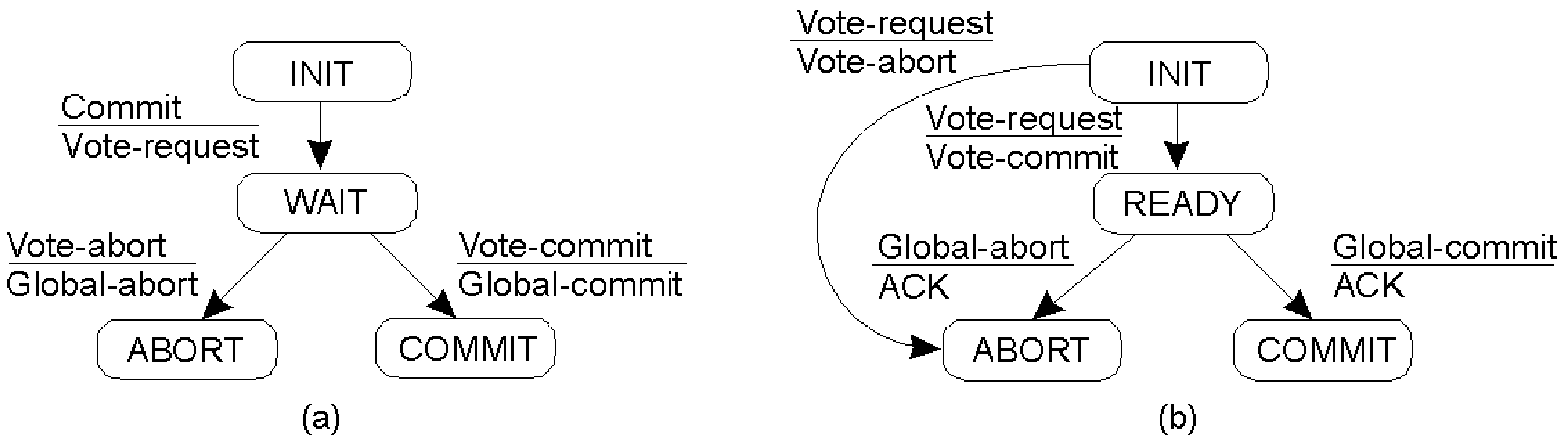
- **The solutions**:
  - *Two-Phase* and *Three-Phase Commit Protocols*.

The Two-Phase Commit Protocol

- First developed in 1978!
- *Summarized:*
  - *GET READY, OK, GO AHEAD*

1. The coordinator sends a *VOTE_REQUEST* message to all group members.
2. A group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
3. All votes are collected by the coordinator
   - *GLOBAL_COMMIT* is sent if all group members voted to commit
   - *GLOBAL_ABORT* is sent if one group member voted to abort
4. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

