ECE151 – Lecture 11

Chapter 7

Fault Tolerance

Basic Concepts

Process resilience

Reliable client-server communication

Reliable group communication

Distributed commit

Recovery

Dependability

Basics: A *component* provides *services* to *clients*. To provide services, the component may require the services from other components a component may **depend** on some other component.

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	How easy can a failed system be repaired

Note: For distributed systems, components can be either 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: build a component in such a way that it can meet its specifications in the presence of faults (i.e., **mask** the presence of faults)

Fault removal: reduce the presence, number, seriousness of faults

Fault forecasting: estimate the present number, future incidence, and the consequences of faults

Failure Models

Different types of failures.

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*	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

Crash Failures

Problem: Clients cannot distinguish between a crashed component and one that is just a bit slow

Examples: Consider a server from which a client is exepcting output:

Is the server perhaps exhibiting timing or omission failures

Is the channel between client and server faulty (crashed, or exhibiting timing or omission failures)

Fail-silent: The component exhibits omission or crash failures; clients cannot tell what went wrong

Fail-stop: The component exhibits crash failures, but its failure can be detected (either through announcement or timeouts)

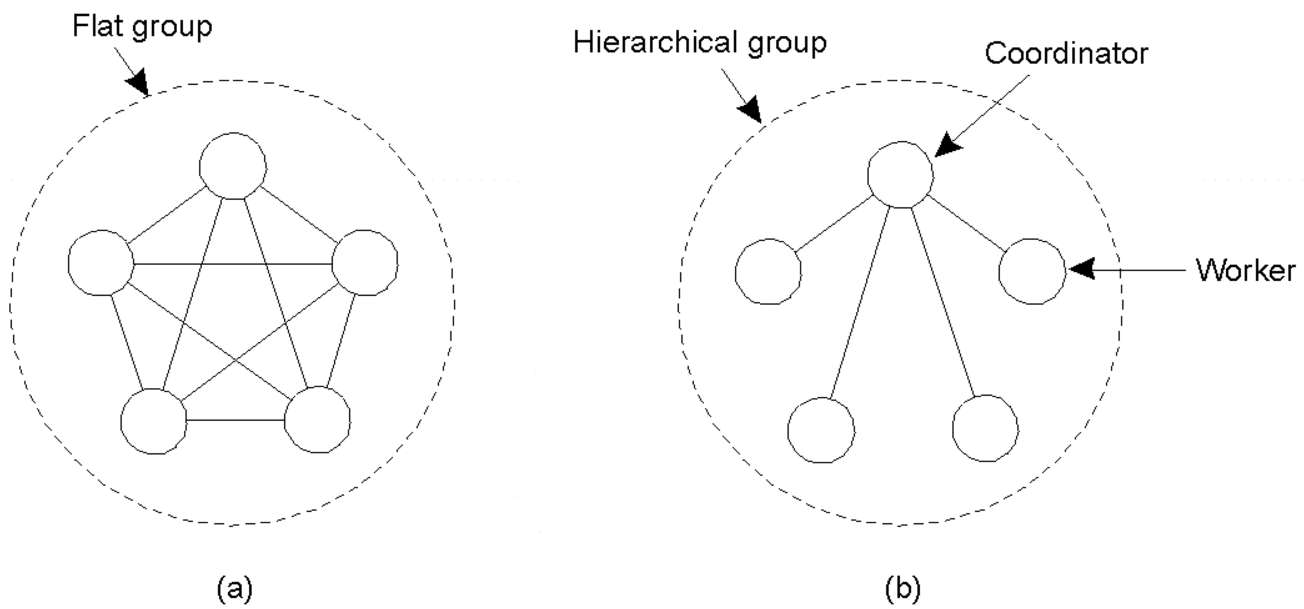
Fail-safe: The component exhibits arbitrary, but benign failures (they can’t do any harm)

Process Resilience

Basic issue: Protect yourself against faulty processes by replicating and distributing computations in a group.

Flat groups: Good for fault tolerance as information exchange immediately occurs with all group members; however, may impose more overhead as control is completely distributed (hard to implement).

Hierarchical groups: All communication through a single coordinator Not really fault tolerant and scalable, but relatively easy to implement.



Groups and Failure Masking

Terminology: when a group can mask any *k* concurrent member failures, it is said to be **k-fault tolerant** (*k* is called degree of fault tolerance or resiliance).

