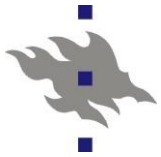
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

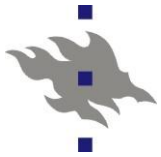
Chapter 5: Distributed Systems: Fault Tolerance

Sini Ruohomaa
Fall 2012
Slides joint work with Jussi Kangasharju et al.



Chapter Outline

- Fault tolerance
- Process resilience
- Reliable group communication
- Distributed commit
- Recovery



Basic Concepts

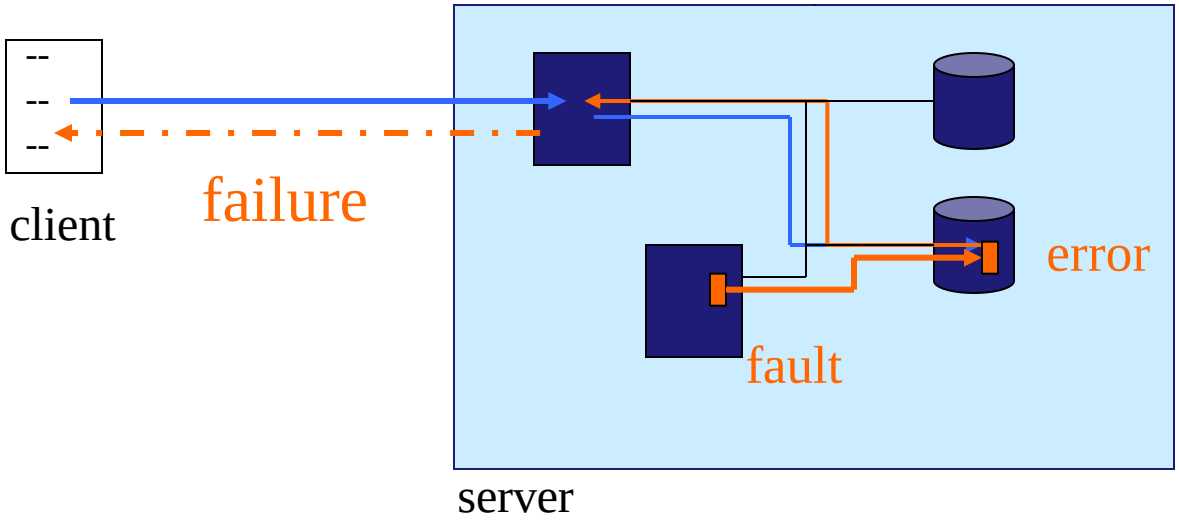
Fault tolerance – for building dependable systems

Dependability includes

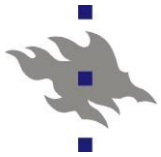
- Availability – system can be used immediately
- Reliability – runs continuously without failure
- Safety – failures do not lead to disaster
- Maintainability – recovery from failure is easy

Note: security is a separate issue from these.

Basic concepts: Fault, error, failure



- Fault: e.g. bad design/bug/physical limitation (causes error)
- Error: system state is incorrect (may lead to failure)
- Failure: cannot meet promises (e.g. full delivery of service)



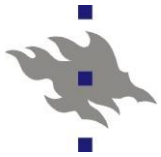
Fault Tolerance

- Faults can be
  - Transient (disappear)
  - Intermittent (disappear and reappear)
  - Permanent (persists until faulty component replaced)
- Detection
  - Which component? What went wrong?
- Recovery
  - Depends on failure, more complicated to do “blindly”
  - Mask the error OR
  - Fail predictably
- Designer’s point of view:
  - Possible failure types?
  - Recovery actions for the types?
- Failure models: a taxonomy
  - Classify failures to have better idea of their severity

Failure models: Different types of failures

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	The server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure (= Byzantine failure)	A server may produce arbitrary responses at arbitrary times

Crash: fail-stop, fail-safe (*detectable*), fail-silent (*seems to have crashed*)

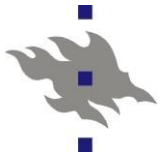


Failure Masking (1)

Principle: Detect and recover without user assistance.

Detection

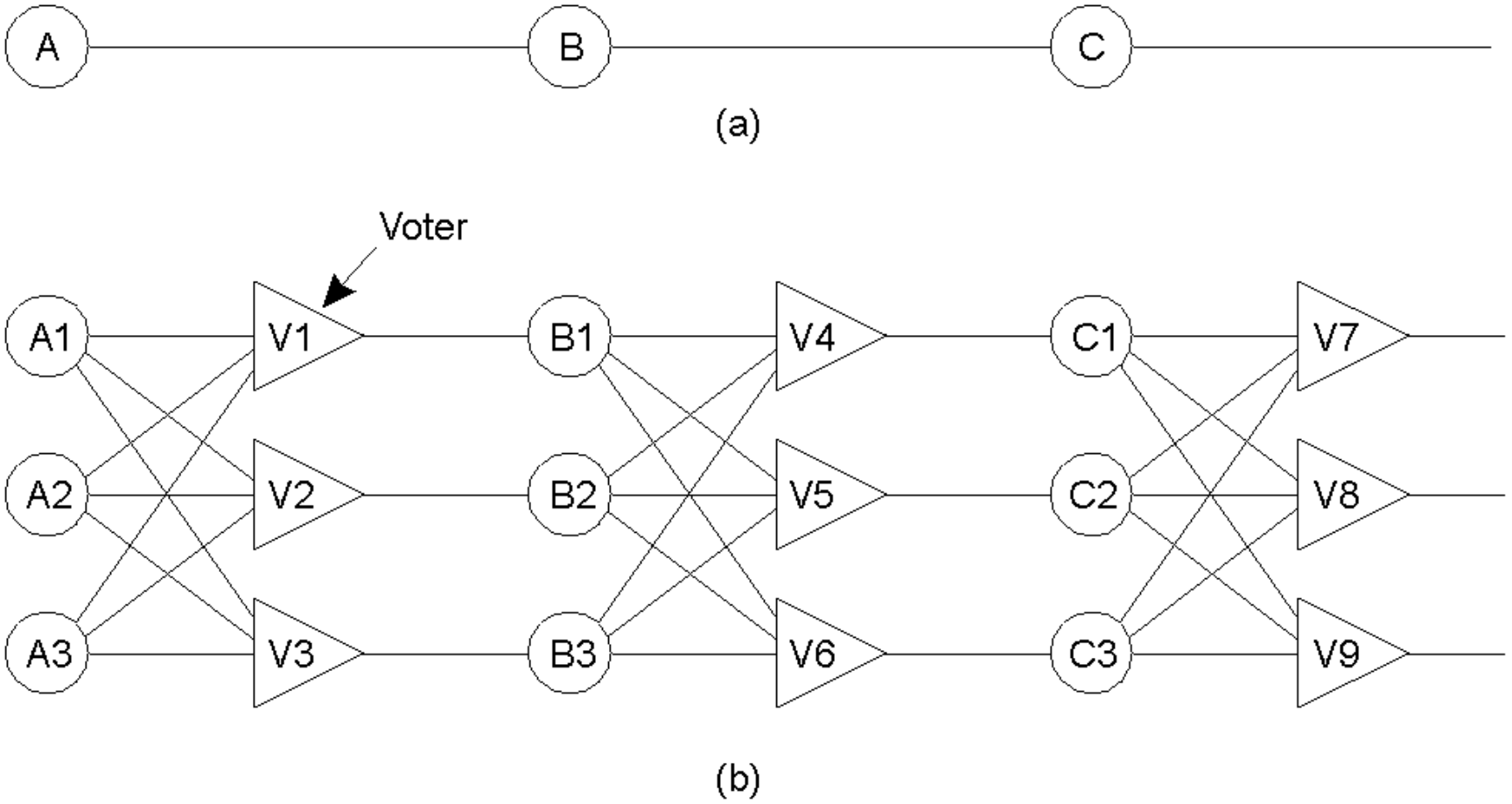
- Redundant information
  - Error detecting codes (parity, checksums)
  - Replicas
- Redundant processing
  - Groupwork and comparison
- Control functions
  - Timers
  - Acknowledgements



