Distributed Systems Lecture 10

Professor Louise E. Moser

Spring 2013

Fault Tolerance

- Definitions and models
- Process resilience
- Reliable client-server communication
- Reliable group communication
- Distributed commit
- Recovery

Dependability

- **Basics:** A **component** provides **services** to **clients**
- To provide services, a component may require services of other components and, thus, it can **depend** on other components
  - **Specifically, a component C depends on C* if the correctness of C's behavior depends on the correctness of C*'s behavior**
- Some properties of dependability

Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	Ease with which a failed system can be repaired

- **Note:** For distributed systems, components can be **processes** or **channels**

Terminology

- **Failure:** When a component is not living up to its specifications, a failure occurs
- **Error:** That part of a component's state that can lead to a failure
- **Fault:** The cause of an error

- **Fault prevention:** Prevent the occurrence of a fault
- **Fault tolerance:** Design a component so that it satisfies its specifications, even in the presence of faults (i.e., **mask** faults)
- **Fault removal:** Reduce the presence, number, and severity of faults
- **Fault forecasting:** Estimate the current number, future incidence, and consequences of faults

Failure Models

- Different types of failure

Type of failure	Description
Crash failure	A server halts, but works 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 does not occur within the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is incorrect The server deviates from the correct flow of control
Arbitrary (Byzantine) failure	A server produces an arbitrary response at an arbitrary time

Crash Failures

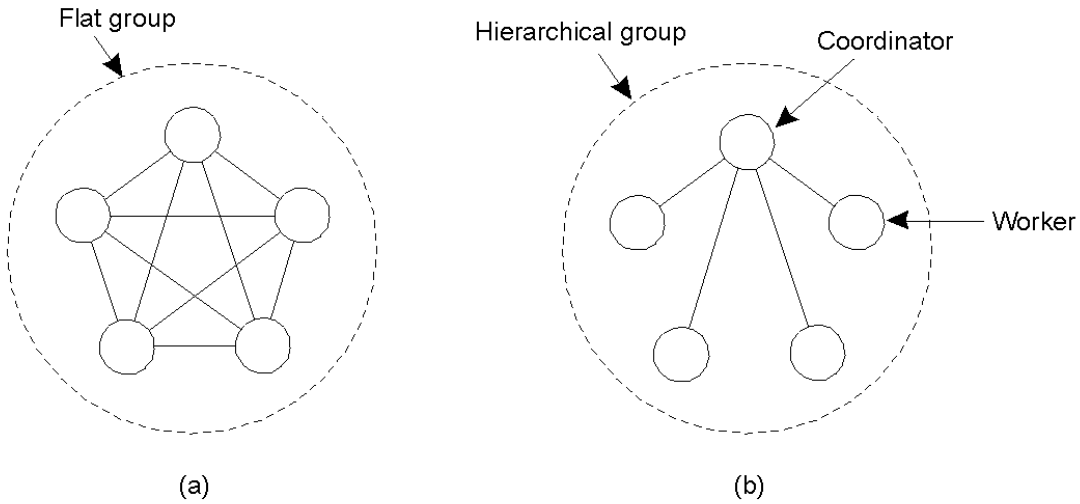
- **Basic Problem:** Clients cannot distinguish between a crashed component and a component that is merely slow
- **Examples:** Consider a server from which a client is expecting a response
  - **Is the server exhibiting timing or omission failures?**
  - **Is the channel between the client and the server faulty (crashed, or exhibiting timing or omission failures)?**
- **Fail-silent:** The component exhibits an omission or crash failure; the client cannot tell what went wrong
- **Fail-stop:** The component exhibits a crash failure, and its failure can be detected (either through an announcement or a timeout)
- **Fail-safe:** The component exhibits an arbitrary but benign failure (that doesn't do any harm)

Process Resilience

- **Basic strategy:** Protect the system against faulty processes by replicating processes and distributing computations to the group of process replicas

