POLITECNICO
MILANO 1863

## Distributed Systems

# Fault Tolerance

Gianpaolo Cugola

Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy

gianpaolo.cugola@polimi.it

# Contents

- **Introduction**
- An example: Client/server communication
- Protection against process failures
  - Agreement in process groups
- Reliable group communication
- Distributed commit
- Recovery techniques

# Why be fault tolerant

- Building dependable systems
  - Availability
    - We want the system to be ready for use as soon as we need it
  - Reliability
    - The system should be able to run continuously for long time
  - Safety
    - It should cause nothing bad to happen
  - Maintainability
    - It should be easy to fix
- Availability vs. Reliability
  - If a system goes down for one millisecond every hour, it has an availability of 99.999%, but is still highly unreliable

# From faults to failures

- A system *fails* when it is not able provide its services
- Failure is the result of an *error*
- An error is caused by a *fault*

- Some faults can be avoided, others cannot
- Building a dependable system demands the ability to deal with the presence of faults
  - A system is said to be *fault tolerant* if it can provide its services even in the presence of faults

# Just one fault is not enough

- Faults can be classified according to the frequency at which they occur
  - *Transient faults* occur once and disappear
  - *Intermittent faults* appear and vanish with no apparent reason
  - *Permanent faults* continue to exist until the failed components are repaired

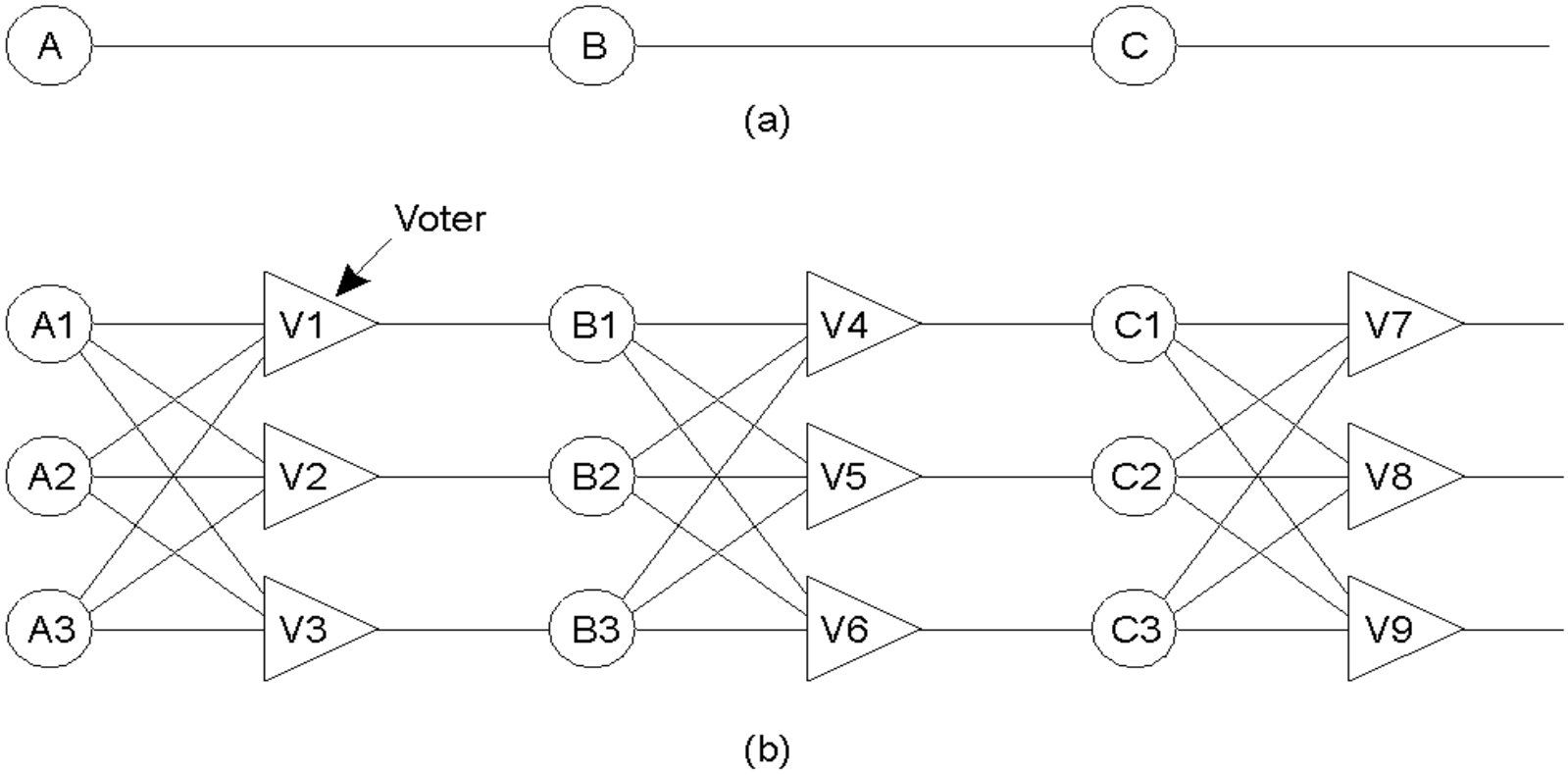
# Faults/Failures in DS

- Different types of failures (failure model)
  - Omission failures
    - Processes: fail-safe (sometimes wrong, but easily detectable output), fail-stop (detectable crash), fail-silent (undetectable crash)
    - Channels: send omission, channel omission, receive omission
  - Timing failures (apply to synchronous systems, only)
    - Occur when one of the time limits defined for the system is violated
  - Byzantine (or arbitrary) failures
    - Processes: may omit intended processing steps or add more
    - Channels: message content may be corrupted, non-existent messages may be delivered, or real messages may be delivered more than once