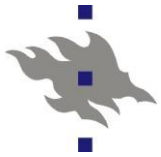
Failure Masking (2)

- RecoveryRedundant information
  - Error correcting codes
  - Replicas
- Redundant processing
  - *Time redundancy: do again if need be*
    - Retry after timeout
    - Recomputation (from checkpoint / with log)
  - *Physical redundancy: more nodes do it*
    - Repeated work and voting
    - Hot and cold standby nodes

Example: Physical Redundancy



Triple modular redundancy in an electric circuit.



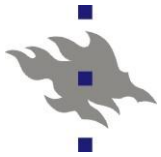
Failure Masking: Implementation (3)

- Failure models vs. implementation issues:

  We may assume the (sub-)system belongs to a given class

  => Certain failures do not occur

  => Easier detection & recovery
- Next, we use these building blocks on achieving
  - Process resilience: protecting against process failures
  - Reliable communication: protecting against channel failures
  - Recovery: backward (to known good state) or forward
  - Durable agreement on transactions: Two-phase commit



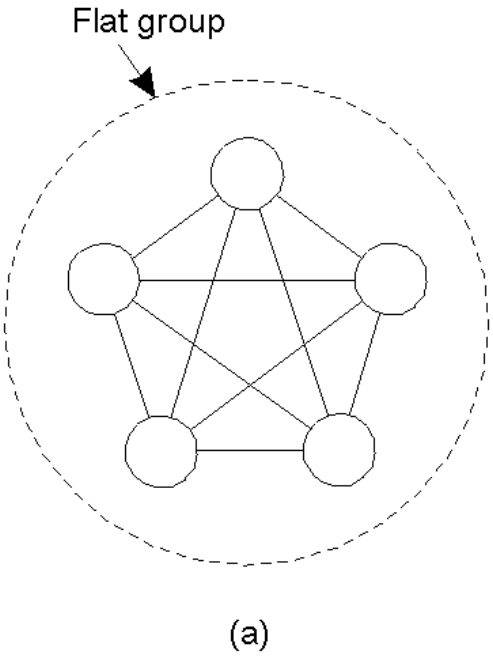
Process Resilience (1)

- Redundant processing with groups
  - Tightly synchronized
    - Flat (“unstructured”) group: voting
    - Hierarchical group: more coordinated

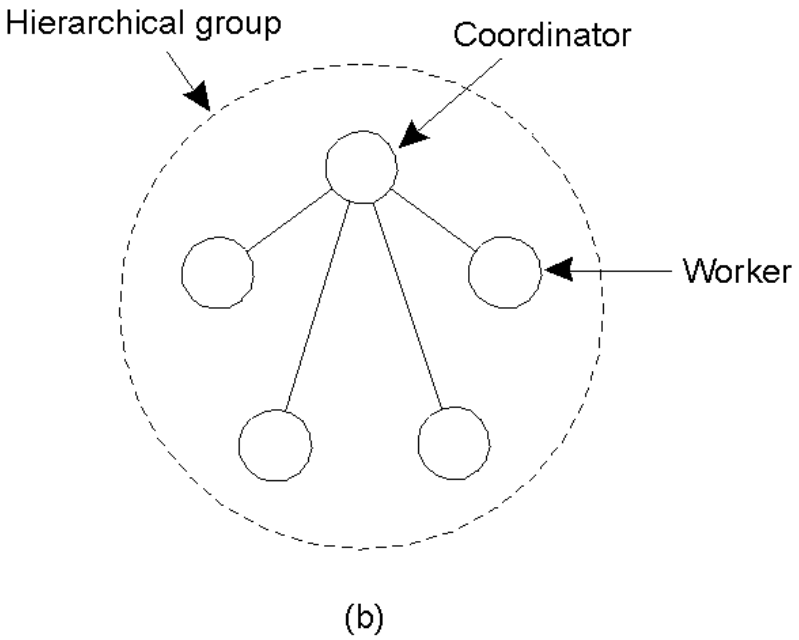
      A **primary** and a **hot standby** (execution-level synchrony)
  - Loosely synchronized
    - Hierarchical group:

      A **primary** and a **cold standby** (checkpoint, log)

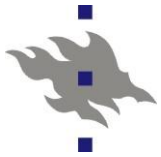
Flat and Hierarchical Groups (1)



Communication in a flat group.

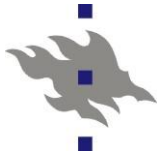


Communication in a simple hierarchical group



Flat and Hierarchical Groups (2)

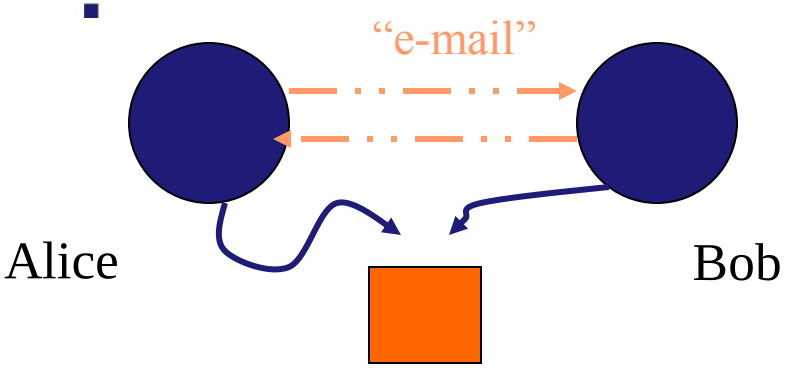
- Flat groups
  - Symmetrical
  - No single point of failure
  - Complicated decision making
- Hierarchical groups
  - The opposite properties: structure, single points of failure but simple decision-making
- Group management by group server OR distributed
- Issues
  - Join, leave (must synchronize with data messages);
  - Crash (*no notification*)



Process Groups

- Communication vs. management
  - Application communication: message passing
  - Group management: message passing
  - Synchronization requirement:
    each group communication operation in a stable group
- Failure masking
  - **k fault tolerant**: tolerates k faulty members
    - Fail-silent: k + 1 components needed (k clean crashes)
    - Byzantine: 2k + 1 components needed (k arbitrary data)
  - A precondition: **atomic multicast**: requests arrive to servers in same order
  - In practice: the probability of a failure must be “small enough”

Agreement in Faulty Systems: Communication (1)



