Fault Tolerance

Dealing successfully with *partial failure* within a Distributed System. (a review by Gartner 1999)

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).

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".
- Distinction between preventing, removing, and forecasting faults (Avizienis et al., 2004).

- Fault tolerance - meaning that a system can provide its services even in the presence of faults.
    - The system can tolerate faults and continue to operate normally.

Types of Faults

- *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.

Failure Models

Different types of failures. (Cristian 1991) and (Hadzilacos and Toueg 1993).

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

Three main types:

- *Information Redundancy* – add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).
- *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.

Distributed Systems Fault Tolerance Topics

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

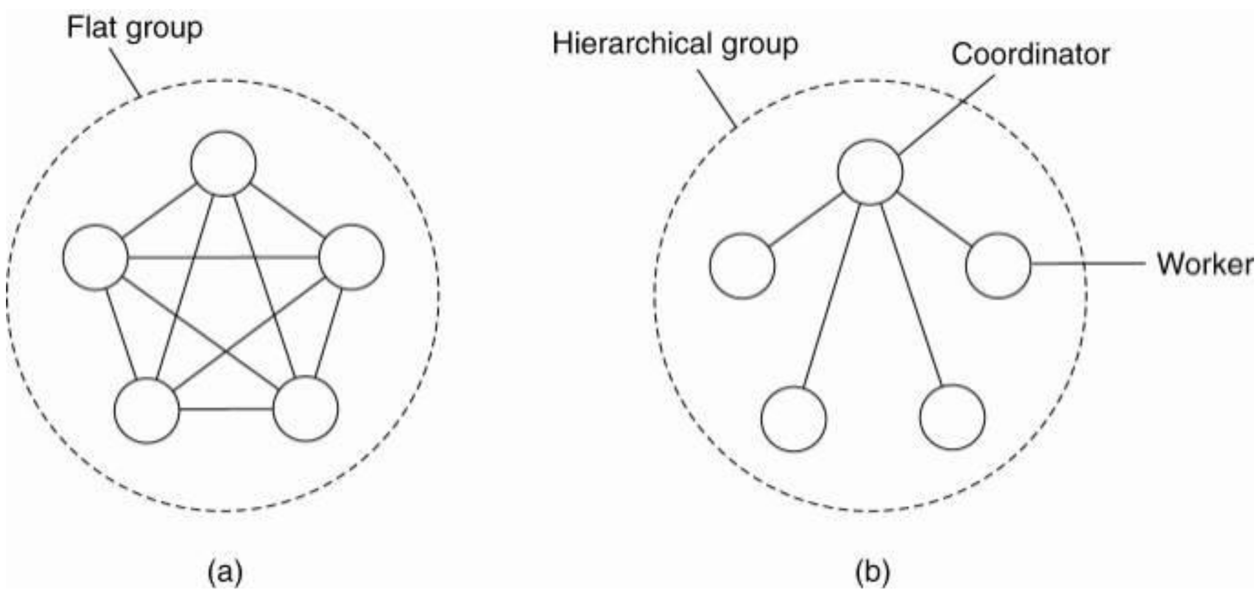
Process Resilience

(Guerraoui and Schiper, 1997)

- 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 Groups versus Hierarchical Groups

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



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.

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

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

Two approaches to arranging the replication of the group:

Primary (backup) Protocols

- A group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations.
- When the primary crashes, the backups execute some election algorithm to choose a new primary.

Replicated-Write Protocols

- Replicated-write protocols are used in the form of active replication, as well as by means of quorum-based protocols.
- Solutions correspond to organizing a collection of identical processes into a flat group.

- Adv. - these groups have no single point of failure, at the cost of distributed coordination.

Agreement in Faulty Systems

Goal of distributed agreement algorithms - have all the non-faulty processes reach consensus on some issue, and to establish that consensus within a finite number of steps.

Complications:

- Different assumptions about the underlying system require different solutions, assuming solutions even exist.
- Turek and Shasha (1992) distinguish the following cases:

1. Synchronous versus asynchronous systems.

- A system is synchronous if and only if the processes are known to operate in a lock-step mode.
- Formally, this means that there should be some constant c >= 1, such that if any processor has taken c + 1 steps, every other process has taken at least 1 step.
- A system that is not synchronous is said to be asynchronous.