(a) Flat group: Good for fault tolerance, as information exchange with group members occurs immediately. But, more overhead (as control is completely distributed) and harder to implement

(b) Hierarchical group: All communication is through a single coordinator. Not really fault-tolerant, but simpler to implement

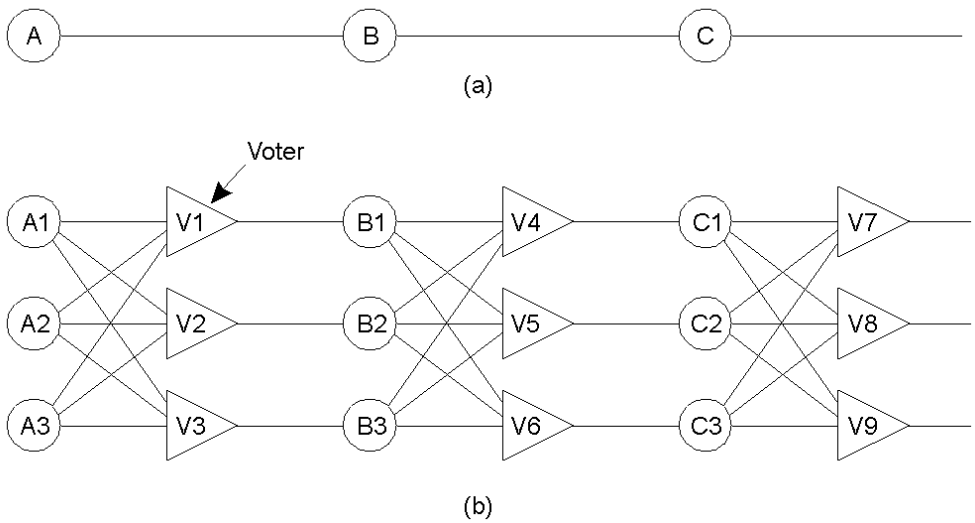


Groups and Fault Masking

- **Terminology:** When a group can mask any *k* concurrent member faults, it is said to be **k fault-tolerant**
  (*k* is called the ***degree of fault tolerance*** or the ***resilience***)
- **Question:** How large does a *k* fault-tolerant group need to be?
- Assume **crash / timing faults** => a total of ***k*+1** members are needed to survive *k* member faults (1 produces a result)
- Assume **arbitrary (Byzantine) faults**, and the group output is defined by voting *with synchronized clocks* => a total of **2*k*+1** members are needed to survive *k* member faults (*k*+1 outvote *k*)
  - **Beware: Clocks need to be kept synchronized which is NOT possible with 2*k*+1 members; for clock synchronization, we need 3*k*+1 members (famous result due to Lamport, Shostak, Pease)**
- Assume all members are identical, and they process the same inputs in the same order. Only then do we have any assurance that they will produce exactly the same outputs

Fault Masking by Redundancy

- **Triple Modular Redundancy (TMR)**
  with synchronized clocks



- TMR can be used to tolerate Byzantine faults provided that, in each case, 2 of the 3 processes (and voters) produce their results within a time bound ($2k+1 = 3$, $k = 1$ faulty, $k+1 = 2$ non-faulty)