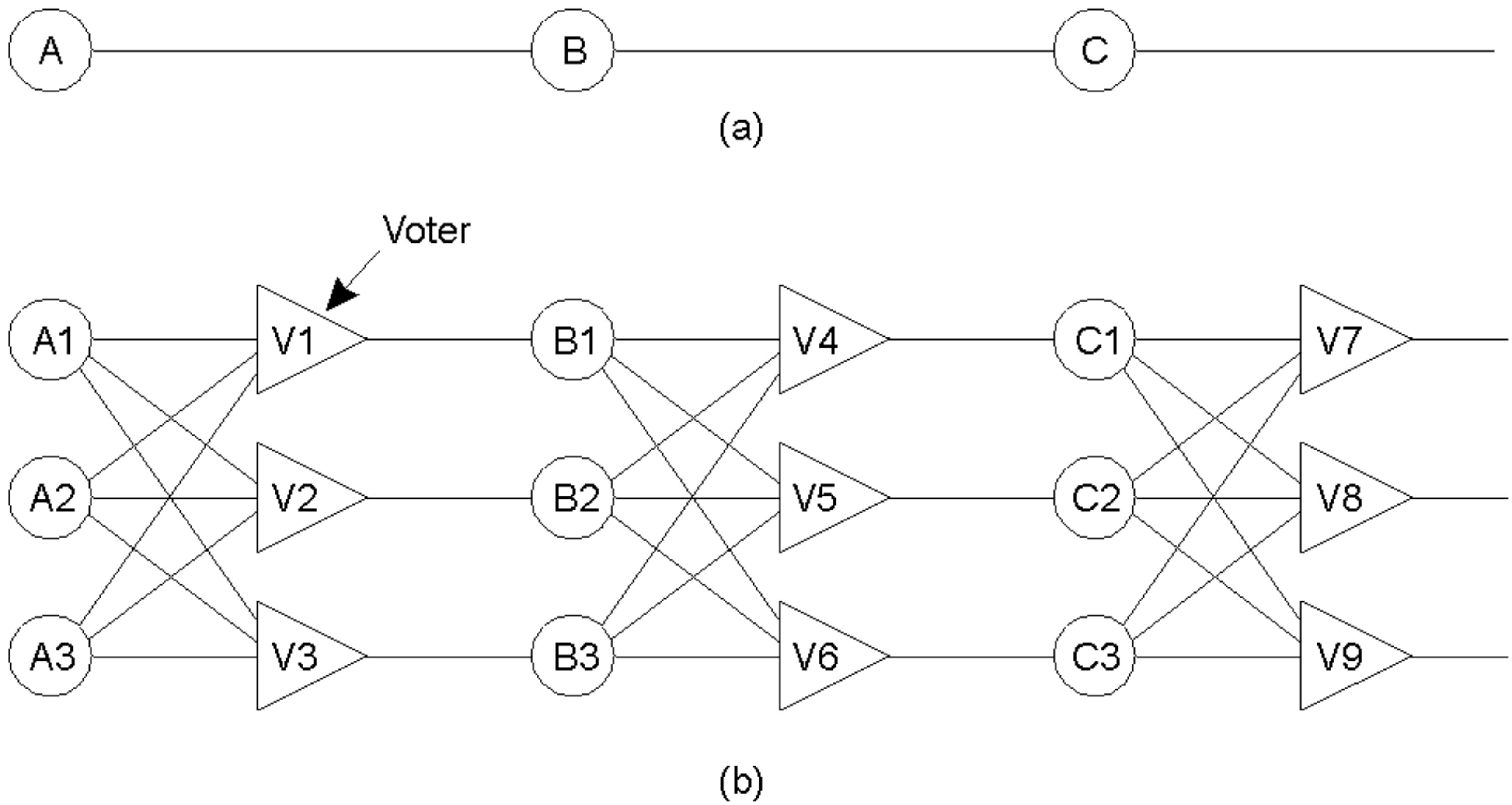
Problem: how large does a *k*-fault tolerant group need to be?

Assume crash/performance failure semantics => a total of $k+1$ members are needed to survive *k* member failures.

Assume arbitrary failure semantics, and group output defined by voting => a total of $2k+1$ members are needed to survive *k* member failures.

Assumption: all members are identical, and process all input in the same order. Only then are we sure that they do exactly the same thing.