2. Communication delay is bounded or not.

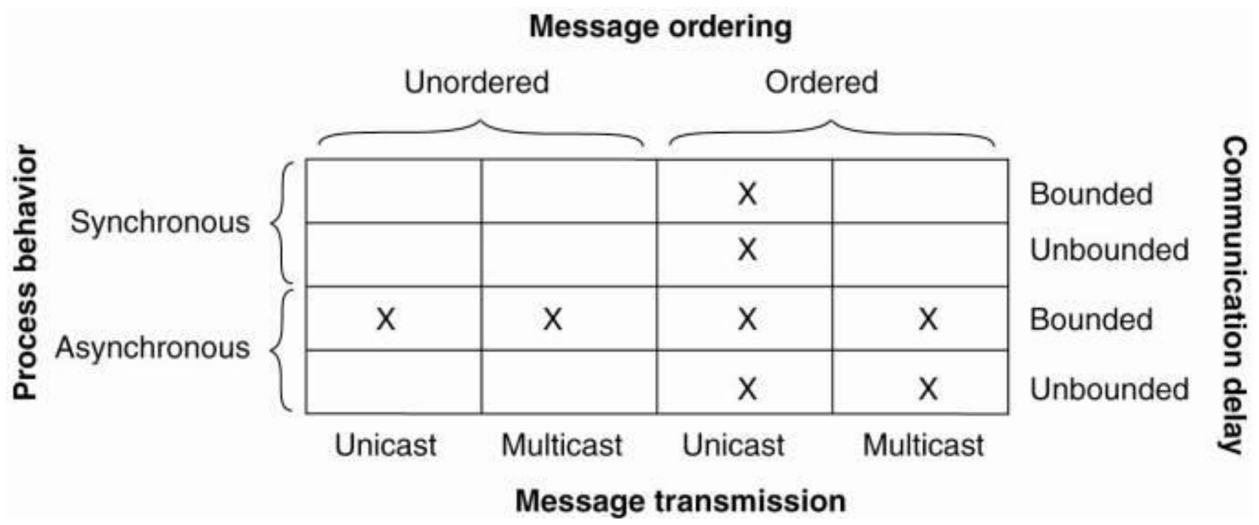
- Delay is bounded if and only if we know that every message is delivered with a globally and predetermined maximum time.

3. Message delivery is ordered or not.

- In other words, we distinguish the situation where messages from the same sender are delivered in the order that they were sent, from the situation in which we do not have such guarantees.

4. Message transmission is done through unicasting or multicasting.

Circumstances under which distributed agreement can be reached.



Process behavior		Message ordering: Unordered		Ordered		Communication delay
		Unicast	Multicast	Unicast	Multicast	
Synchronous				X		Bounded
				X		Unbounded
Asynchronous		X	X	X	X	Bounded
				X	X	Unbounded
		Message transmission				

- **In all other cases, it can be shown that no solution exists.**

Note - most distributed systems in practice assume that processes behave asynchronously, message transmission is unicast, and communication delays are unbounded.

- Known as the Byzantine agreement problem (Lamport et al. 1982)

History Lesson: The Byzantine Empire

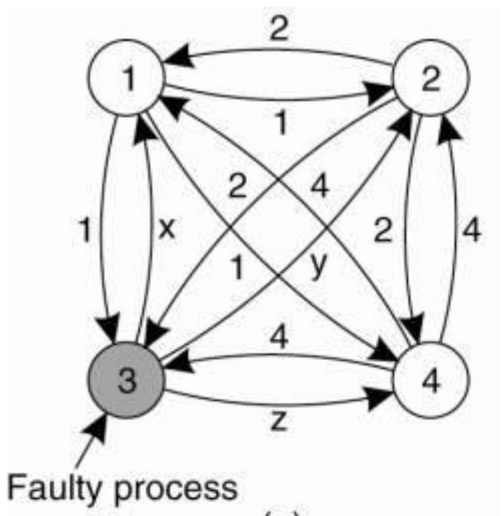
- *Time*: 330-1453 AD.
- *Place*: Balkans and Modern Turkey.
- Endless conspiracies, intrigue, and untruthfullness were alleged to be common practice in the ruling circles of the day (*sounds strangely familiar* …).
- That is: it was typical for intentionally wrong and malicious activity to occur among the ruling group. A similar occurance can surface in a DS, and is known as 'byzantine failure'.