La Tryste

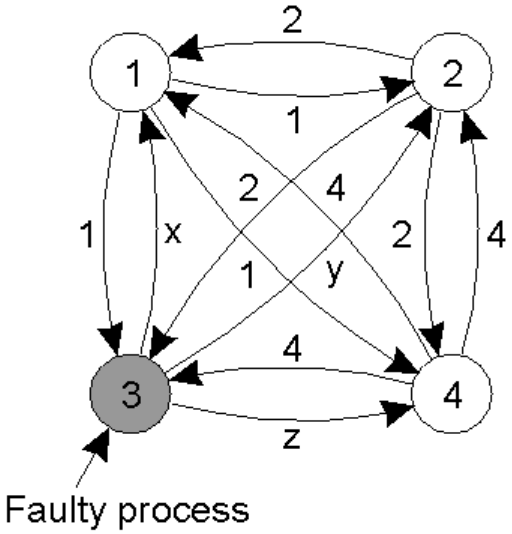
on a rainy day …

We require:
- an agreement formed
- within a bounded time

Faulty data communication: no agreement possible

Alice -> Bob *Let's meet at noon in front of La Tryste …*
Alice <- Bob *OK!!*
Alice: *If Bob doesn't know that I received his message, he will not come …*
Alice -> Bob *I received your message, so it's OK.*
Bob: *If Alice doesn't know that I received her message, she will not come …*
…

Agreement in Faulty Systems: Byzantine failure (2)



(a)

Reliable data communication, unreliable nodes

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)

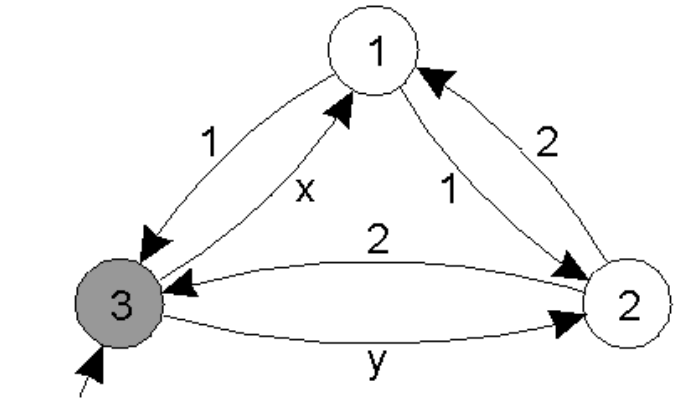
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The Byzantine generals problem for 3 loyal generals and 1 traitor.

a) The generals announce their troop strengths (in units of 1 kilosoldiers).
b) The vectors that each general assembles based on (a)
c) The vectors that each general receives in step 3.

Agreement in Faulty Systems: Byzantine (3)



(a)

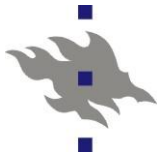
1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

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

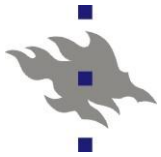
(c)

The same as in previous slide, except now with 2 loyal generals and one traitor: 3 successfully prevents agreement between 1 and 2.



Reliable Group Communication

- Lower-level data communication support varies:
  - Unreliable multicast (LAN)
  - Reliable point-to-point channels
  - Unreliable point-to-point channels
- Group communication
  - Individual point-to-point message passing
  - Implemented in middleware or in application
- Reliability
  - ACKs: catch lost messages, lost members
  - Membership changes in mid-multicast?
  - Consistency – message ordering?



Reliability of Group Communication?

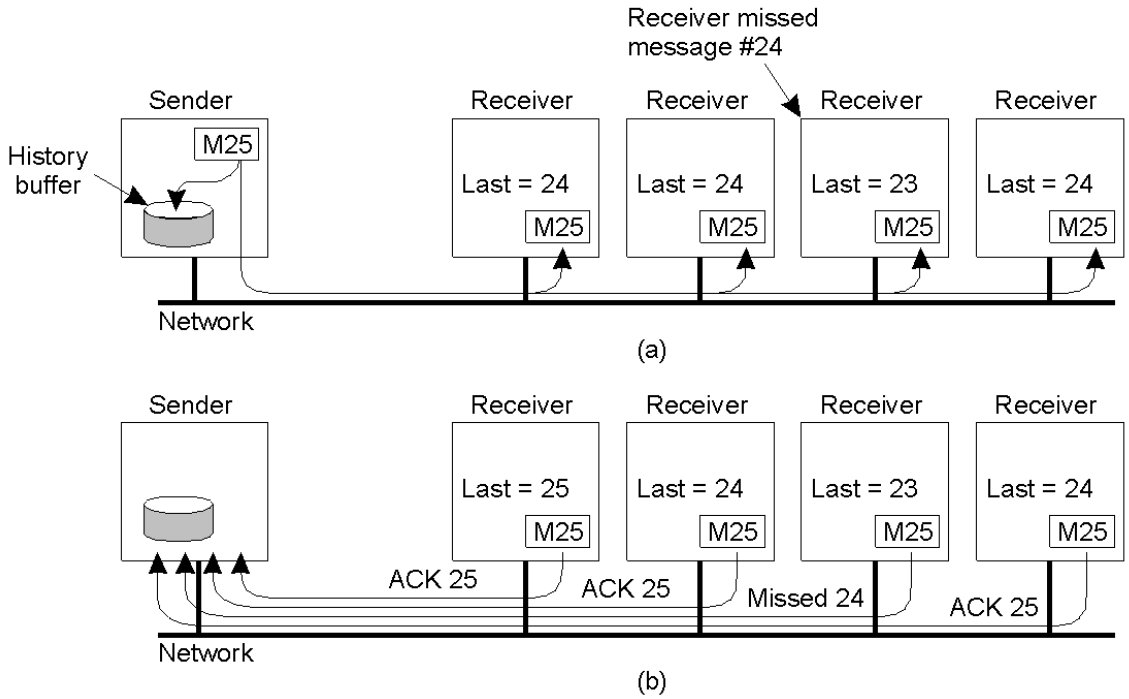
- A sent message is received by all members

  ("ACKs from all => ok!")
- Problem: during a multicast operation
  - An old member disappears from the group
  - A new member joins the group
- Solution
  - Membership changes synchronize multicasting

  => During a multicast operation no membership changes

  An additional problem: the sender disappears (remember: multicast ~ for (all P_i in G) {send m to P_i})