Failure Masking by Redundancy



Groups and Failure Masking

Assumption: Group members are not identical, i.e., we have a distributed computation

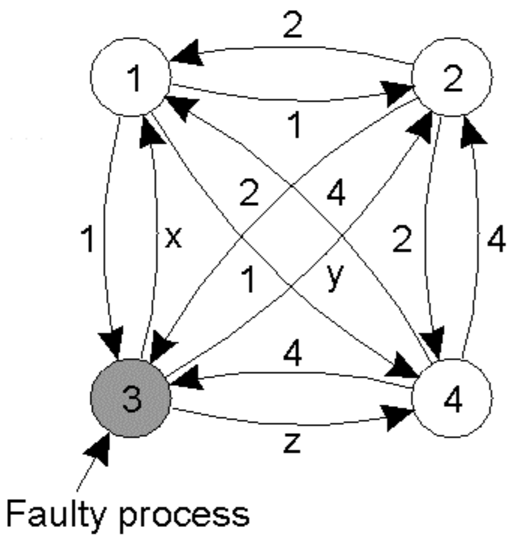
Problem: Nonfaulty group members should reach agreement on the same value

Observation: Assuming arbitrary failure semantics, we need $3k+1$ group members to survive the attacks of k faulty members

Note: This is also known as **Byzantine failures**.

Essence: We are trying to reach a majority vote among the group of loyalists, in the presence of k traitors need $2k+1$ loyalists.

Agreement in Faulty Systems



(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)

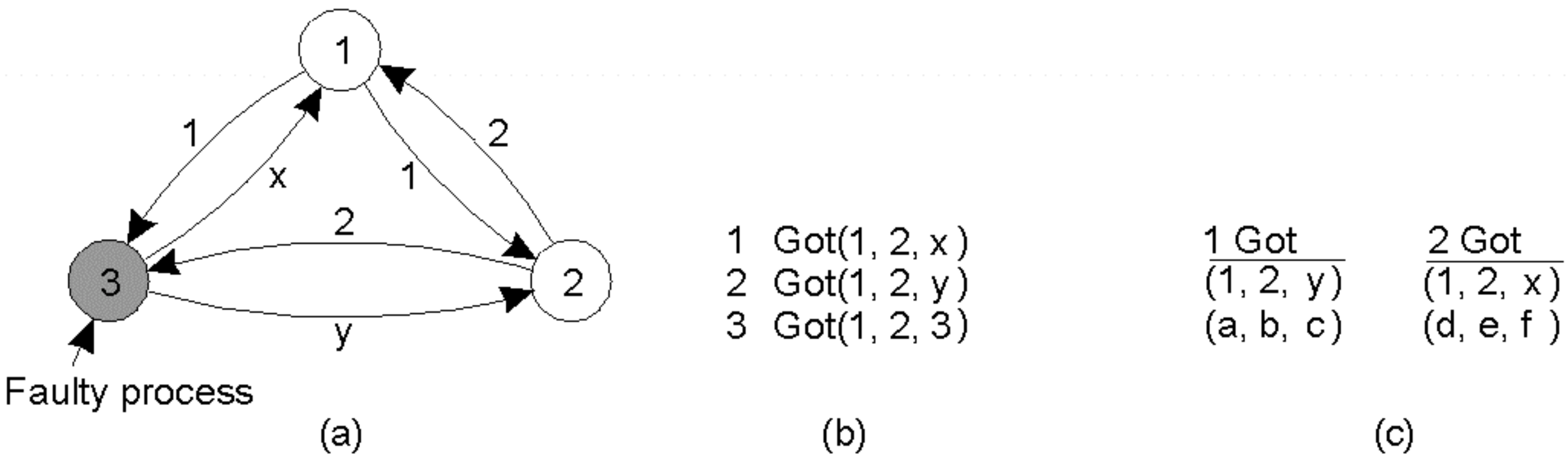
The Byzantine generals problem for 3 loyal generals and1 traitor.

a) The generals announce their troop strengths (in units of 1 kilosoldiers).

b) The vectors that each general assembles based on (a)

c) The vectors that each general receives in step 3.

Agreement in Faulty Systems



The same as in previous slide, except now with 2 loyal generals and one traitor.

Reliable Communication

So far: Concentrated on **process resilience** (by means of process groups). What about reliable communication channels?

Error detection:

Framing of packets to allow for bit error detection

Use of frame numbering to detect packet loss

Error correction:

Add so much redundancy that corrupted packets can be automatically *corrected*

Request retransmission of lost, or last *N* packets

Observation: Most of this work assumes point-to-point communication

Reliable RPC

What can go wrong?:

1: Client cannot locate server

2: Client request is lost

3: Server crashes

4: Server response is lost

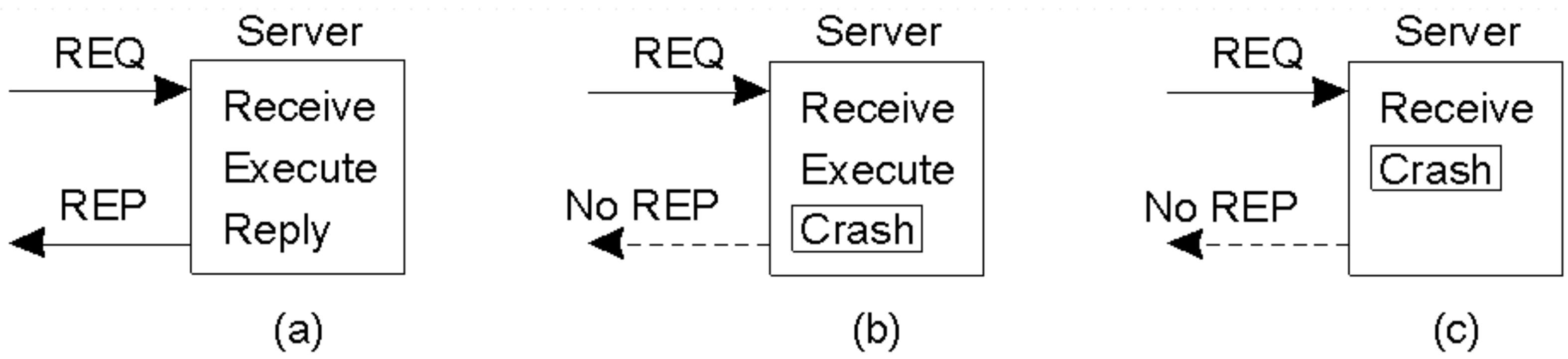
5: Client crashes

[1:] Relatively simple – just report back to client

[2:] Just resend message

Server Crashes

[3:] Server crashes are harder as you don't what it had already 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 lost replies can be hard, because it can also be that the server had crashed. You don't know whether the server has carried out the operation

Solution: None, except that you can try to make your operations **idempotent**: repeatable without any harm done if it happened to be carried out before.

[5:] **Problem:** The server is doing work and holding resources for nothing (called doing an **orphan** computation).

Orphan is killed (or rolled back) by client when it reboots

Broadcast new epoch number when recovering servers kill orphans

Require computations to complete in a *T* time units. Old ones are simply removed.

Server Crashes

Client	Server					
	Strategy M -> P			Strategy P -> M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

Different combinations of client and server strategies in the presence of server crashes.

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 can receive messages from channel *c*

Simple reliability: If process *P* is in *RCV*(*c*) at the time message *m* was submitted to *c* and *P* does not leave *RCV*(*c*), *m* should be delivered to *P*

Atomic multicast: How can we ensure that a message *m* submitted to channel *c* is delivered to process *P* in *RCV*(*c*) only if *m* is delivered to *all* members of *RCV*(*c*)

Reliable Multicasting

Observation: If we can stick to a local-area network, reliable multicasting is "easy"

Principle: Let the sender log messages submitted to channel *c*:

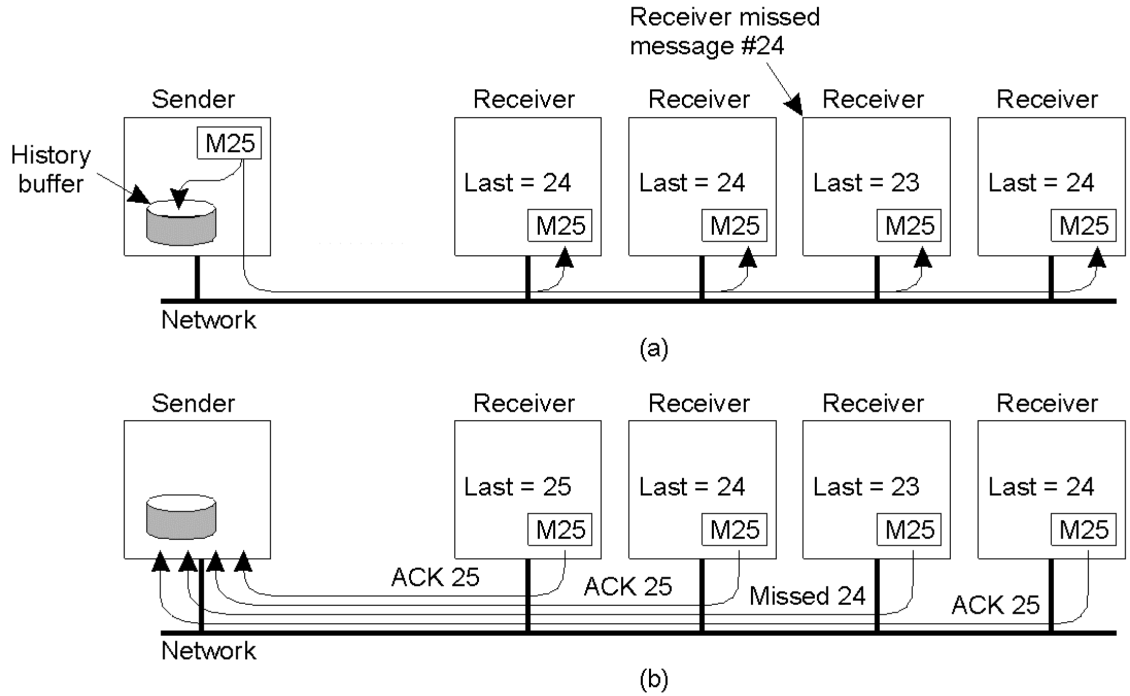
If *P* sends message *m*, *m* is stored in a **history buffer**