Groups and Fault Masking

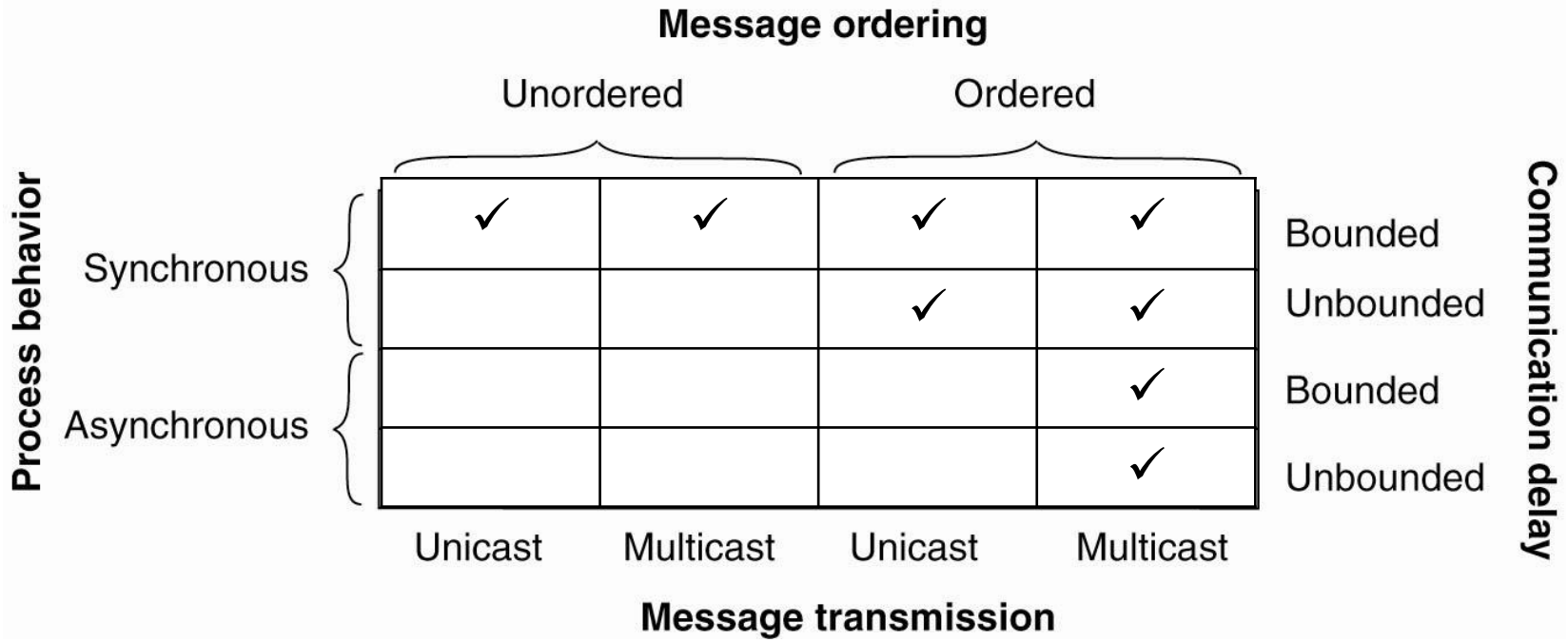
- **Assumption:** Group members are not identical, i.e., we have a distributed computation in which different members do different things
- **Problem:** Non-faulty group members must reach agreement on the same value
- Assume **arbitrary (Byzantine) faults** *without synchronized clocks* => a total of **3*k*+1** group members are needed to survive the attacks of *k* faulty members
  - **In particular, 3k+1 processes are required to synchronize clocks, e.g., to tolerate (mask) k = 1 Byzantine faulty processes, we need 3k+1 = 4 processes**
- **Essence:** We need to reach a majority vote among the group of loyalists. In the presence of *k* traitors, we need *(3k+1)–k = 2k+1* loyalists

Agreement in Faulty Systems

- Possible cases
  - **Synchronous vs. asynchronous systems**
  - **Communication delay is bounded or not**
  - **Message delivery is ordered or not**
  - **Message transmission is unicasting or multicasting**