Basic Reliable-Multicasting Scheme



Message transmission

Reporting feedback

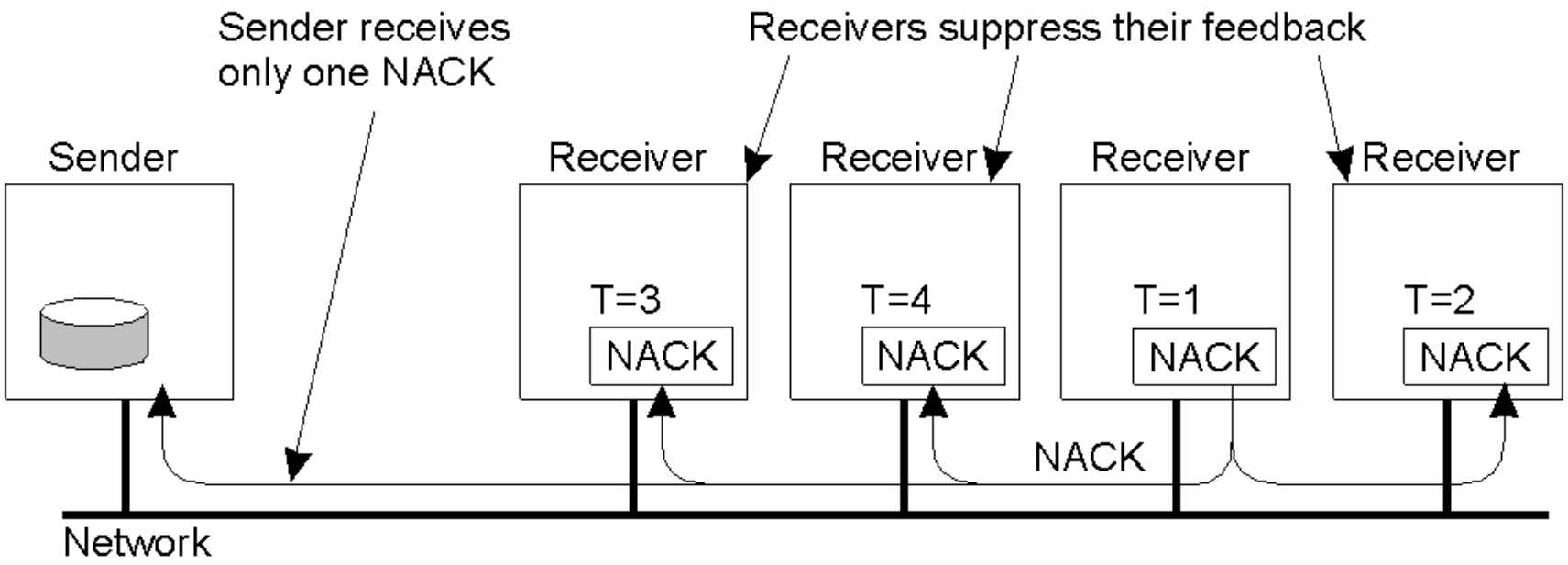
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail

Scalability?

Feedback implosion !

Scalability: Feedback Suppression

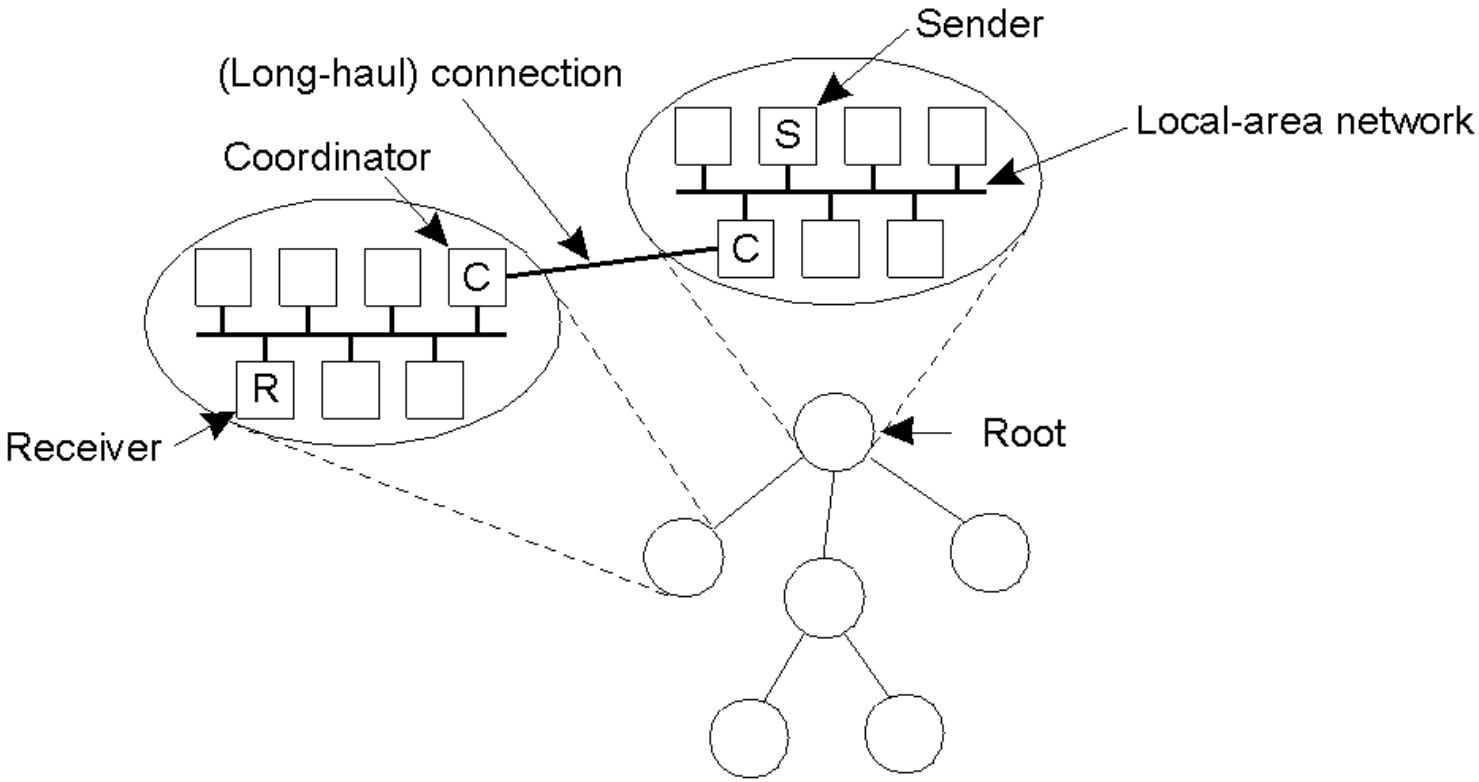
1. Never acknowledge successful delivery.



2. Multicast negative acknowledgements – suppress redundant NACKs

Problem: detection of lost messages and lost group members

Hierarchical Feedback Control

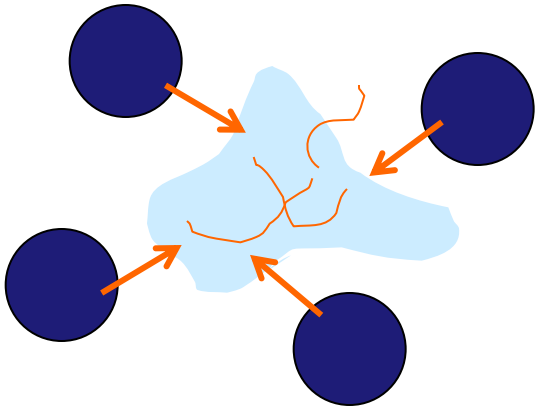


The essence of hierarchical reliable multicasting.

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

b) A local coordinator handles retransmission requests.

Recall: Basic Multicast



Guarantee:

the message will eventually be delivered to all member of the group (during the multicast: a fixed membership)

Group view: $G = \{p_i\}$

"delivery list"

G: group, **m**: message, $\mathbf{p_i}$: a member of the group

Implementation of *Basic_multicast(G, m)* :

