

ICS 362 Distributed Systems

Distributed Systems: Part 16

Lecturer: Toby Daniel

Fault Tolerance

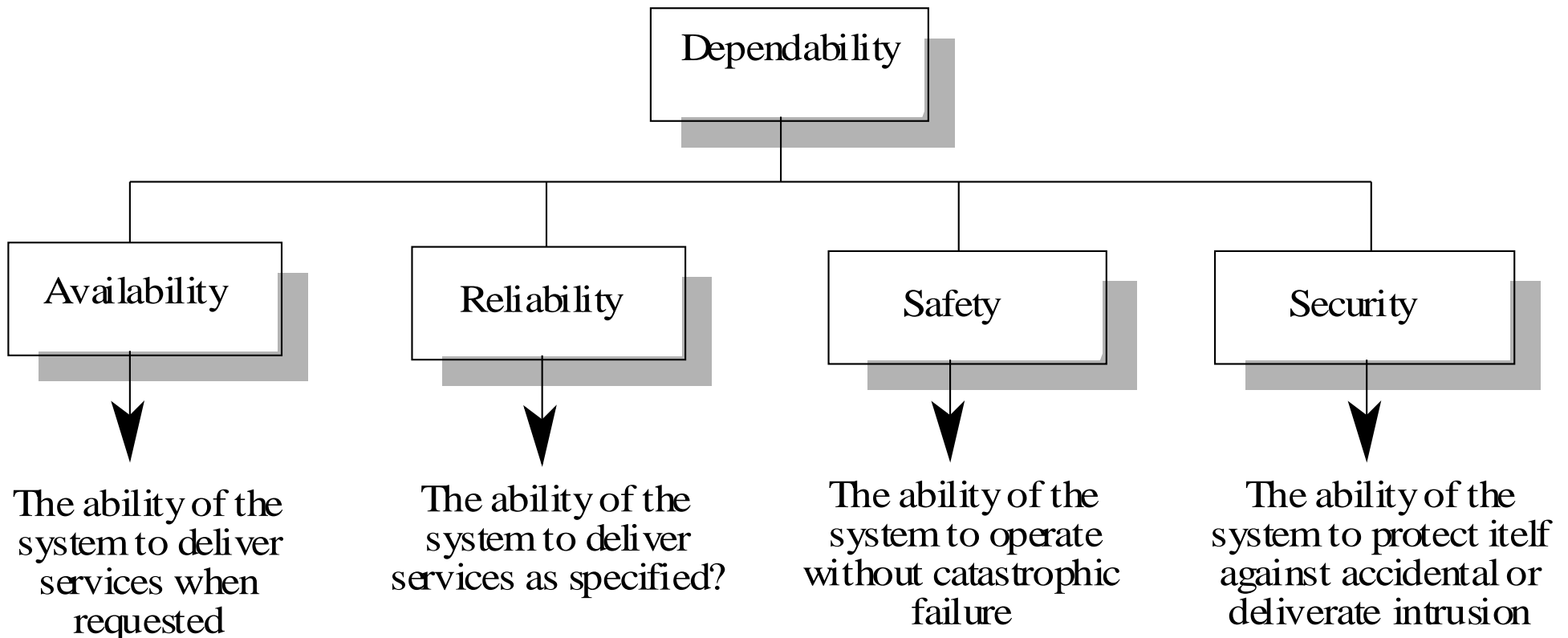


Fault Tolerance

- While in non-distributed systems a fault is often total – bringing down the whole application.
- In distributed systems it is possible to have a partial fault, from which we can recover.

Dependability

Fault tolerance is closely related to dependability.



Dependability in Distributed Systems

- **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).

What is failure?



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 three 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