Agreement in Faulty Systems

- Circumstances in which distributed agreement can be reached



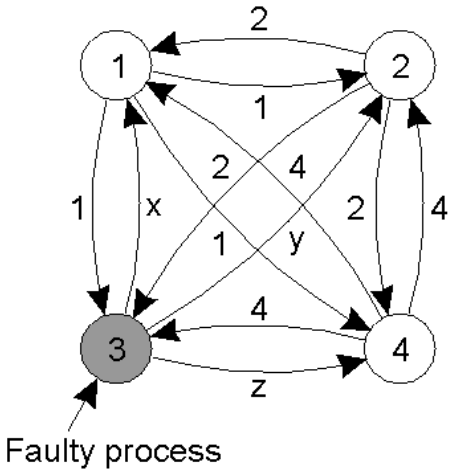
Agreement in Byzantine Faulty Systems

- The **Byzantine Agreement (Generals)** problem, with 3 loyal generals and 1 traitor, can be solved

(a) Each general sends its value to the other generals (the traitor sends different values to different generals)

(b) Each general assembles a vector based on what it received, and sends its vector to the other generals

(c) Each general receives the vectors that the other generals sent, compares them, and chooses the values where 3 of the 4 agree



(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)

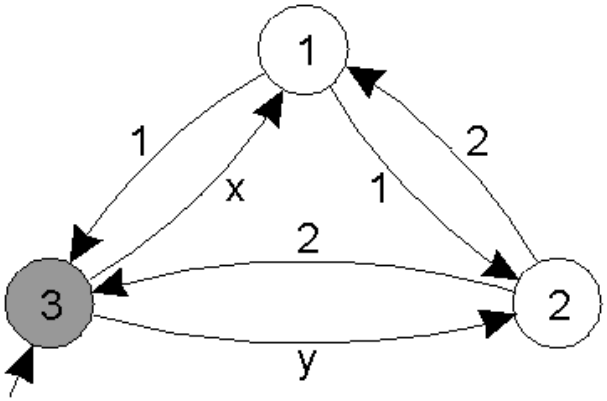
(b)

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

Agreement in Byzantine Faulty Systems

- The Byzantine Agreement problem, with only 2 loyal generals and 1 traitor, CANNOT be solved if a general chooses the values where 2 of the 3 agree
  - **The reason is that the traitor 3 could send (a, b, c) = (1, 1, x) to 1, and (d, e, f) = (2, 2, y) to 2, so 1 does not know whether 2 or 3 is lying and 2 does not know whether 1 or 3 is lying, so no agreement can be reached on any of the values**



(a)

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

(b)

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

(c)

Fault Detection

- Why do we need it?
  - **A process should be able to determine if another process is faulty**
  - **A process in a group should be able to determine if another process is *not* faulty and is still a member of the group**
- Two basic mechanisms
  - **Pull (ping or probe): A process sends "Are you alive" messages to another process to see if it is ok**
  - **Push: A process sends "I am alive" messages to another process to report that is ok**
- Typically, **fault detection timeouts** are used
  - **However, if the network is unreliable or slow, a process might not get a response before the timeout and conclude incorrectly that the process is faulty - an example of a false positive**

Reliable Communication

- **So far:** We've concentrated on *process* resilience (by means of process groups)
- What about reliable *communication channels*?
- **Error detection**
  - **Add extra bits to detect bit errors**
  - **Use sequence numbers to detect missing packets**
- **Error correction**
  - **Add so many redundant bits that corrupted packets can be automatically *corrected* at the destination**
  - **Request retransmission of lost packets or the last *N* packets**
- **Note:** Usually assumes point-to-point (rather than multicast) communication

Reliable RPC

- **What can go wrong?**
  - **(1)** Client cannot locate the server
  - **(2)** Client request is lost
  - **(3)** Server crashes
  - **(4)** Server reply is lost
  - **(5)** Client crashes
- **How do we fix it?**
  - **(1)** Relatively simple – just report back to the client
  - **(2)** Relatively simple – just resend the request