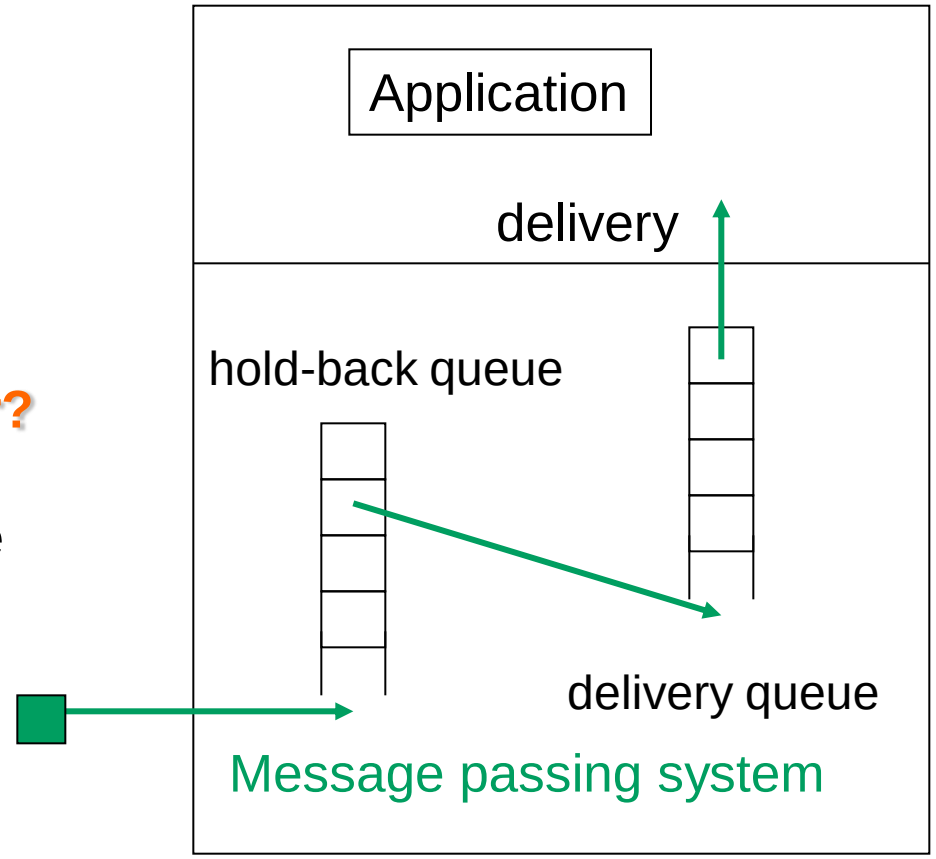
1. for each p_i in G: *send*(p_i,*m*) (a reliable one-to-one send)
2. on *receive(m)* at p_i : *deliver(m)* at p_i

Recall: Message Delivery

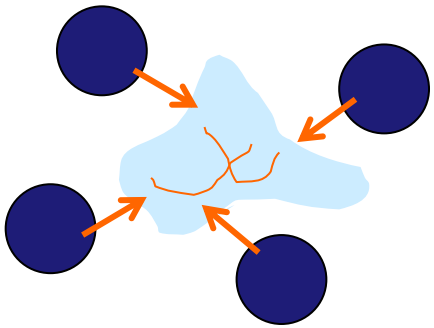
Delivery of messages

- new message => HBQ
- decision making
  - delivery order
  - **deliver or not to deliver?**

- the message is allowed to be delivered: HBQ => DQ
- when at the head of DQ: message => application (application: *receive* …)



Reliable Multicast and Group Changes



Assume

- reliable point-to-point communication
- group G={p_i}: each p_i has group view

Reliable_multicast (G, m):

if message m is delivered to one in G,
then it is delivered to all in G

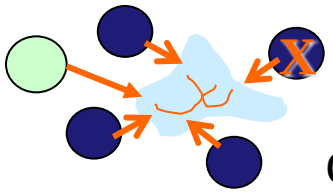
- Group change (join, leave) => change of group view
- Change of group view: update as a multicast *vc (view change)*
- *What about **concurrent group change and multicast**:*

  concurrent messages ***m*** and ***vc*** *?*

***Want virtual synchrony**:*

***all** nonfaulty processes see **m** and **vc** in the same order*

Virtually Synchronous Reliable MC (1)



Group change: $G_i = G_{i+1}$

Virtual synchrony: “all” processes see m and vc in the same order – options:

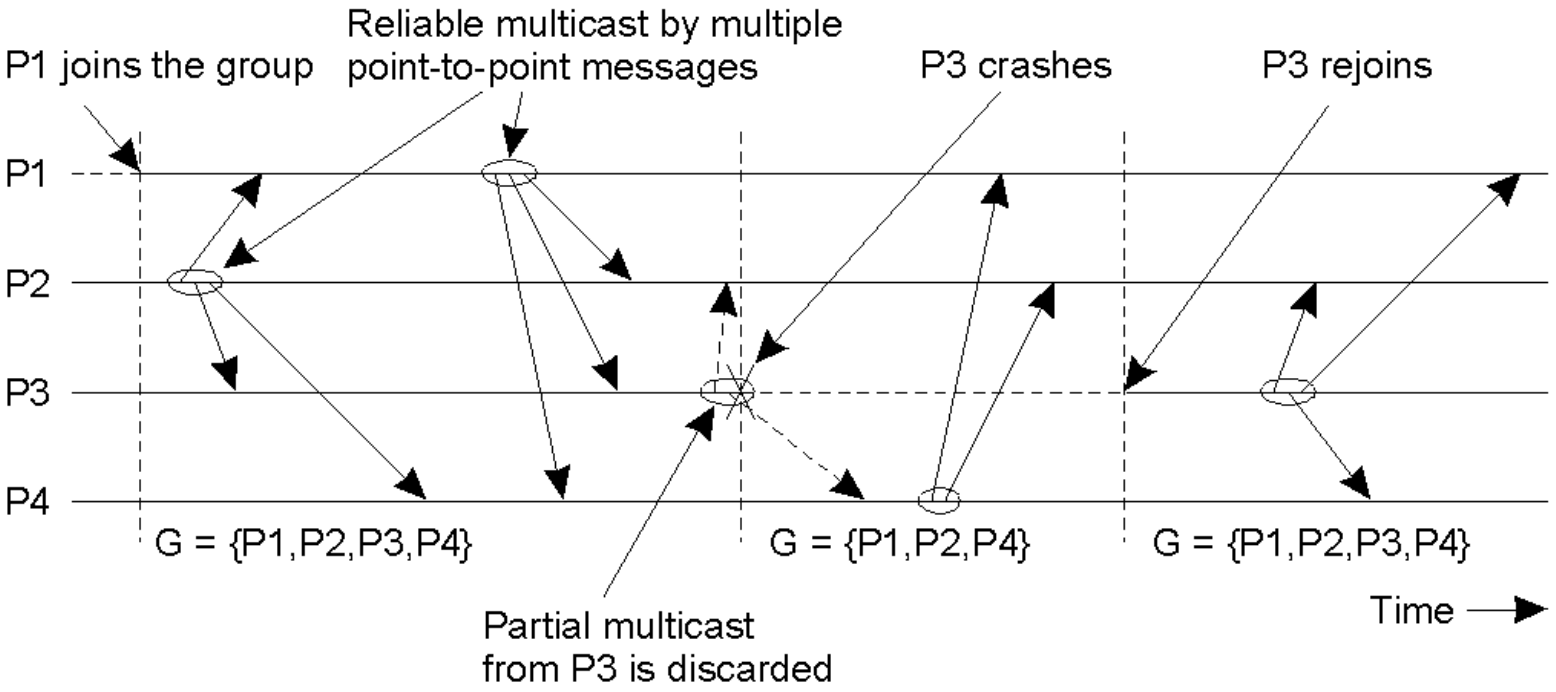
- m, vc => m is delivered to **all nonfaulty** processes in G_i (OR alternative: this order is not allowed!)
- vc, m => m is delivered to all processes in G_{i+1}

Problem: the sender fails *(during the multicast)*

Alternative solutions:

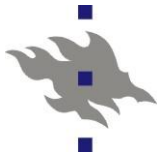
- m is delivered to all other members of G_i (=> ordering m, vc)
- m is ignored by all other members of G_i (=> ordering vc, m)