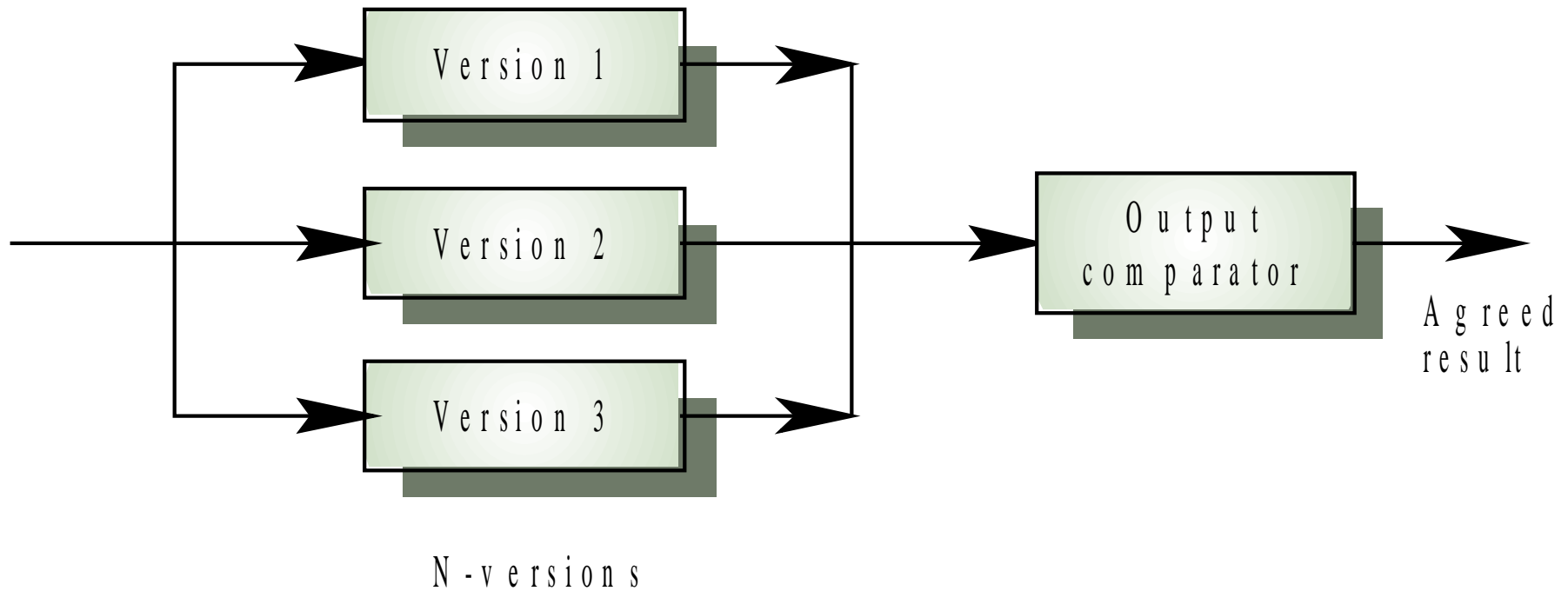
Types of Failure

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts.
Omission failure <i>Receive omission</i> <i>Send omission</i>	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 <i>Value failure</i> <i>State transition failure</i>	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.

Masking Failure through Redundancy

- **Strategy:** hide the occurrence of failure from other processes using *redundancy*. Three main types:
- *Information Redundancy* – add extra bits to allow for error detection/recovery
- *Time Redundancy* – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).
- *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