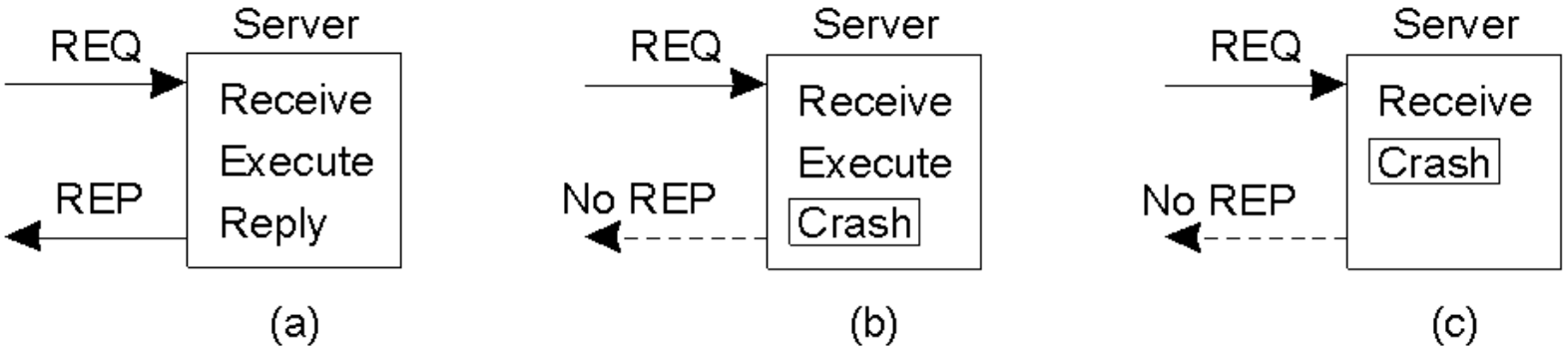
Reliable RPC

(3) Server crashes are harder, as you don't what the server had done
A server in client-server communication

(a) Normal case

(b) Crash after execution

(c) Crash before execution



We need to decide on what we expect from the server

- **At-least-once-semantics:** The server guarantees it will carry out an operation at least once, no matter what
- **At-most-once-semantics:** The server guarantees it will carry out an operation at most once

Reliable RPC

(4) Detecting a lost reply can be hard, because it might be that the server had crashed. You don't know whether the server actually carried out the operation

- **Solution: None, except that if you're lucky the operation is idempotent, i.e., repeatable without any harm done if it is carried out twice**

(5) The server is doing work and holding resources for nothing (called an **orphan** computation)

Possible Solutions:

- **Client kills (or rolls back) the orphan when the server recovers**
- **Server broadcasts new epoch number and kills the orphan when the server recovers**
- **Require orphan computation to complete within *T* time units and, if it doesn't, remove it**

Server Crashes

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

Server Crashes

- These events can happen in six different orders, in particular:
- **M →P**
  - **M →P →C: A crash occurs after the server sends the message and prints the text**
  - **M →C (→P): A crash occurs after the server sends the message, but before it prints the text**
  - **C (→M →P): A crash occurs before the server does anything**
- **P →M**
  - **P →M →C: A crash occurs after the server sends the message and prints the text**
  - **P →C (→M): The server prints the text, and then a crash occurs before the server sends the message**
  - **C (→P →M): A crash occurs before the server does anything**

Server Crashes

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

Client	Server					
	Strategy M -> P			Strategy P -> M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always (i.e., send twice)	DUP	OK	OK	DUP	DUP	OK
Never (i.e., send once)	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed (i.e., M received)	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 Multicasting

- **Basic 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 receive messages from channel *c***
- **Simple reliability:** If process *R* is in *RCV*(*c*) at the time that message *m* was submitted to channel *c* and *R* does not leave *RCV*(*c*), then *m* is delivered to *R*
- **Atomic multicast:** A message *m* submitted to channel *c* is delivered to process *R* in *RCV*(*c*) if and only if *m* is delivered to ***all*** members of *RCV*(*c*)