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

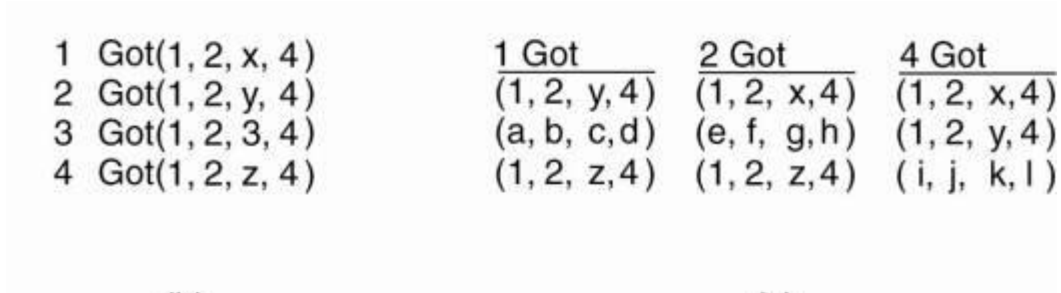
How does a process group deal with a faulty member?

The “Byzantine Generals Problem” for 3 loyal generals and 1 traitor.

1. The generals announce their troop strengths (in units of 1 kilosoldiers) to the other members of the group by sending a message.
2. 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.
3. 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.



(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)

Goal of Byzantine agreement is that consensus is reached on the value for the non-faulty processes only.

Solution in computer terms:

- Assume that processes are synchronous, messages are unicast while preserving ordering, and communication delay is bounded.
- Assume N processes, where each process i will provide a value vi to the others.
- Goal - let each process construct a vector V of length N, such that if process i is non-faulty, V [i] = vi
    - ELSE V [i] is undefined.
- We assume that there are at most k faulty processes.
- Algorithm for the case of N = 4 and k = 1.

Algorithm operates in four steps.

1. Every non-faulty process i sends vi to every other process using reliable unicasting.

- Faulty processes may send anything and different values to different processes.
- Let vi =i. In (Fig.a) t process 1 reports 1, process 2 reports 2, process 3 lies to everyone, giving x, y, and z, respectively, and process 4 reports a value of 4.

2. The results of the announcements of step 1 are collected together in the form of the vectors (Fig.b).
3. Every process passes its vector from (Fig.b) to every other process.

- Every process gets three vectors, one from every other process.
- Process 3 lies, inventing 12 new values, a through l.
- Results in (Fig.c).

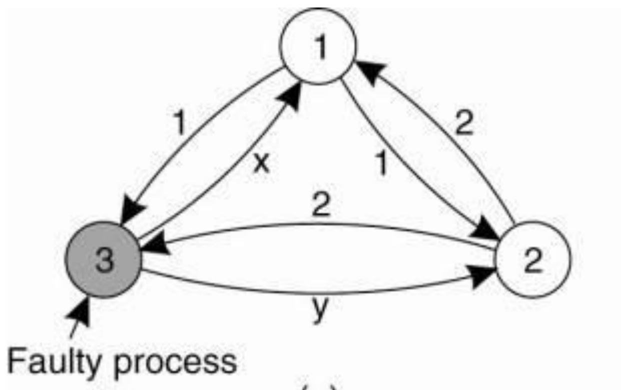
4. Each process examines the ith element of each of the newly received vectors.

- If any value has a majority, that value is put into the result vector.
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN.
- From (Fig.c) - 1, 2, and 4 all come to agreement on the values for v1, v2, and v4, which is the correct result.
- What these processes conclude regarding v 3 cannot be decided, but is also irrelevant.

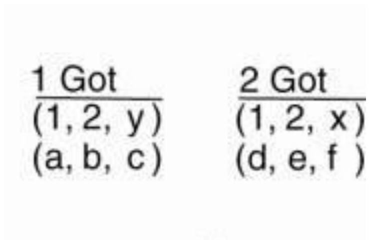
Example Again:
With 2 loyal generals and 1 traitor.
Note: It is no longer possible to determine the majority value in each column, and the algorithm has failed to produce agreement.

- Lamport et al. (1982) proved that in a system with k faulty processes, agreement can be achieved only if 2k + 1 correctly functioning processes are present, for a total of 3k + 1.
- Agreement is possible only if more than two-thirds of the processes are working properly.

Two correct process and one faulty process.

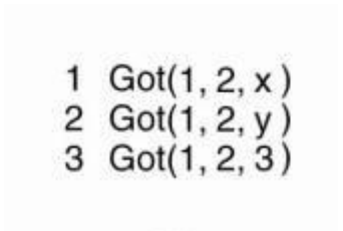


(a)



1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)



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

(b)

Reliable Client-Server Communication

Kinds of Failures:

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

Detecting process failures:

Processes actively send "are you alive?" messages to each other (for which they obviously expect an answer)

- Makes sense only when it can be guaranteed that there is enough communication between processes.

Processes passively wait until messages come in from different processes.

- In practice, actively pinging processes is usually followed.