Triple Modular Redundancy



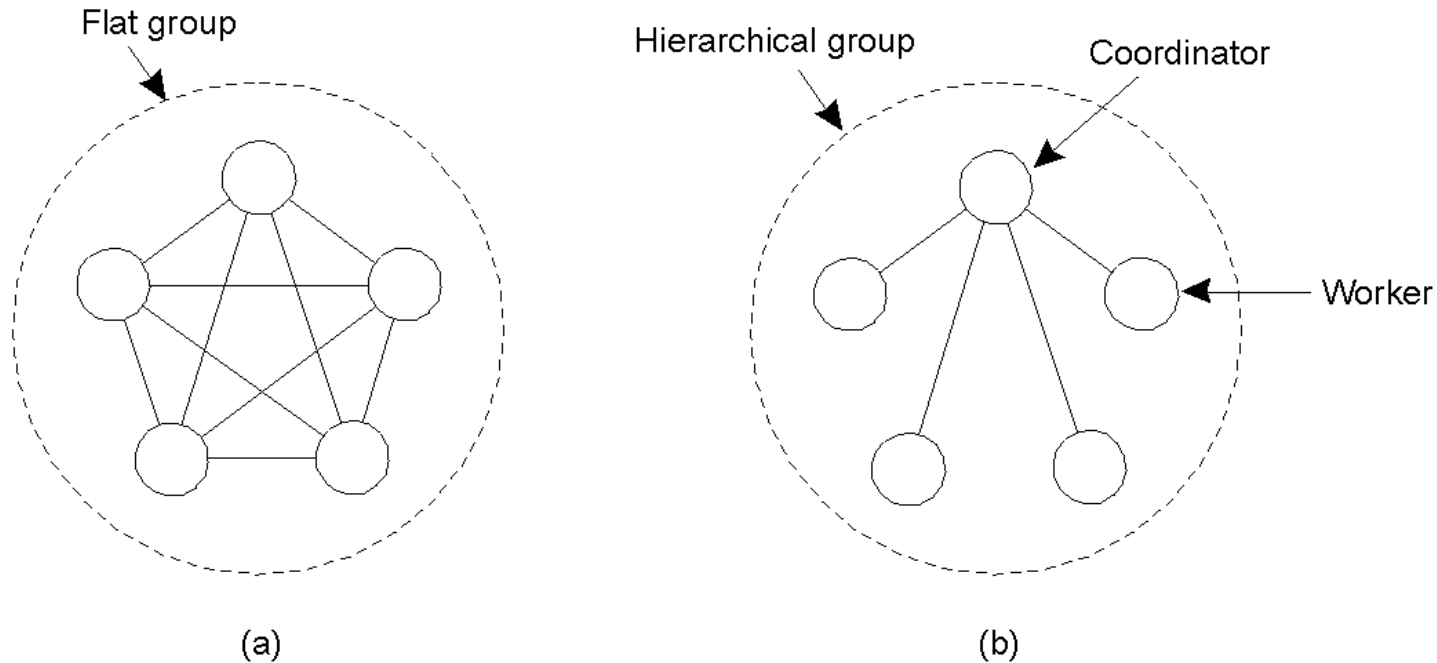
We'll look at the following:

1. Process Resilience
- 2.
3. Reliable Client/Server Communications
- 4.
5. Reliable Group Communication
- 6.
7. Distributed COMMIT
- 8.
9. Recovery Strategies

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 (the group members), and then *only one* of them performs the required service.
- If one of the processes fail, 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

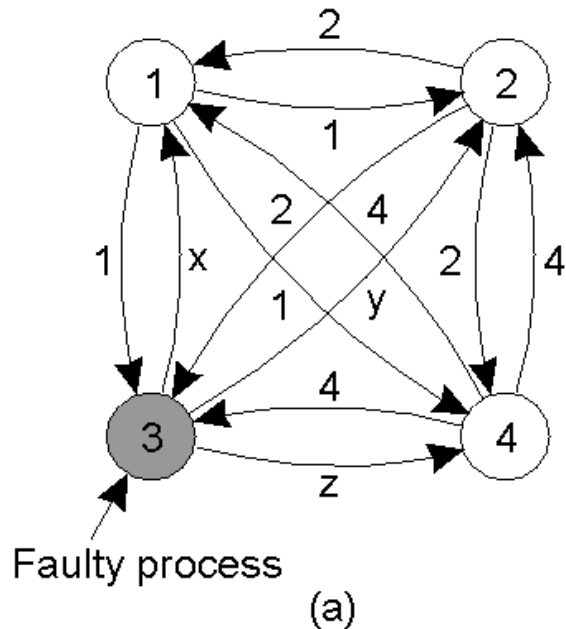


- a) **Communication in a flat group** – all the processes are equal, decisions are made collectively. **Note:** no single point-of-failure, however: decision making is complicated as consensus is required.
- b) **Communication in a simple hierarchical group** - one of the processes is elected to be the coordinator, which selects another process (a worker) to perform the operation. **Note:** single point-of failure, however: decisions are easily and quickly made by the coordinator without first having to get consensus.

Failure Masking and Replication

- A system is 'k fault tolerant' if it can survive faults in k components and still meet its specifications.
 - If components fail quietly, then $k+1$ components are needed to be k fault tolerant.
 - If components exhibit 'Byzantine Failures' then $2k+1$ components are needed.