Reliable Multicasting

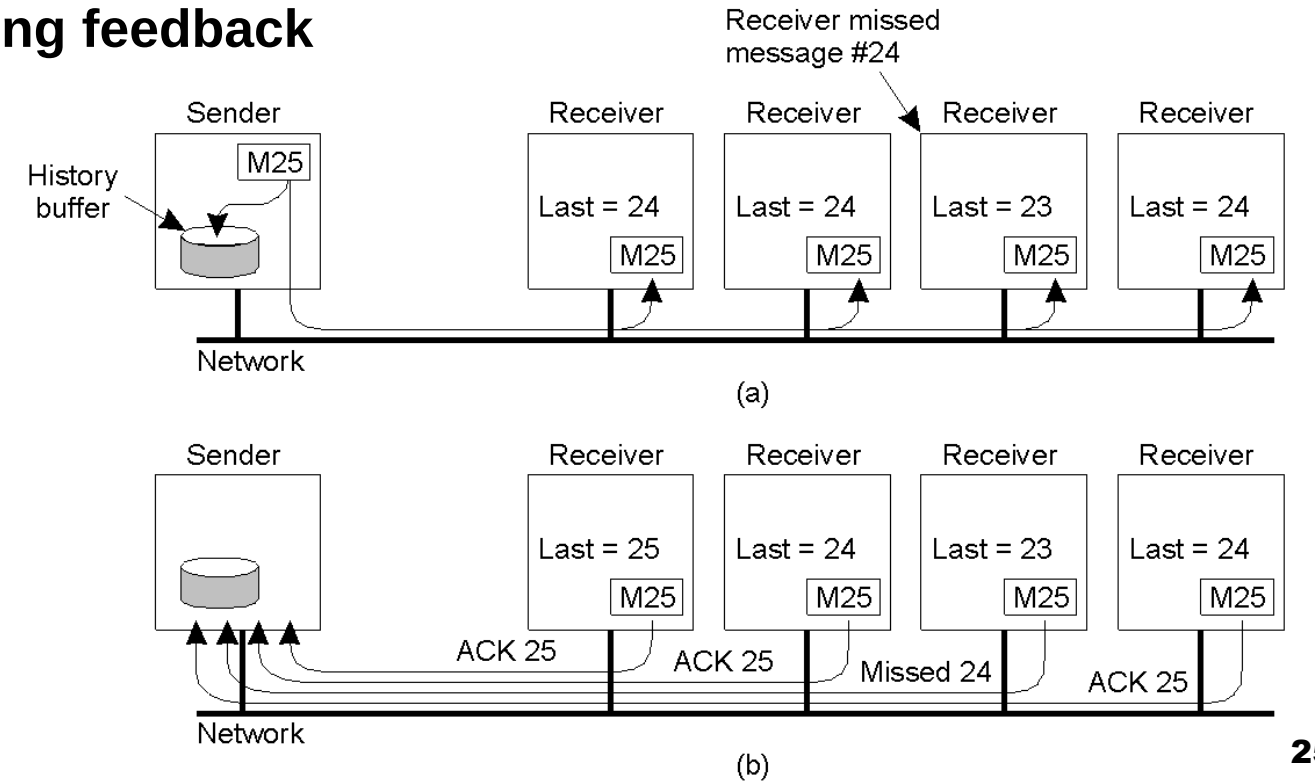
- **Observation:** On a local-area network, reliable multicasting is relatively easy (e.g., on Ethernet, use UDP multicast with extra acknowledgments and retransmissions)
- **Idea:** When *S* sends message *m*, it stores *m* in a **history (send) buffer**
- Each receiver *R* either (1) acknowledges the receipt of *m*, or (2) requests the sender *S* to retransmit *m* when *R* detects that *m* was lost
- The sender *S* removes *m* from its send buffer, when it has received acknowledgments from *all* of the intended receivers
- **Question:** Why doesn't this scale?