Each receiver acknowledges the receipt of *m*, or requests retransmission by *P* when the receiver notices that a message was lost

Sender *P* removes *m* from history buffer when everyone has acknowledged receipt

Question: Why doesn't this scale?

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

Feedback Suppression

Basic idea: Let a process *P* suppress its own feedback when it notices another process *Q* is already asking for a retransmission

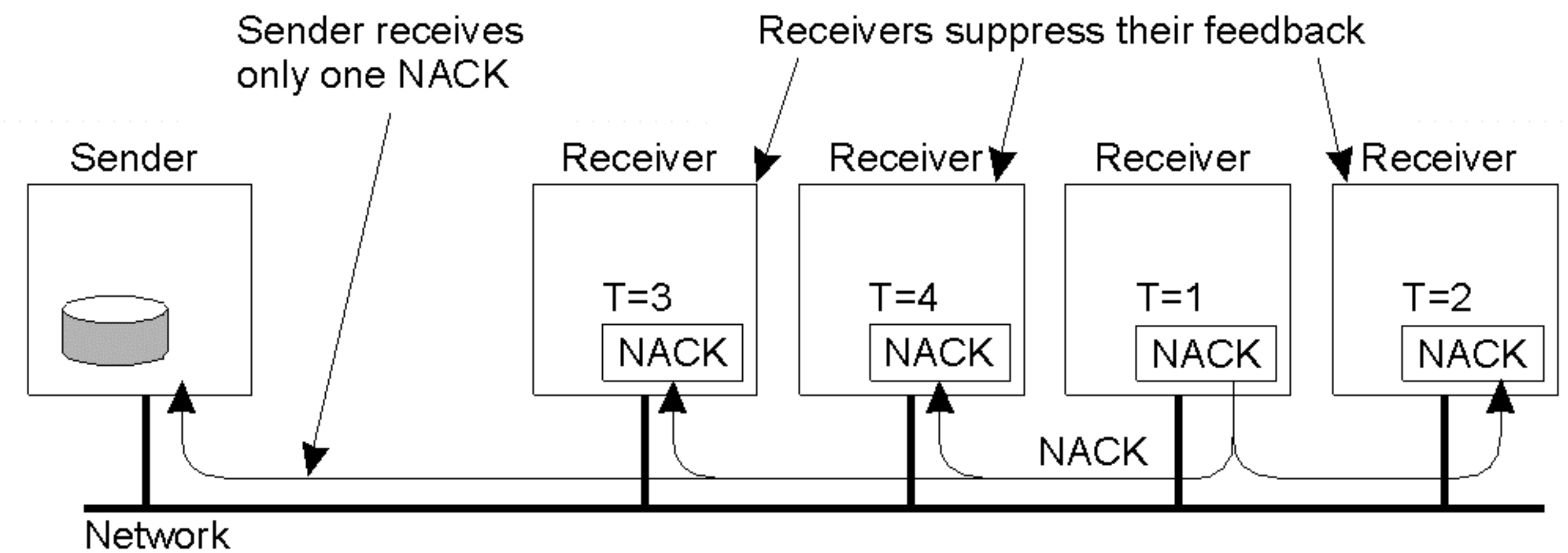
Assumptions:

All receivers listen to a common **feedback channel** to which feedback messages are submitted

Process *P* schedules its own feedback message *randomly*, and suppresses it when observing another feedback message