Dealing with Malicious Members



1 Got(1, 2, x, 4)
 2 Got(1, 2, y, 4)
 3 Got(1, 2, 3, 4)
 4 Got(1, 2, z, 4)

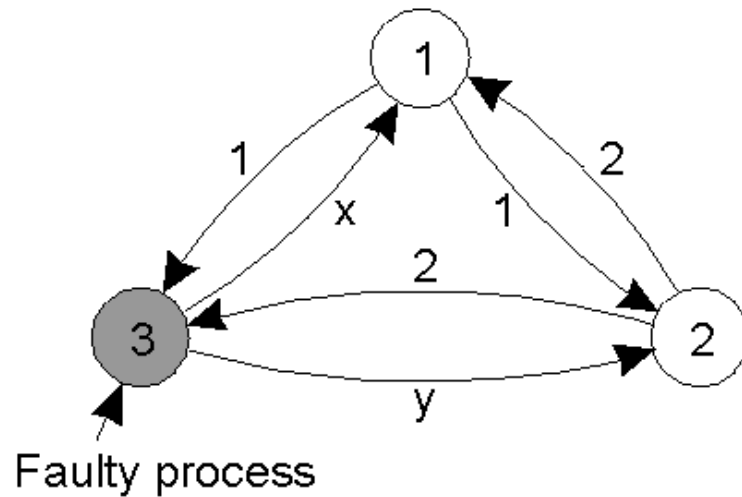
(b)

1 Got	2 Got	4 Got
<u>(1, 2, y, 4)</u>	<u>(1, 2, x, 4)</u>	<u>(1, 2, x, 4)</u>
(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.
- b) The generals announce their troop strengths (in units of 1 kilosoldiers) to the other members of the group by sending a message.
- c) 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.
- d) 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.

Not enough Generals



(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. Note: 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 two-thirds of the processes have to be working correctly. That is: if there are k faulty processes, we need $2k + 1$ functioning processes to reach agreement.

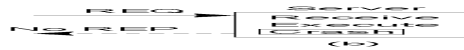
Reliable Client/Server Communications

- In addition to process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures.
- In practice, the focus is on masking *crash* and *omission* failures.
- ***For example:*** the 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).

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

5 Classes – 3 results



- A server in client-server communication.
- c) The normal case.
- d) Crash *after* service execution.
- e) Crash *before* service execution.

Solutions and Problems

- An appropriate exception handling mechanism can deal with a missing server. However, such technologies tend to be very language-specific, and they also tend to be 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.

Server Crash semantics

- 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*.

Idempotency

- Lost replies are difficult to deal with.
- *Why* was there no reply? Is the server *dead*, *slow*, or did the reply just go *missing*? Emmmmmm?
- A request that can be repeated any number of times without any nasty side-effects is said to be *idempotent*. (For example: a read of a static web-page is said to be idempotent).
- *Nonidempotent* requests (for example, the electronic transfer of funds) are a little harder to deal with. A 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.