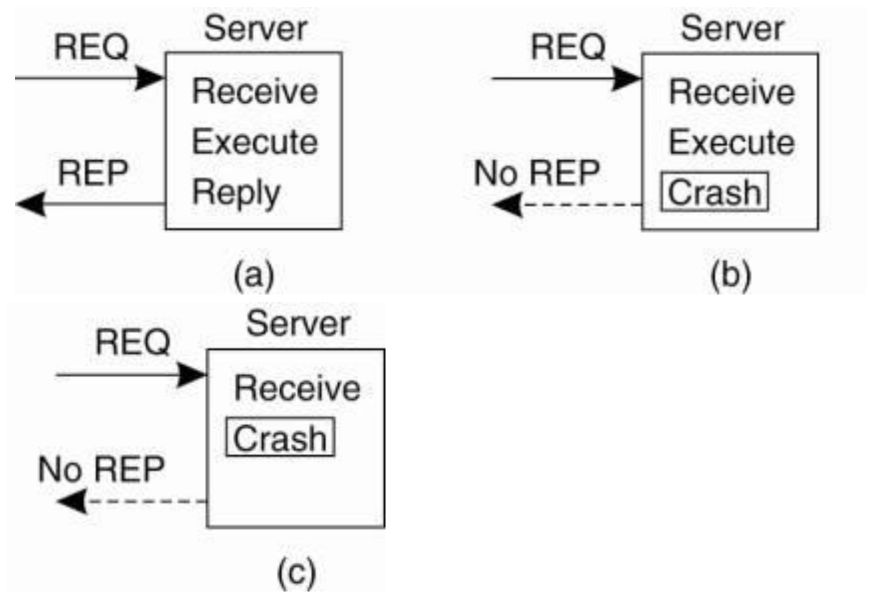
Example: RPC Semantics and Failures

Remote Procedure Call (RPC) mechanism works well as long as both the client and server function perfectly!!!

Five classes of RPC failure can be identified:

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

A server in client-server communication.
(a). A request arrives, is carried out, and a reply is sent.
(b). A request arrives and is carried out, just as before, but the server crashes before it can send the reply.
(c). Again a request arrives, but this time the server crashes before it can even be carried out. And, no reply is sent back.



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 that 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*.

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.

- (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.

Client Crashes

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

Four orphan solutions have been proposed:

1. ***extermination*** (the orphan is simply killed-off),
2. ***reincarnation*** (each client session has an *epoch* associated with it, making orphans easy to spot),
3. ***gentle reincarnation*** (when a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed),
4. ***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.
Orphan elimination is discussed in more detail by Panzieri and Shrivastava (1988).

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).

Small group: multiple, reliable point-to-point channels will do the job, however, such a solution *scales poorly* as the group membership grows.

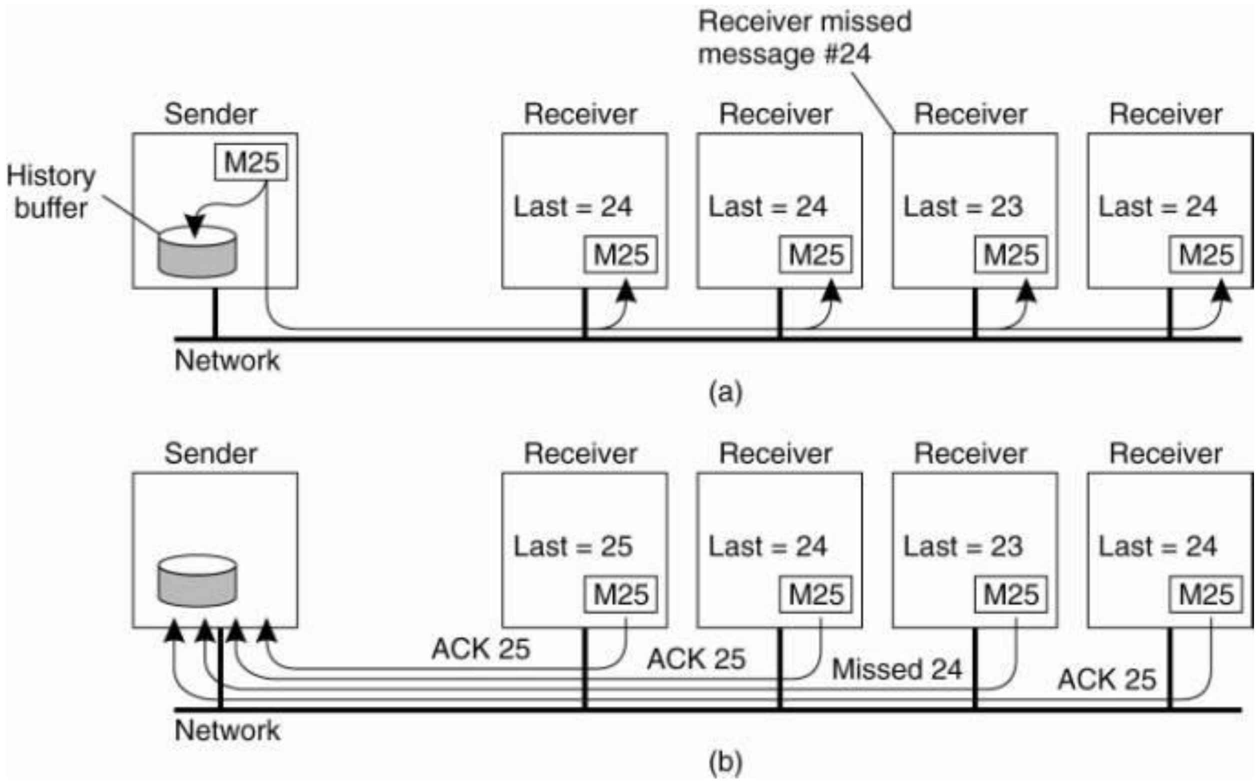
- What happens if a process *joins* the group during communication?
- Worse: what happens if the sender of the multiple, reliable point-to-point channels *crashes* half way through sending the messages?