Virtually Synchronous Reliable MC (2)



The principle of virtual synchronous multicast:

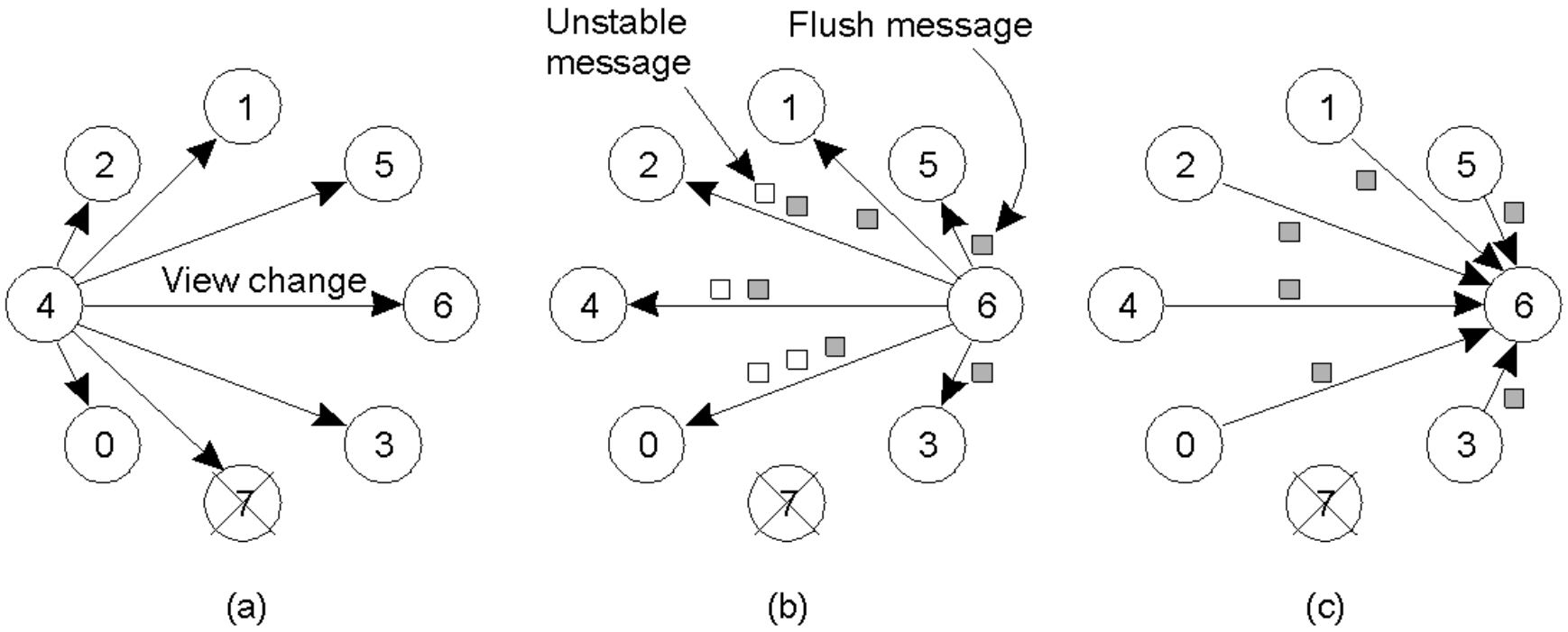
- a reliable multicast, and if the sender crashes
- the message may be delivered to all or ignored by all recipients



Implementing Virtual Synchrony

- Communication: reliable, order-preserving, point-to-point
- Requirement: all messages are delivered to all nonfaulty processes in G
- Solution
  - each p_j in G keeps a message in the hold-back queue until it knows that all p_j in G have received it
  - a message received by all is called **stable**
  - only stable messages are allowed to be delivered
  - view change G_i => G_{i+1} :
    - multicast **all unstable messages** to all p_j in G_{i+1}
    - multicast a **flush message** to all p_j in G_{i+1} (“I’ve no more unstables!”)
    - after having received a flush message from all: install the new view G_{i+1}

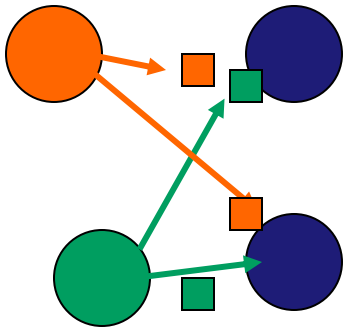
Implementing Virtual Synchrony



b) Process 6 sends out all its unstable messages, followed by a flush message

c) Process 6 installs the new view when it has received a flush message from everyone else

Ordered Multicast



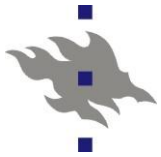
Need:
all messages are delivered in the intended order

(Recall the different orderings from earlier)

If p: multicast(G,m) and if (for any m')

- for **FIFO** multicast(G, m) **<** multicast(G, m')
- for **causal** multicast(G, m) **->** multicast(G, m')
- for **total** if at any q: deliver(m) < deliver(m')

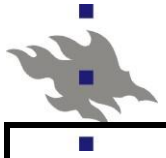
then for all q in G : deliver(m) < deliver(m')



Reliable FIFO-Ordered Multicast

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
(P1 and P4's receives are omitted)



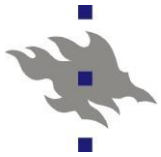
Virtually Synchronous Multicasting

Virtually synchronous 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

Six different versions of virtually synchronous reliable multicasting

- virtually synchronous: everybody or nobody (members of the current group)

  (sender fails: either everybody else or nobody)
- **atomic multicasting**: virtually synchronous reliable multicasting with totally-ordered delivery.