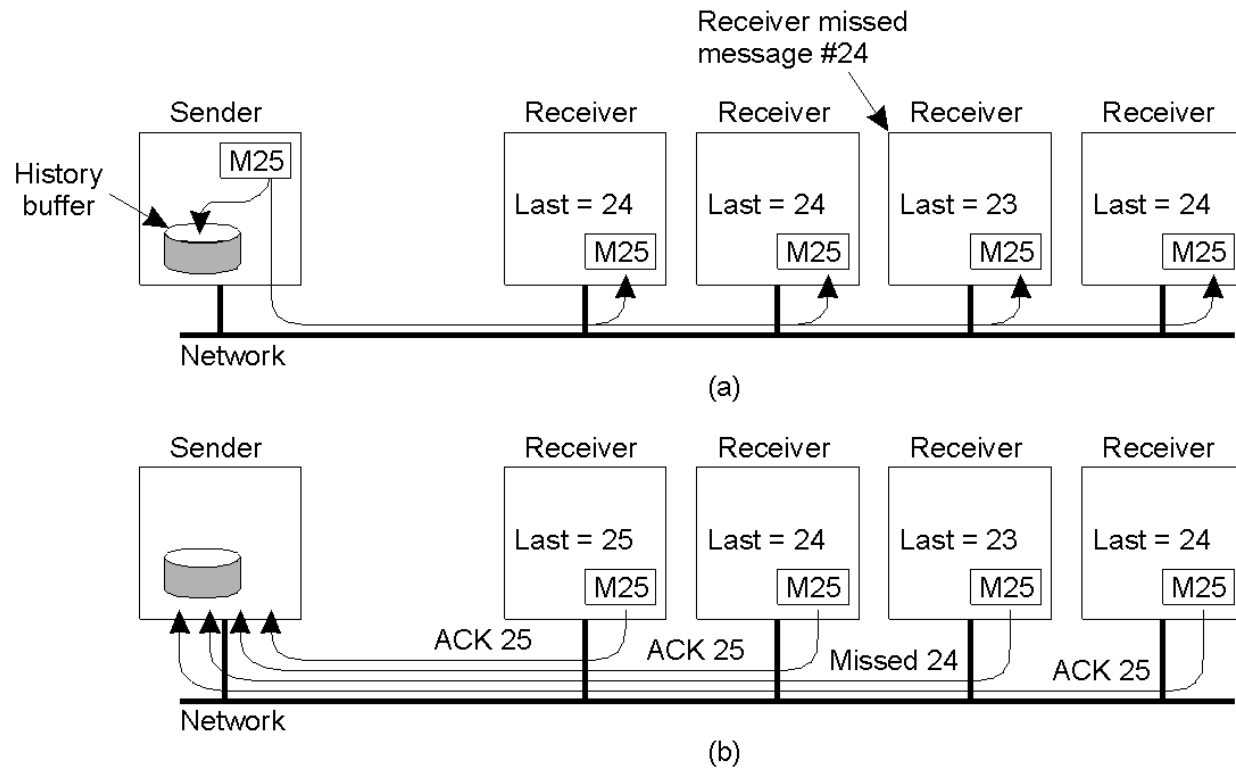
Orphans

- 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 standard amount of time, it is assumed to have expired).
- In practice, however, none of these methods are desirable for dealing with orphans. Research continues ...

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* (as multicasting services tend to be *inherently* unreliable).
- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution *scales poorly* as the group membership grows. Also:
 - 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 Multicasting