Basic Reliable-Multicasting Schemes

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).
- Assume that messages are received in the order they are sent.
- Each multicast message is stored locally in a history buffer at the sender.
- Assuming the receivers are known to the sender, the sender simply keeps the message in its history buffer until each receiver has returned an acknowledgment.
- If a receiver detects it is missing a message, it may return a negative acknowledgment, requesting the sender for a retransmission.

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 … there are *too many* ACKs.

A extensive and detailed survey of total-order broadcasts can be found in Defago et al. (2004).

Scalability in Reliable Multicasting

- Receivers *never* acknowledge successful delivery.
- Only missing messages are reported.
- Negative acknoledgements (NACK) are multicast to all group members. (Don't send any more.)
- This allows other members to supress their feedback, if necessary.
- To avoid “retransmission clashes”, each member is required to wait a random delay prior to NACKing.
- See Towsley et al. (1997) for details - but no hard guarantees can be given that feedback implosions will never happen.

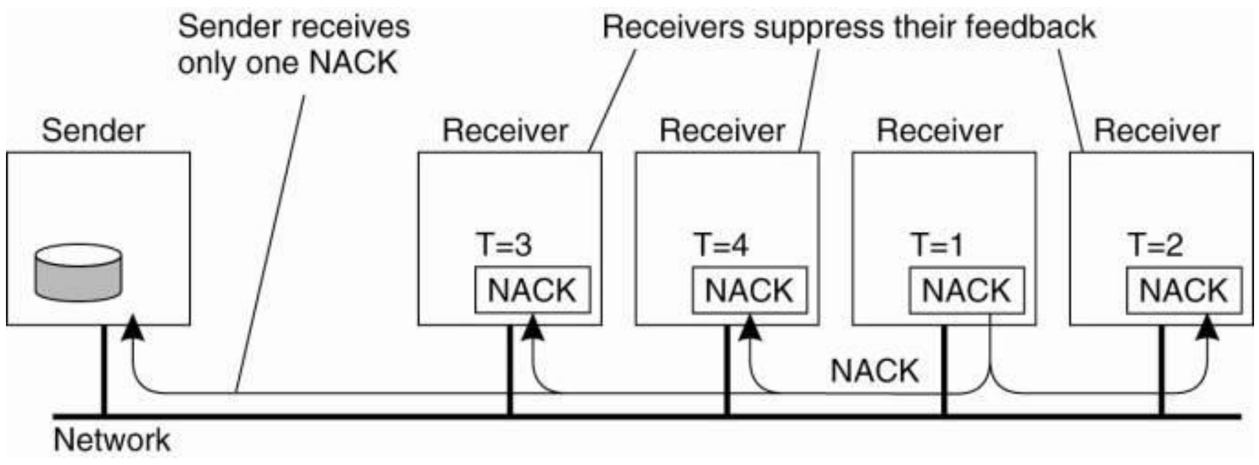
A comparison between different scalable reliable multicasting can be found in Levine and Garcia-Luna-Aceves (1998).

Nonhierarchical Feedback Control

Feedback Suppression – reducing the number of feedback messages to the sender (as implemented in the *Scalable Reliable Multicasting Protocol*). Floyd et al. (1997) and Liu et al. (1998)

- 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.