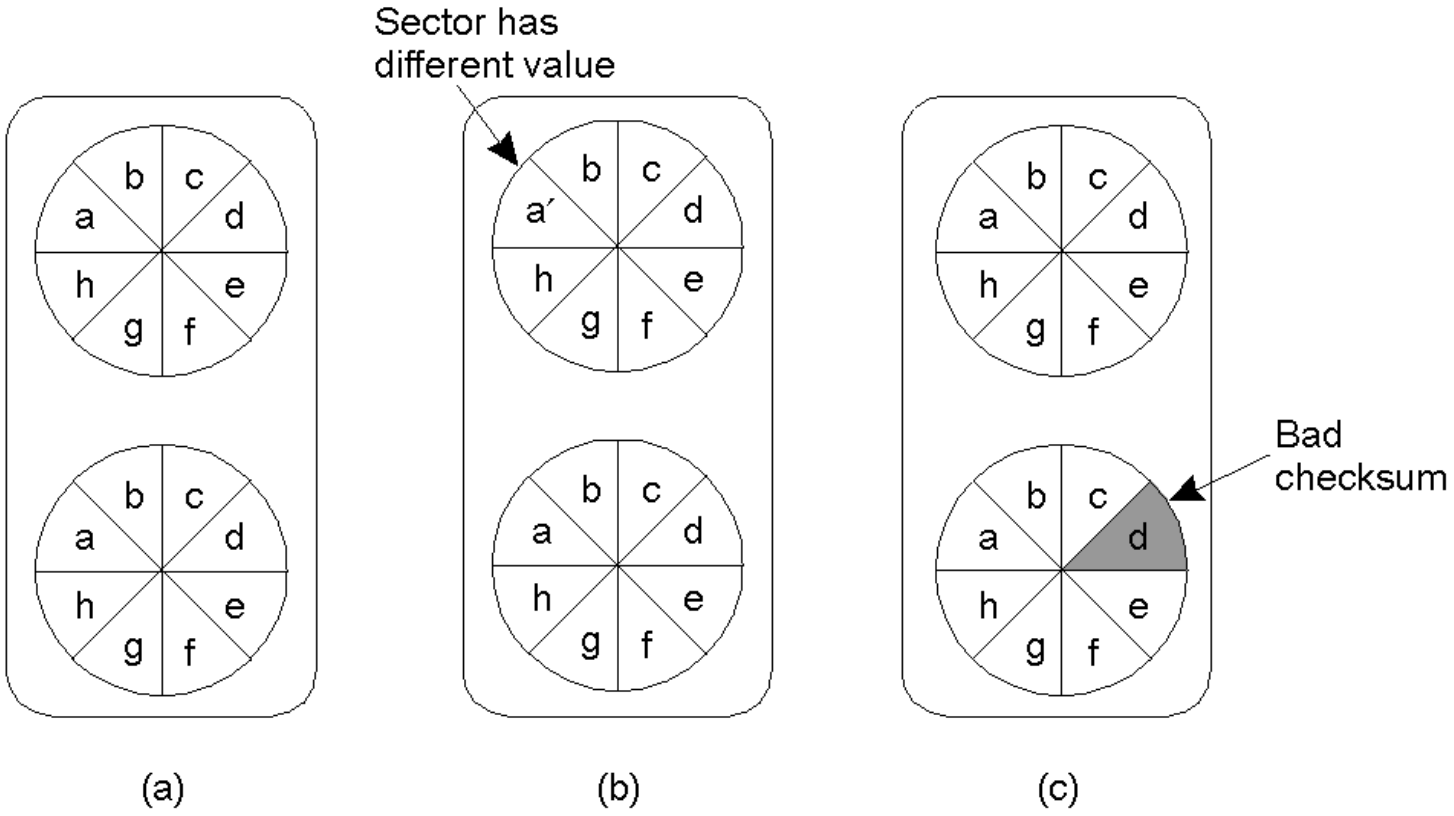


Recovery

- Fault tolerance: recovery from an error (erroneous state => error-free state)
- Two approaches
  - Backward recovery: back into a previous correct state
  - Forward recovery:
    - Detect that the new state is erroneous
    - Bring the system in a correct new state

    Challenge: the possible errors must be known in advance
  - Forward: continuous need for redundancy
  - Backward:
    - Expensive when needed
    - Recovery after a failure is not always possible

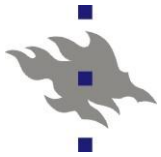
Wanted: Stable Storage



Stable Storage

Crash after drive 1 is updated, but copy to drive 2 was not done

Bad sector



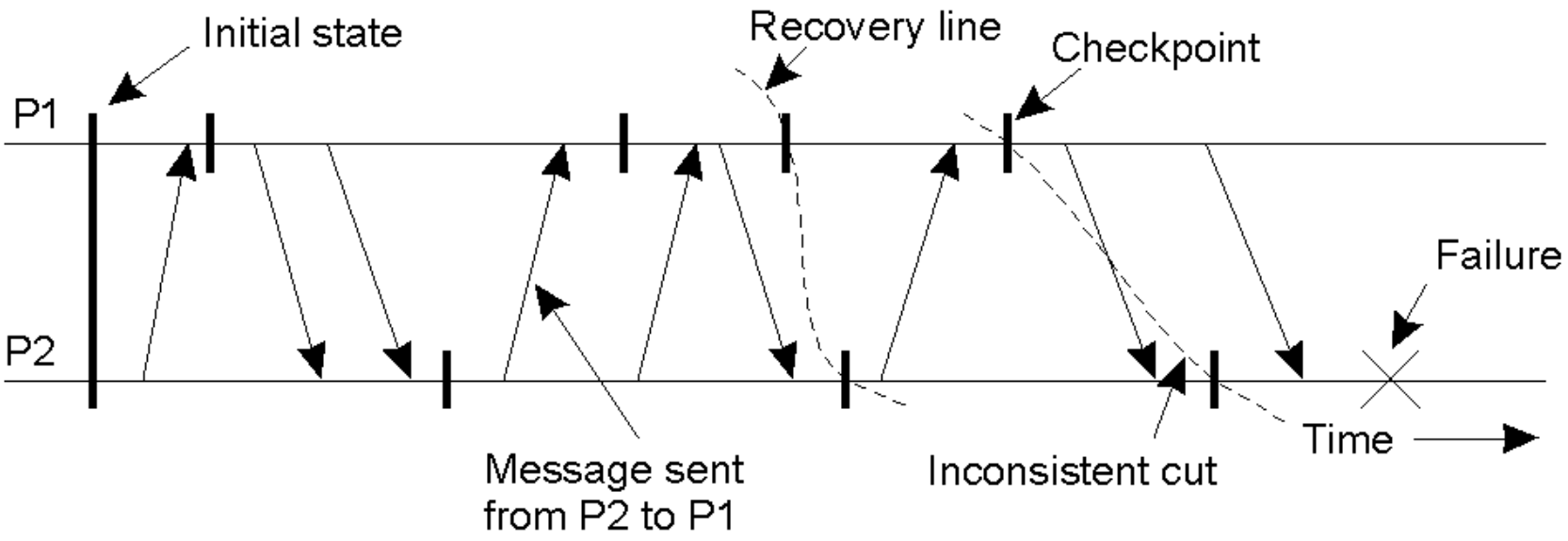
Implementing Stable Storage

- Careful block operations (fault tolerance: transient faults)
  - careful_read: {get_block, check_parity, error=> N retries}
  - careful_write: {write_block, get_block, compare, error=> N retries}
  - irrecoverable failure => report to the “client”
- Stable Storage operations (fault tolerance: data storage errors)
  - stable_get:

    {careful_read(replica_1), if failure then careful_read(replica_2)}
  - stable_put: {careful_write(replica_1), careful_write(replica_2)}
  - error/failure recovery: read both replicas and compare
    - both good and the same => ok
    - both good and different => replace replica_2 with replica_1
    - one good, one bad => replace data lost to the bad block from the good block