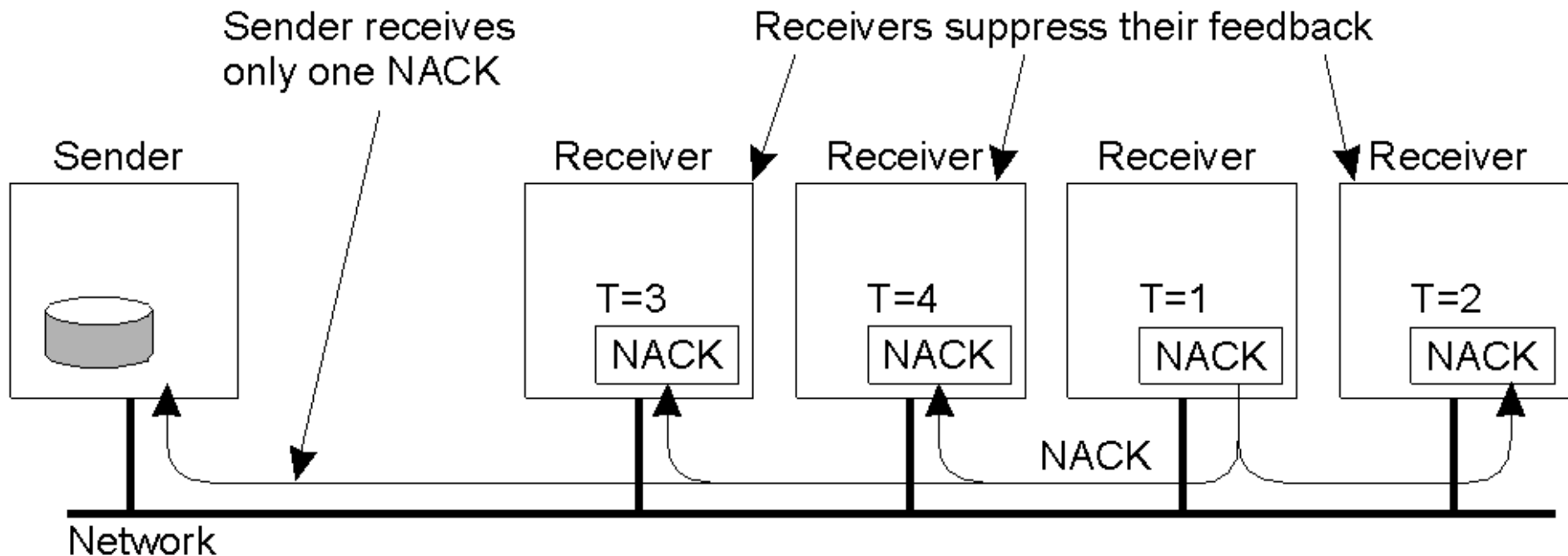


- This is a simple solution to reliable multicasting when *all receivers are known* and are assumed *not to fail*. The sending process assigns a sequence number to outgoing messages (making it easy to spot when a message is missing).
- b) Message transmission – note that the third receiver is expecting 24.
- c) Reporting feedback – the third receiver informs the sender.
- d) But, how long does the sender keep its *history-buffer* populated?
- e) Also, such schemes **perform poorly** as the group grows ... there are *too many* ACKs.

Scalable Reliable Multicasting

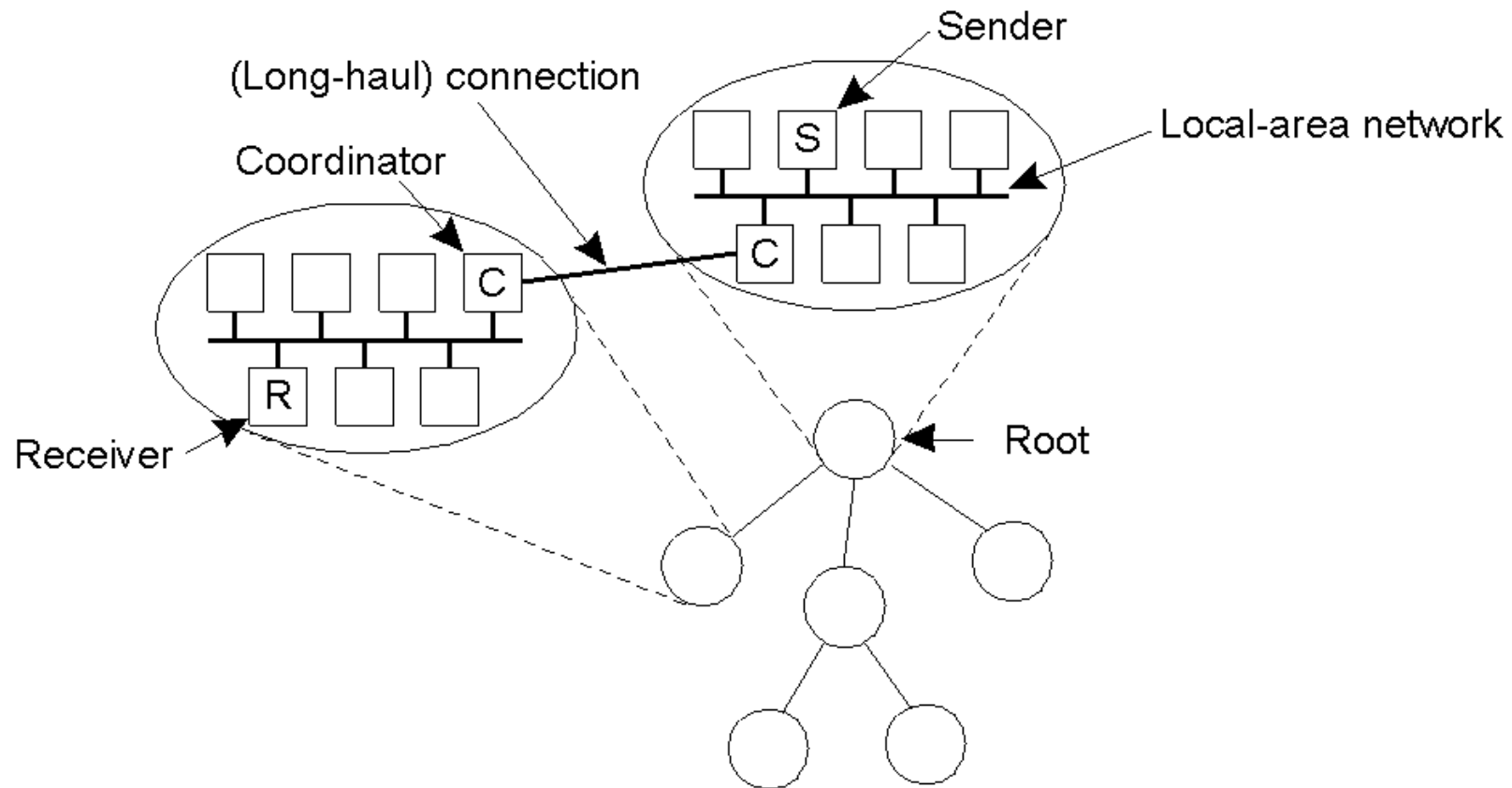
- Receivers *never* acknowledge successful delivery.
- **Only missing messages are reported.**
- NACKs are multicast to all group members.
- This allows other members to suppress their feedback, if necessary.
- To avoid “retransmission clashes”, each member is required to wait a random delay prior to NACKing.