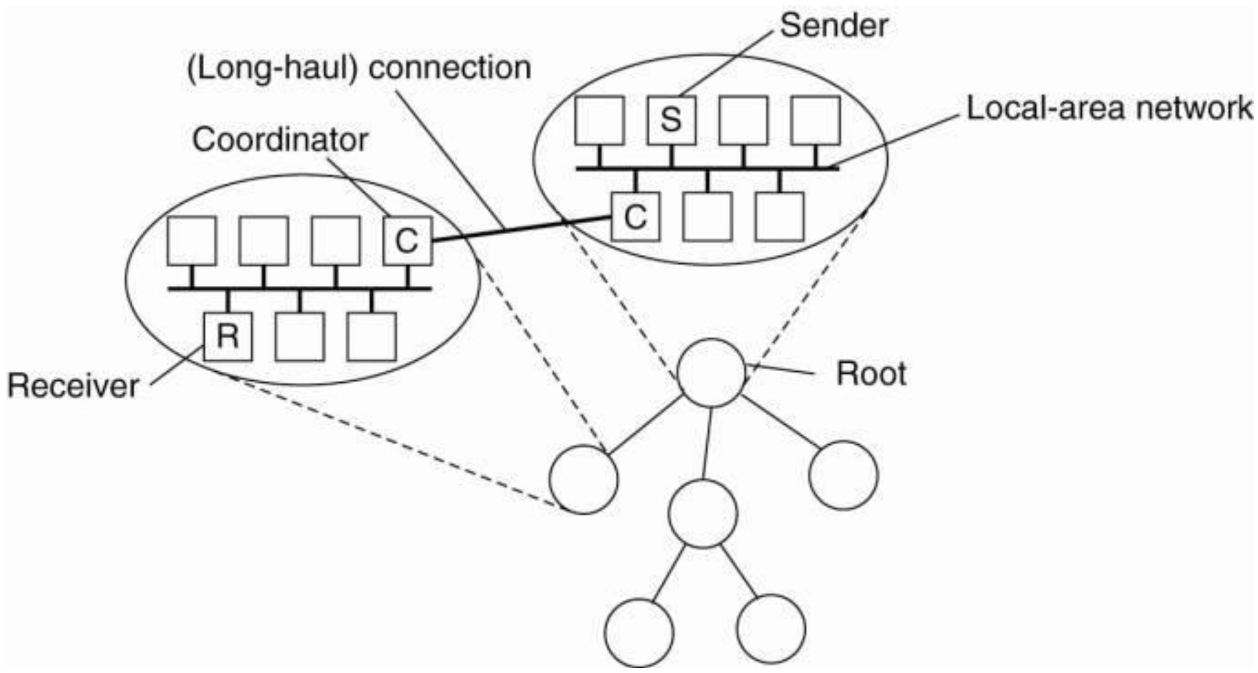
Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.



Hierarchical Feedback Control

Hierarchical reliable multicasting - the main characteristic is that it supports the creation of **very large groups**.

a)Sub-groups within the entire group are created, with each *local coordinator* forwarding messages to its children.
b)A local coordinator handles retransmission requests *locally*, using any appropriate multicasting method for small groups.



Main problem : construction of the tree.

Conclusion:

- Building reliable multicast schemes that can scale to a large number of receivers spread across a wide-area network, is a difficult problem.
- No single best solution exists, and each solution introduces new problems.

Atomic Multicast
Atomic multicast problem:

- A requirement where the system needs to ensure that all processes get the message, or that none of them get it.
- An additional requirement is that all messages arrive at all processes in sequential order.

- Atomic multicasting ensures that nonfaulty processes maintain a consistent view of the database, and forces reconciliation when a replica recovers and rejoins the group.

Virtual Synchrony

The concept of virtual synchrony was proposed by Kenneth Birman as the abstraction that group communication protocols should attempt to build on top of an asynchronous system.

Virtual synchrony is defined as follows:

1. All recipients have identical group views when a message is delivered. (The group view of a recipient defines the set of "correct" processors from the perspective of that recepient.)
2. The destination list of the message consists precisely of the members in that view
3. The message should be delivered either to all members in its destination list or to no one at all. The latter case can occur only if the sender fails during transmission.
4. Messages should be delivered in causal or total order (depending on application semantics).

Reliable multicast with the above properties is said to be virtually synchronous (Birman and Joseph, 1987).