Two-Phase Commit Finite State Machines



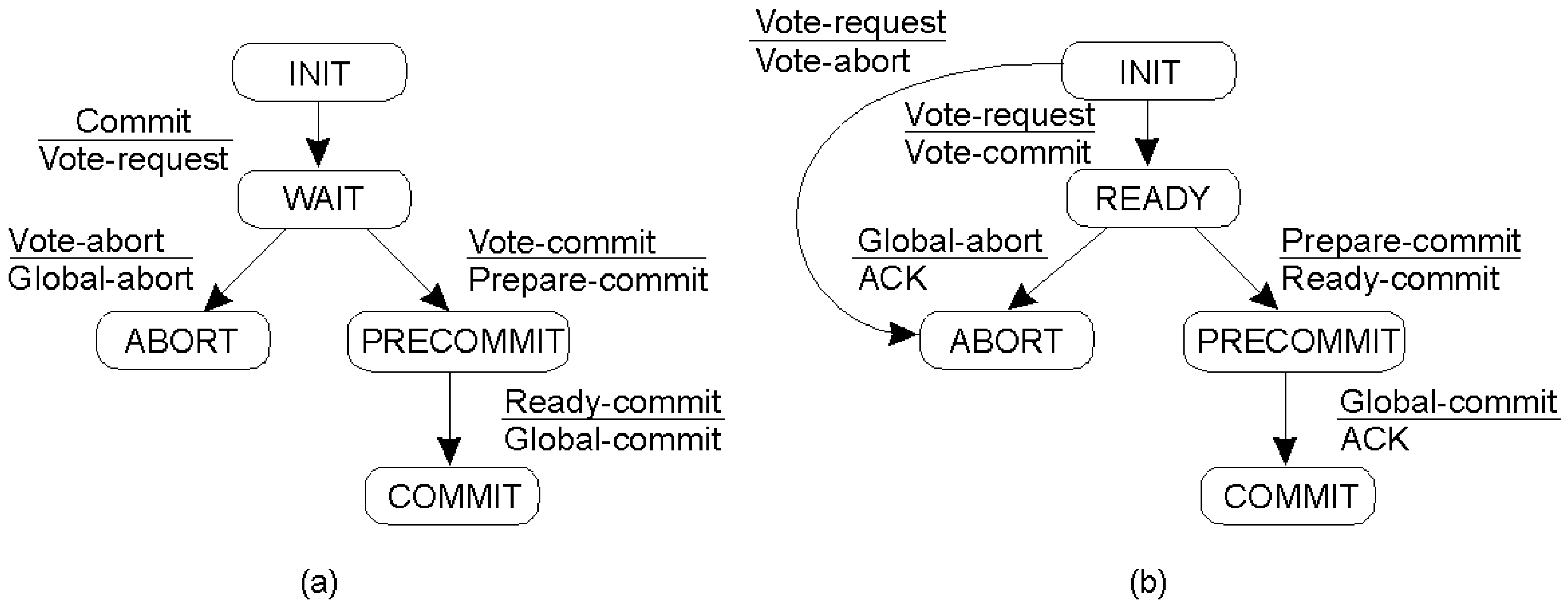
(a) Finite state machine for the **coordinator**.

(b) Finite state machine for a **participant** (group member).

Big Problem with Two-Phase Commit

- Can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded ***deadlock***.
  - If coordinator crashes, group members may not be able to *reach a final decision*,
    - and they may, therefore, block until the coordinator *recovers*
- Two-Phase Commit is known as a **blocking-commit protocol** for this reason
- The solution?
  - Timeouts
  - *Three-Phase Commit Protocol.*

Three-Phase Commit



(a) Finite state machine for coordinator.

(b) Finite state machine for group member.

- **Main point**: although 3PC is generally regarded as *better* than 2PC, it is *not applied often in practice*, as the conditions under which 2PC blocks rarely occur.

5. Recovery Strategies

- Once a failure has occurred, it is essential that the process where the failure happened *recovers* to a correct state.

- Recovery from an error is *fundamental* to fault tolerance.

- Two main forms of recovery:

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.
2. **Forward Recovery**: bring the system into a correct state, from which it can then continue to execute.

Forward and Backward Recovery

- **Disadvantage of Backward Recovery**:
  - Checkpointing (can be very expensive, especially when errors are very rare).
  - Despite the cost, backward recovery is implemented more often ("logging" of information can be thought of as a type of checkpointing).

- **Disadvantage of Forward Recovery**:
  - All potential errors need to be accounted for *up-front*.
  - When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

Recovery Example

- **Consider, for example, Reliable Communications.**
    - *Retransmission* of a lost/damaged packet is an example of a backward recovery technique.
    - When a lost/damaged packet can be reconstructed as a result of the receipt of other successfully delivered packets, then this is known as *Erasure Correction* (an example of forward recovery technique).

Summary (1 of 2)

- Fault Tolerance:
  - *Characteristic by which a system can mask the occurrence and recovery from failures.*
  - *A system is fault tolerant if it can continue to operate even in the presence of failures.*

- Types of failure:
  - *Crash* (system halts);
  - *Omission* (incoming request ignored);
  - *Timing* (responding too soon or too late);
  - *Response* (getting the order wrong);
  - *Arbitrary/Byzantine* (indeterminate, unpredictable).

Summary (2 of 2)

- Fault Tolerance is generally achieved through use of *redundancy* and *reliable multitasking protocols*.
- Processes, client/server and group communications can all be "enhanced" to tolerate faults in a DS.
- Commit protocols allow for fault tolerant multicasting (with *two-phase* the most popular type).
- *Recovery* from errors within a DS tends to rely heavily on **Backward Recovery** techniques that employ some type of *checkpointing* or *logging* mechanism (although **Forward Recovery** is also possible).