Non-Hierarchical Feedback Control



- *Feedback Suppression* – reducing the number of feedback messages to the sender (as implemented in the *Scalable Reliable Multicasting Protocol*).
- Successful delivery is never acknowledged, 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. In this way, 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**.
- b) Sub-groups within the entire group are created, with each *local coordinator* forwarding messages to its children.
- c) A local coordinator handles retransmission requests *locally*, using any appropriate multicasting method for small groups.

Distributed Commit

- **General Goal:**
- *We want an operation to be performed by all group members, or none at all.*
- [In the case of atomic multicasting, the operation is the delivery of the message.]
- There are three types of “commit protocol”: single-phase, two-phase and three-phase commit.

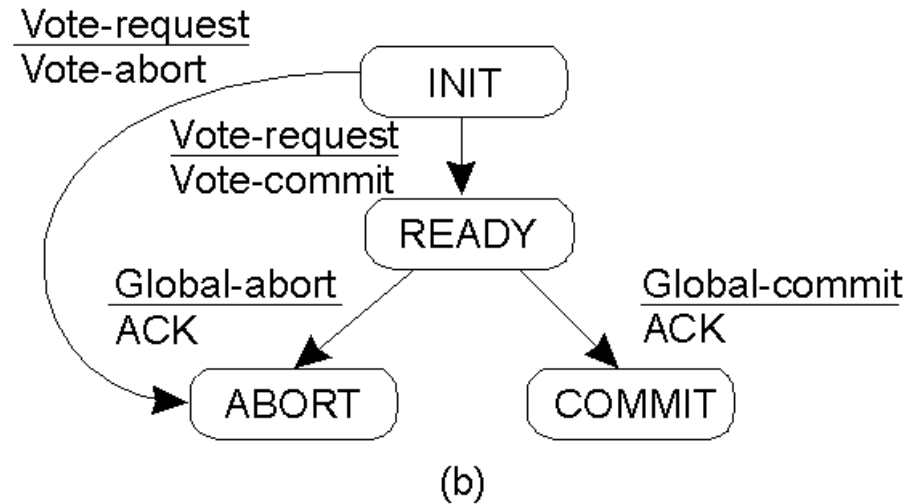
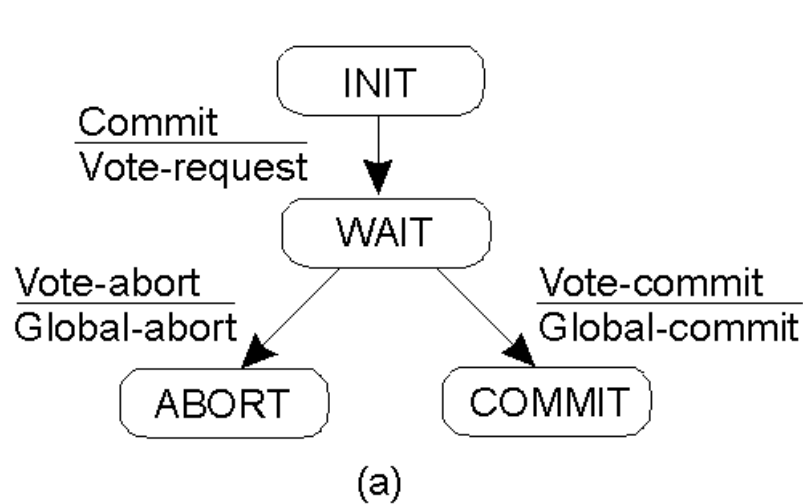
Commit Protocols

- **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's no way to tell the coordinator! Whoops ...
- **The solutions:**
- The *Two-Phase* and *Three-Phase Commit Protocols*.