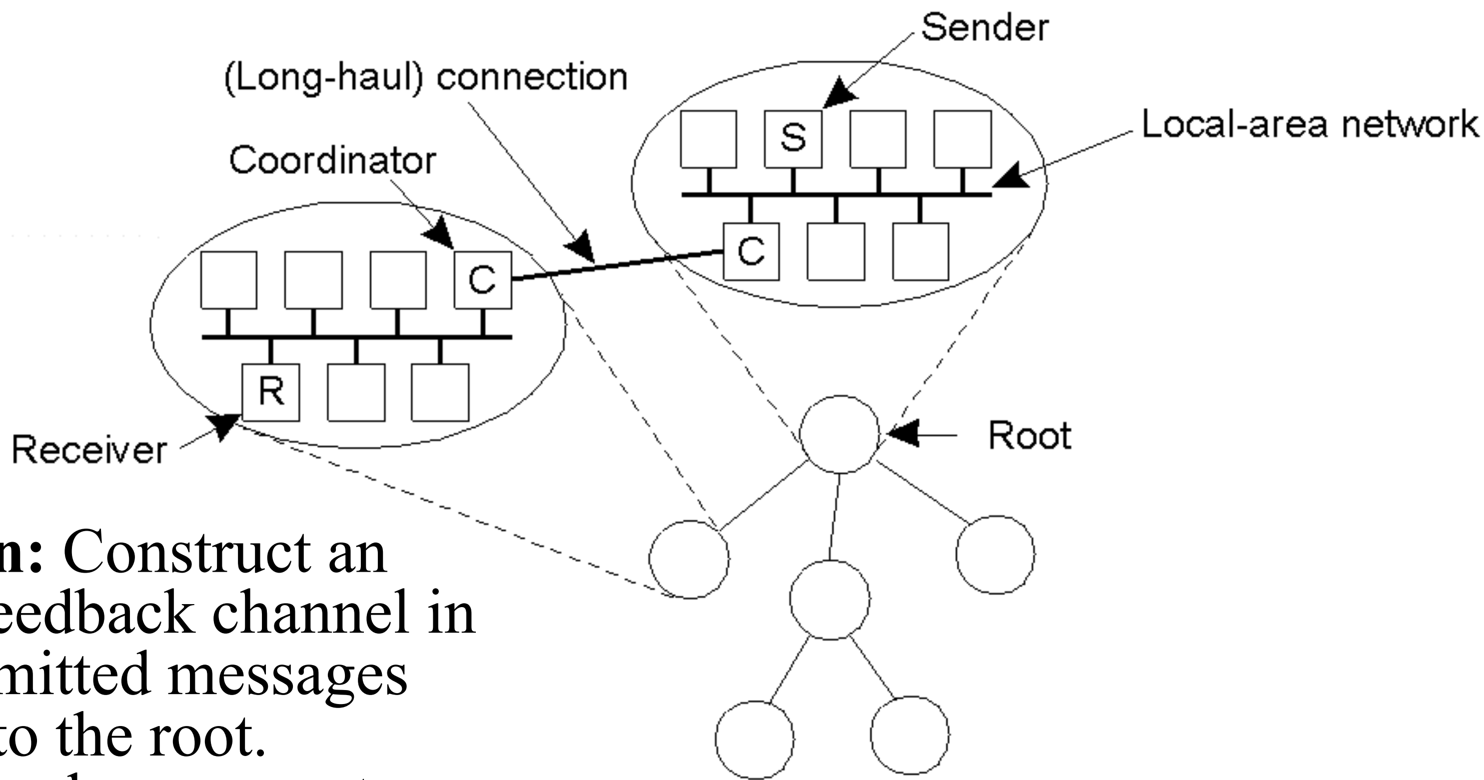
Question: Why is the random schedule so important?