- Whole idea of atomic multicasting is that a multicast message m is uniquely associated with a list of processes to which it should be delivered.
- Delivery list corresponds to a group view, namely, the view on the set of processes contained in the group, which the sender had at the time message m was multicast. (Virtual synchrony #2)
- Each process on that list has the same view. In other words, they should all agree that m should be delivered to each one of them and to no other process. (Virtual synchrony #1)
- Need to guarantee that m is either delivered to all processes in the list in order or m is not delivered at all. (Virtual synchrony #3 & #4)

Message Ordering

Four different orderings:

1. Unordered multicasts

- virtually synchronous multicast in which no guarantees are given concerning the order in which received messages are delivered by different processes

2. FIFO-ordered multicasts

- the communication layer is forced to deliver incoming messages from the same process in the same order as they have been sent

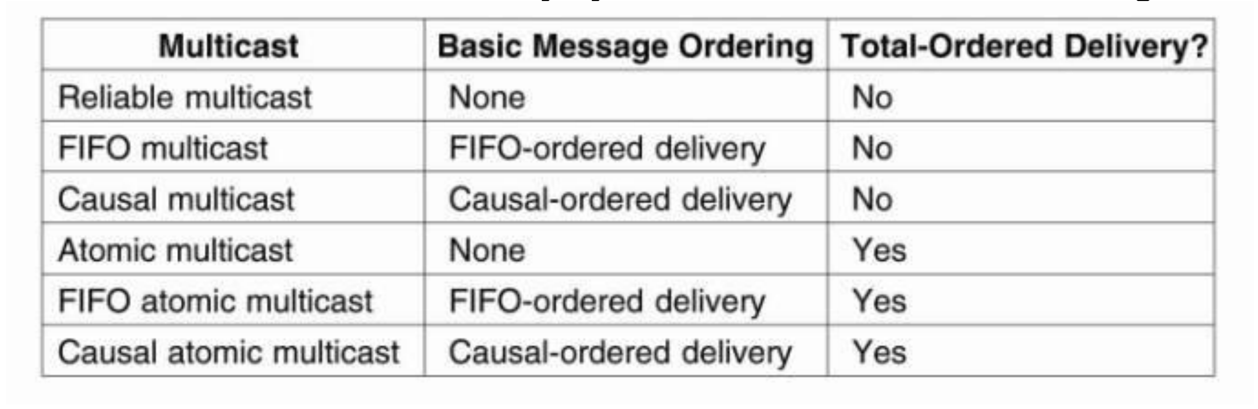
3. Causally-ordered multicasts

- delivers messages so that potential causality between different messages is preserved

4. Totally-ordered multicasts

- regardless of whether message delivery is unordered, FIFO ordered, or causally ordered, it is required additionally that when messages are delivered, they are delivered in the same order to all group members.
- Virtually synchronous reliable multicasting offering totally-ordered delivery of messages is called atomic multicasting.
- With the three different message ordering constraints discussed above, this leads to six forms of reliable multicasting (Hadzilacos and Toueg, 1993).

Six different versions of virtually synchronous reliable multicasting.



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

Distributed Commit

Examples of distributed commit, and how it can be solved are discussed in Tanisch (2000).

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.

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!
- **The solutions**: T*wo-Phase* and *Three-Phase Commit Protocols*

Two-Phase Commit Protocol:

- First developed in 1978!!! Gray (1978)
- *Summarized: GET READY, OK, GO AHEAD.*

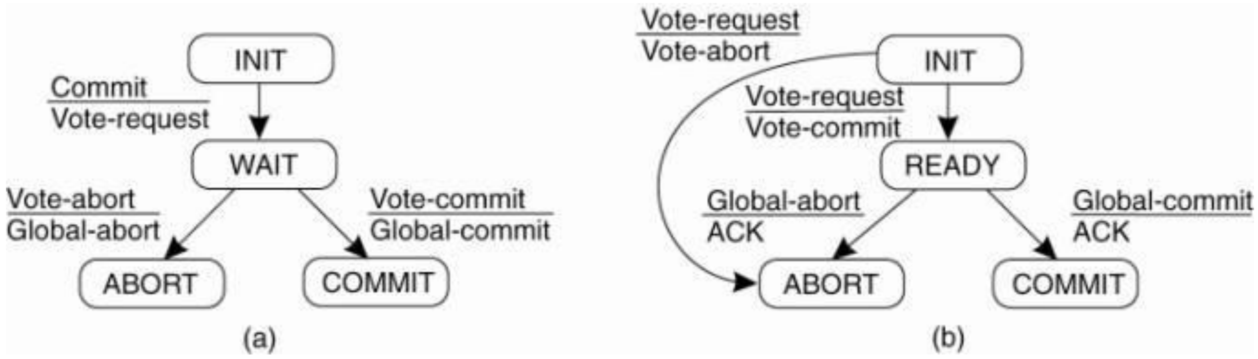
1. The coordinator sends a *VOTE_REQUEST* message to all group members.
2. A group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
3. All votes are collected by the coordinator.