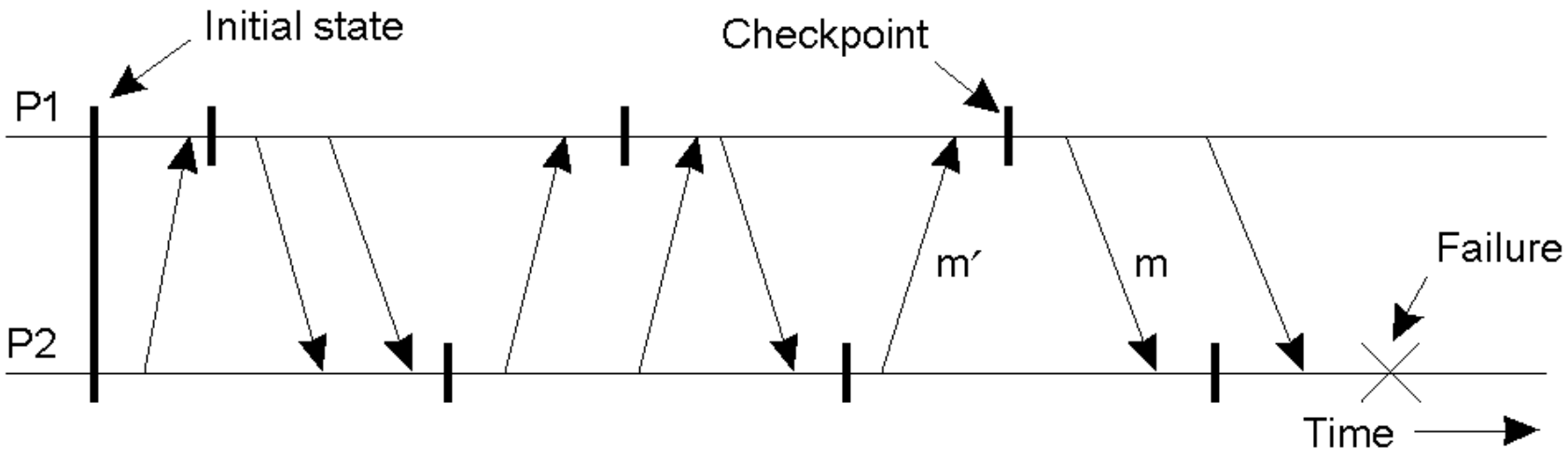
Backward recovery with checkpointing

Needed: a consistent global state to be used as a **recovery line**



A recovery line: the most recent distributed snapshot

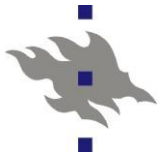
Independent Checkpointing



Each process records its local state from time to time
⇒ difficult to find a recovery line

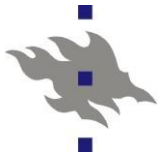
If the most recently saved states do not form a valid recovery line
⇒ rollback to a previous saved state (…threat: the domino effect).

A solution: coordinated checkpointing

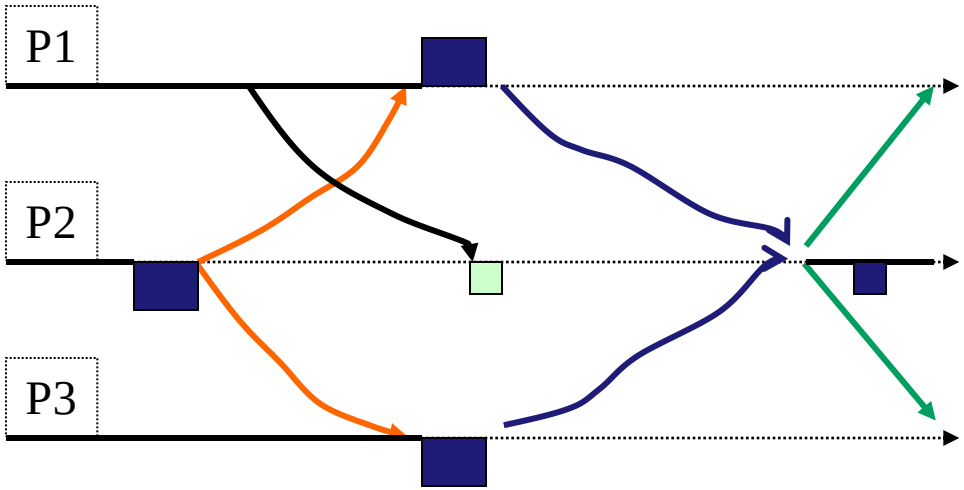


Coordinated Checkpointing (1)

- Non-blocking checkpointing
  - Recall: distributed snapshot
- Blocking checkpointing
  - coordinator: multicast CHECKPOINT_REQ
  - partner:
    - Take a local checkpoint
    - Acknowledge the coordinator
    - Wait (and queue any subsequent messages)
  - coordinator:
    - Wait for all acknowledgements
    - Multicast CHECKPOINT_DONE
  - coordinator, partner: Continue (and process queued messages)



Coordinated Checkpointing (2)



checkpoint request
ack
checkpoint done

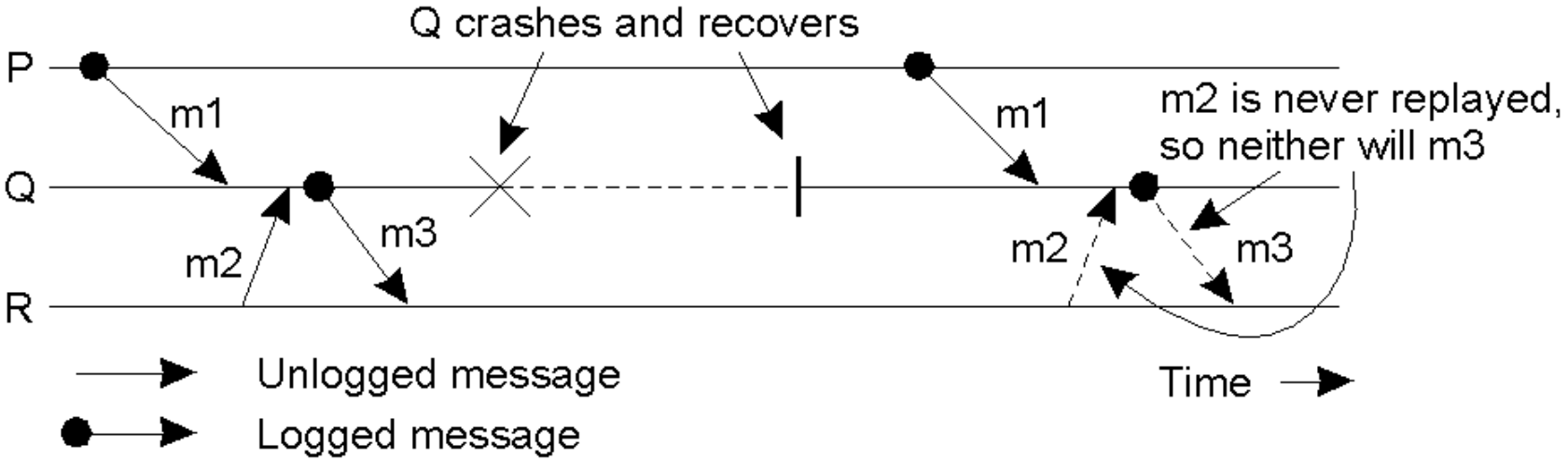
local checkpoint
message

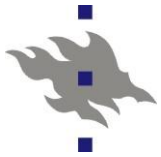
Message Logging

Improving efficiency: checkpointing and message logging

Recovery: most recent checkpoint + replay of messages

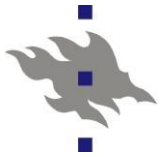
What messages to log? Everything that could lead to orphan processes if not replayed. E.g. m2 leads to m3 being sent – not logging m2 causes R to become orphaned (inconsistent with Q after recovery).





Two-Phase Commit: Basic Idea

- Idea: Transaction: messages, processing in a group
- At end: distributed commit: everyone agrees
  - Either the entire transaction occurred, or nothing (atomic!)
- Naïve idea: have coordinator tell everyone “let’s do this!”
  - One-phase commit (but no way to say “I can’t!”)
- Two-phase commit: coordinator asks: “do we do this?”
  - Everyone responds “yes” (vote_commit) or “no” (vote_abort)
  - If everyone says yes, coordinator says: “let’s do this!”, else “abort!”
- What could possibly go wrong here? – See exercises.



Chapter Summary

- Fault tolerance
- Process resilience
- Reliable group communication
- Distributed commit
- Recovery