Feedback Suppression



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

Hierarchical Feedback Control



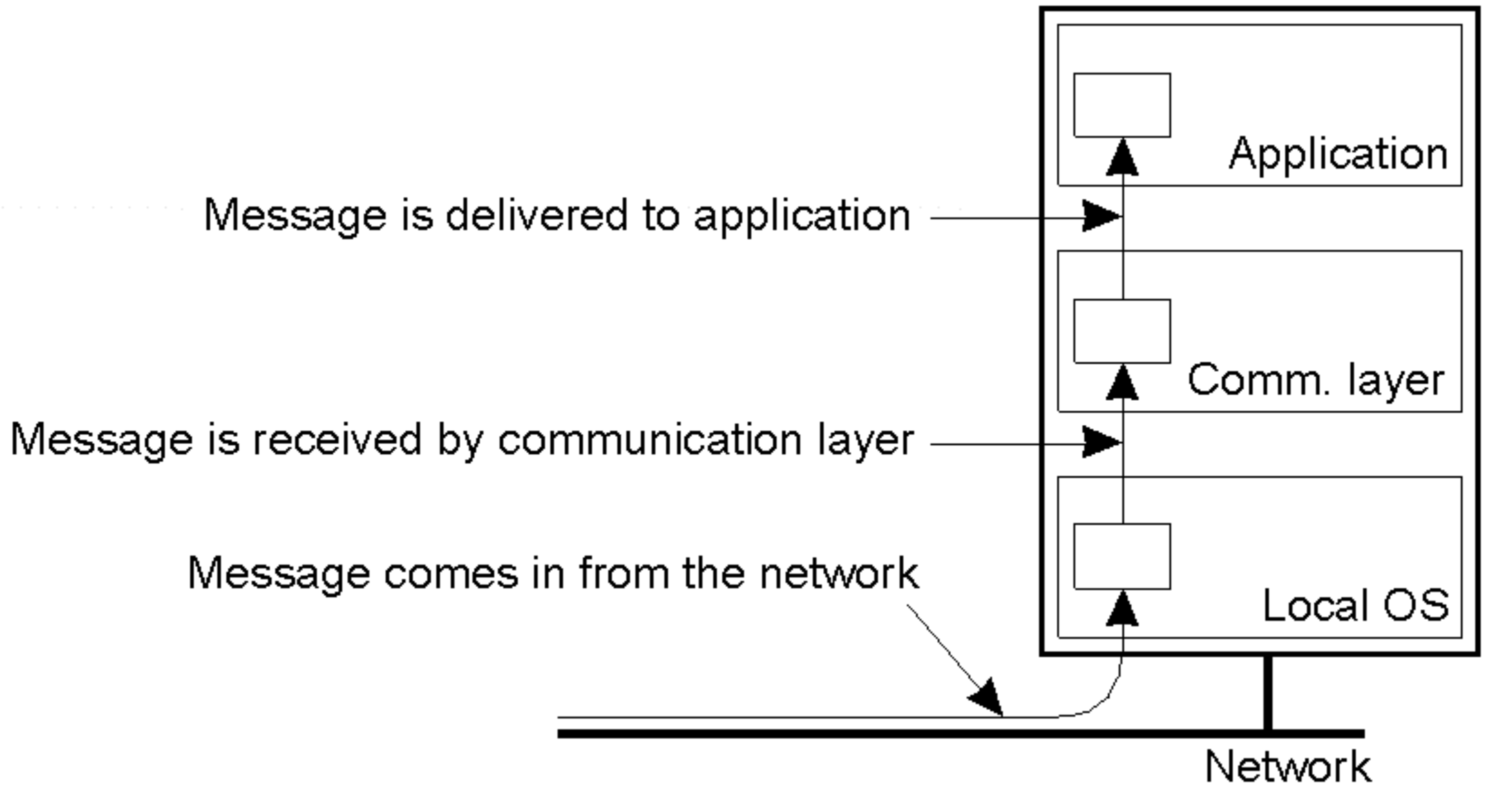
Basic solution: Construct an hierarchical feedback channel in which all submitted messages are sent only to the root. Intermediate nodes aggregate feedback messages before passing them on.

The essence of hierarchical reliable multicasting.

a) Each local coordinator forwards the message to its children.

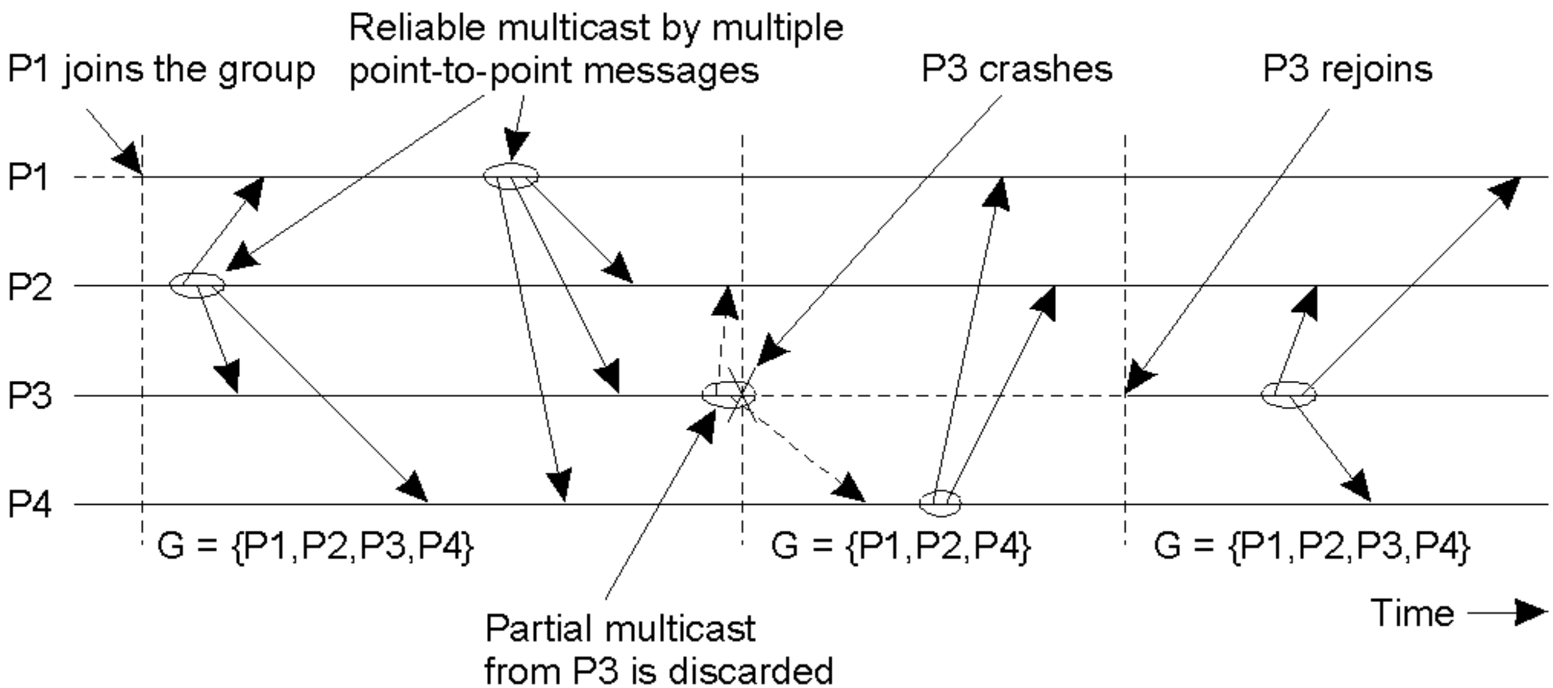
b) A local coordinator handles retransmission requests.