- 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.

4. Group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

First phase - voting phase - steps 1 and 2.
Second phase - decision phase steps 3 and 4.

(a) The finite state machine for the coordinator in 2PC.
(b) The finite state machine for a participant.



Big Problem with Two-Phase Commit

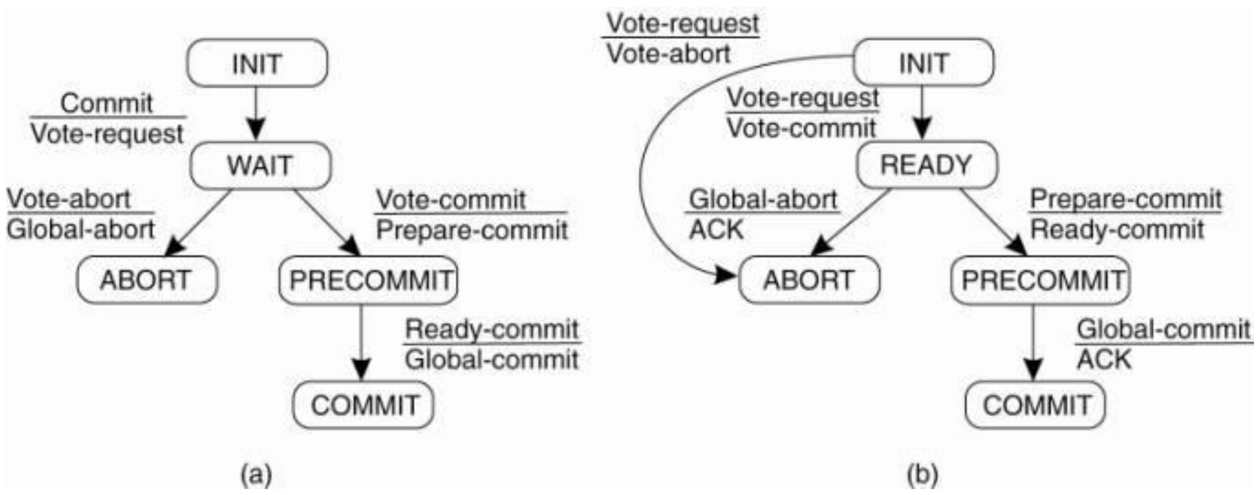
- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.
- If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers …*
- Two-Phase Commit is known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

Three-Phase Commit Protocol:

Essence: 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.

(a) The finite state machine for the coordinator in 3PC.
(b) The finite state machine for a participant.


INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Prepare-commit
ABORT
PRECOMMIT
Ready-commit / Global-commit
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Prepare-commit / Ready-commit
ABORT
PRECOMMIT
Global-commit / ACK
COMMIT
(b)

Recovery

- 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

Backward Recovery:

Advantages

- Generally applicable independent of any specific system or process.
- It can be integrated into (the middleware layer) of a distributed system as a general-purpose service.

Disadvantages:

- 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.].

- Recovery mechanisms are independent of the distributed application for which they are actually used – thus no guarantees can be given that once recovery has taken place, the same or similar failure will not happen again.

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.

Example
Consider as an example: Reliable Communications.
Retransmission of a lost/damaged packet - backward recovery technique.
Erasure Correction - When a lost/damaged packet can be reconstructed as a result of the receipt of other successfully delivered packets - forward recovery technique. [see Rizzo (1997)]

- Elnozahy et al. (2002) and (Elnozahy and Planck, 2004) provide a survey of checkpointing and logging in distributed systems.
- See also Alvisi and Marzullo (1998) for message-logging schemes.

Recovery-Oriented Computing
Recovery-oriented computing - Start over again (Candea et al., 2004a).

- Underlying principle - it may be much cheaper to optimize for recovery, then it is aiming for systems that are free from failures for a long time.

Different flavors:

- Simply reboot (part of a system)
- e.g. restart Internet servers (Candea et al., 2004, 2006).

- To reboot only a part of the system - i the fault is properly localized.
- means deleting all instances of the identified components, along with the threads operating on them, and (often) to just restart the associated requests.
- Apply checkpointing and recovery techniques, but to continue execution in a changed environment.
- Basic idea - many failures can be simply avoided if programs are given extra buffer space, memory is zeroed before allocated, changing the ordering of message delivery (as long as this does not affect semantics), and so on (Qin et al., 2005).