Reliable Multicasting

- A simple reliable multicasting scheme when all receivers are known and are assumed not to fail

(a) Transmitting messages

(b) Providing feedback

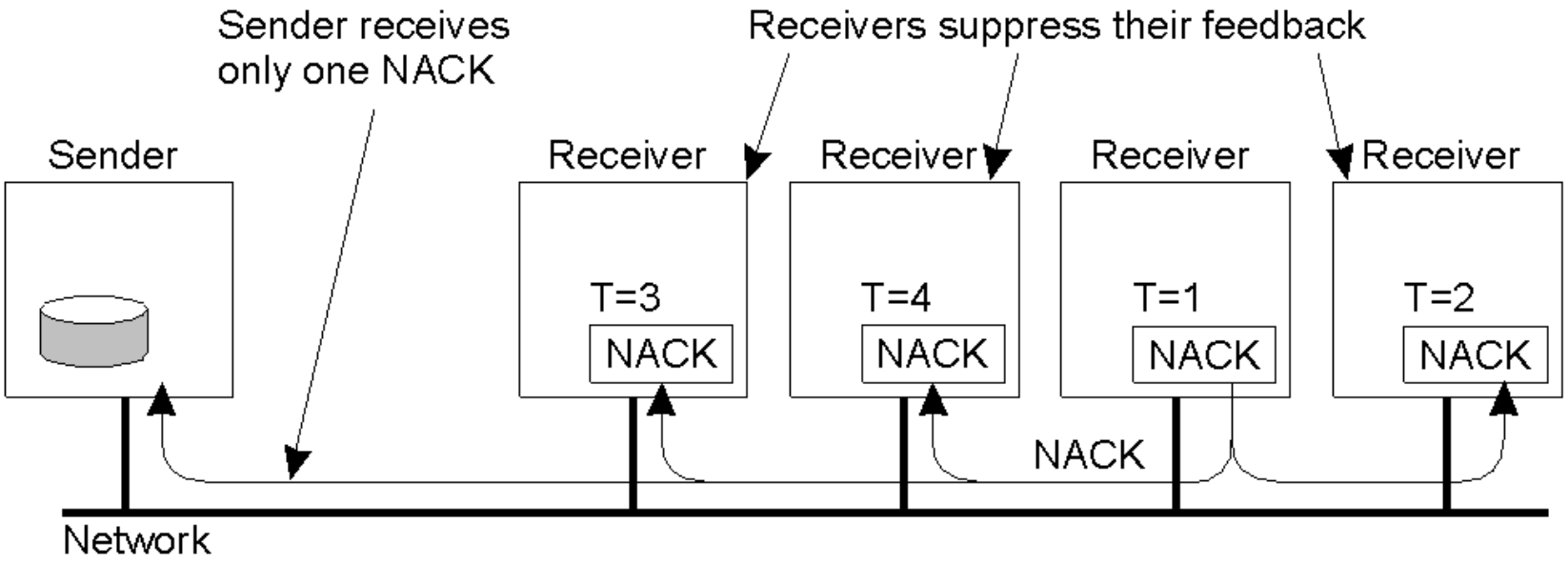


Feedback Control

- **Basic idea:** A process *P* suppresses its own feedback message, i.e., retransmission request (NACK) when it notices that another process *Q* has already sent a NACK
- **Assumptions:**
  - **All receivers listen to a common channel to which feedback messages are submitted**
  - **A process schedules its own feedback message *randomly*, and suppresses its feedback message when it observes a feedback message from another process**

Flat Feedback Control

- Several receivers need to send feedback messages, i.e., retransmission requests (NACKs), but when they hear the first NACK they suppress their own NACKs



Hierarchical Feedback Control

- The essence of hierarchical reliable multicasting
- All messages are sent to the root node, which forwards them to local coordinators
- Each local coordinator forwards the messages to its children (other nodes in its local-area network)
- Each local coordinator handles feedback messages, i.e., retransmission requests (NACKs)
- Intermediate nodes aggregate feedback messages before passing them on