Virtual Synchrony



Idea: Formulate reliable multicasting in the presence of process failures in terms of process groups and changes to group membership:

Virtual Synchrony



The principle of virtual synchronous multicast.

Guarantee: A message is delivered only to the nonfaulty members of the current group. All members should agree on the current group membership.

Message Ordering

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

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

Message Ordering

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

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

Virtual Synchrony

Essence: We consider **views** $V \subseteq RCV(c) \cup SND(c)$

Processes are added or deleted from a view V through **view changes** to V^*; a view change is executed *locally* by each $P \in V \cap V^*$

(1) For each consistent state, there is a **unique view** on which all its members agree. **Note:** implies that all nonfaulty processes see all view changes in the same order

(2) If message *m* is sent to V before a view change *vc* to V^*, then either all $P \in V$ that excute *vc* receive *m*, or no processes $P \in V$ that execute *vc* receive *m*. **Note:** all nonfaulty members in the same view get to see the same set of multicast messages.

(3) A message sent to view V can be delivered only to processes in V, and is discarded by successive views

A reliable multicast algorithm satisfying (1)–(3) is **virtually synchronous**

Virtual Synchrony

A sender to a view V need not be member of V

If a sender $S \in V$ crashes, its multicast message m is *flushed* before S is removed from V: m will never be delivered after the point that $S \notin V$

Note: Messages from S may still be delivered to all, or none (nonfaulty) processes in V before they all agree on a new view to which S does not belong

If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V. Or we may decide to deliver m to members in V - P

Observation: Virtually synchronous behavior can be seen independent from the ordering of message delivery.

The only issue is that messages are delivered to an *agreed upon* group of receivers.

Virtual Synchrony Implementation

- The current view is known at each P by means of a delivery list DEST[P]
- If $P \in$ DEST[P] then $Q \in$ DEST[P]
- Messages received by P are queued in QUEUE[P]
- If P fails, the group view must change, but not before all messages from P have been flushed
- Each P attaches a (stepwise increasing) **timestamp** with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in RCVD[P]
- The vector RCVD[P] is sent (as a control message) to all members in DEST[P]
- Each P records RCVD[P] in REMOTE[P][Q]

Virtual Synchrony Implementation

Observation: REMOTE[*P*][*Q*] shows what *P* knows about message arrival at *Q*

1	2 3 1 5
2	2 2 2 4
3	3 1 4 5
4	4 2 2 4
min	2 1 1 4

A message is **stable** if it has been received by all *Q* (shown as the **min** vector)

Stable messages can be delivered to the next layer (which may deal with ordering). **Note:** Causal message delivery is free

As soon as all messages from the faulty process have been flushed, that process can be removed from the (local) views

Virtual Synchrony Implementation

Remains: What if a sender *P* failed and not all its messages made it to the nonfaulty members of the current view?

Solution: Select a coordinator which has all (unstable) messages from *P*, and forward those to the other group members.

Note: Member failure is assumed to be detected and subsequently multicast to the current view as a view change. That view change will not be carried out before all messages in the current view have been delivered.

Implementing Virtual Synchrony

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