Two-Phase Commit Protocol

- First developed in 1978!!!
 - *Summarized: GET READY, OK, GO AHEAD.*
3. The coordinator sends a *VOTE_REQUEST* message to all group members.
 4. The group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
 5. All votes are collected by the coordinator. A *GLOBAL_COMMIT* is sent if all the group members voted to commit. If one group member voted to abort, a *GLOBAL_ABORT* is sent.
 6. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

2 Phase Commit, Finite State Machines

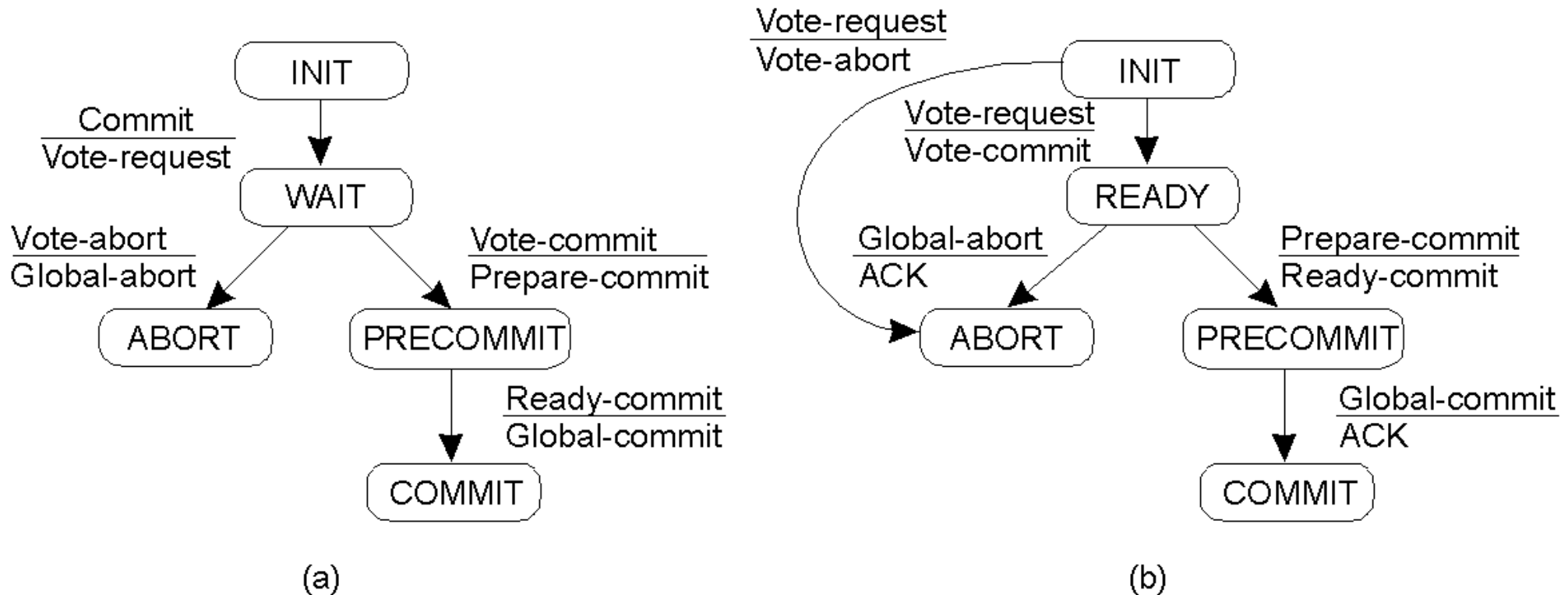


- a) The finite state machine for the coordinator.
- b) The finite state machine for a participant (group member).

Problem

- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.
- If the coordinator crashes, the 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? *The Three-Phase Commit Protocol*.

3 Phase Commit



- a) Finite state machine for the coordinator.
- b) Finite state machine for a group member.
- c) **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.
- d) Refer to the textbook for details on how this works.

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:
 - 7. Backward Recovery:** return the system to some previous correct state (using *checkpoints*), then continue executing.
 - 8. 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. The “logging” of information can be thought of as a type of checkpointing.].

- **Disadvantage of Forward Recovery:**
- In order to work, 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 Examples

- **Consider as an 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*. This is an example of a forward recovery technique.