# General techniques

- The key technique to mask a failure is redundancy
  - Information redundancy
    - Hamming codes …
  - Time redundancy
    - Hard luck! Try again?
  - Physical redundancy
    - Two is better than one
    - First implemented in biological systems

# A redundancy example – Triple Modular Redundancy




Replicate each sub-module: they won't be all rotten

# Contents

- Introduction
- **An example: Client/server communication**
- Protection against process failures
  - Agreement in process groups
- Reliable group communication
- Distributed commit
- Recovery techniques

# Reliable C/S communication

- Reliable point-to-point communication is usually provided by the TCP protocol, which masks omission failures using acknowledgements and retransmissions
- Crash failures, however, e.g. the loss of a connection, are not so easily masked
- This becomes critical when trying to provide high-level communication facilities that completely mask communication issues
- Completely masking is just not feasible

# The RPC illusion

- Ideally RPC should provide the semantics of procedure calls in a distributed setting
- In reality a number of problems arise:
  - Server cannot be located
  - Lost Requests
  - Server crashes
  - Lost replies
  - Client crashes

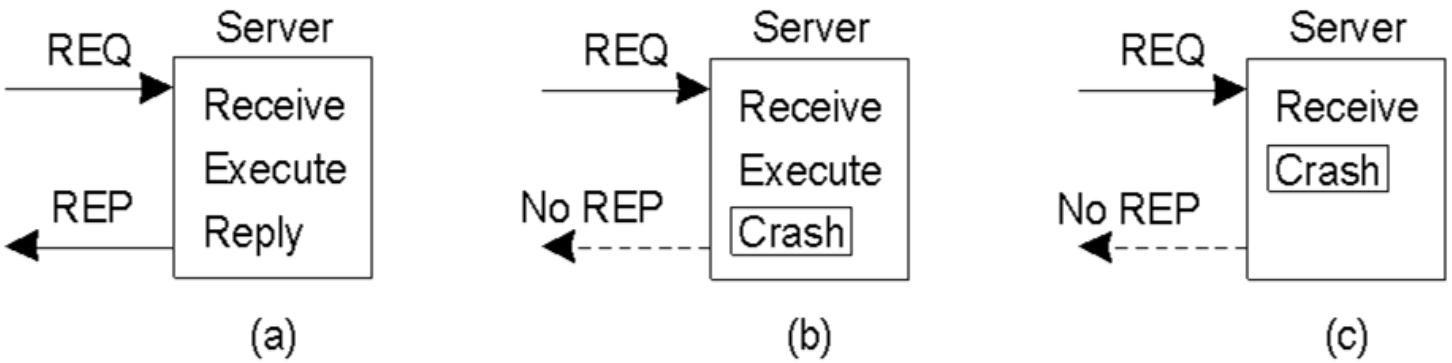
- Have you ever wondered whether the '+' operator would actually work while writing a C program?

# The benign cases

- If the server cannot be located by the client, the client can handle this as an exception
  - This can also happen when the client is using an obsolete version of the protocol
  - Annoying but tolerable
- If the request from the client is lost, the client can try and send it again
  - If too many requests are lost, the client can conclude the server is down
  - But what if the reply has been lost?

# The server crashes

- The problem is that the client cannot tell when the server crashed




- It is easy to ensure that the job has been done:
  - *At most once*
  - *At least once*

# A print server

- The server performs two operations:
  - Print the requested document (P)
  - Send a confirmation message (M)
- The message can be sent either before or after printing
- The server can crash at any time.
  - Before beginning: C (-> P -> M) or C (->M -> P)
  - At the end: P -> M -> C or M -> P -> C
  - In the middle: P->C (->M) or M-> C (->P)

# The printing client

- Assuming the client knows the server crashed, it can use four strategies
  - Always reissue request
  - Never reissue request
  - Reissue only when message not received
  - Reissue only when message received

- Some strategies may be better than the others but no one can give *exactly once* semantics

# Printer server and client

**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 |

- All strategies turn out to be suboptimal

# Lost replies

- Similar problems occur when the reply from the server is lost
  - Was the reply lost? Or did the request not arrive?
  - Some requests are *idempotent* others are not
- The server should cache clients' requests so as not to repeat duplicate ones
  - For example using sequence numbers
- How long to keep this information?
- Can we have them reach an agreement?

# Clients crash too

- A computation started by a dead client is called an *orphan*
- Orphans still consume resources on the server. The server must get rid of them
  - *Extermination*: RPC's are logged by the client and orphans killed (on client request) after a reboot -> costly (logging each call) and problem with grand orphans and network partitions
  - *Reincaration*: when a clients reboots it starts a new epoch and sends a broadcast message to servers, who kill old computations started on behalf of that client (including grand orphans). Replies sent by orphans bring an obsolete epoch and can be easily discarded
  - *Gentle reincarnation*: as before but servers kill old computations only if owner cannot be located
  - *Expiration*: Remote computation expire after some time. Clients wait to reboot to let remote computations to expire
- Reincarnation may seem to solve all problems, but what if the killed orphans had open files?
  - There's a lot of bookeeping involved, and not all issues can be solved

# Contents

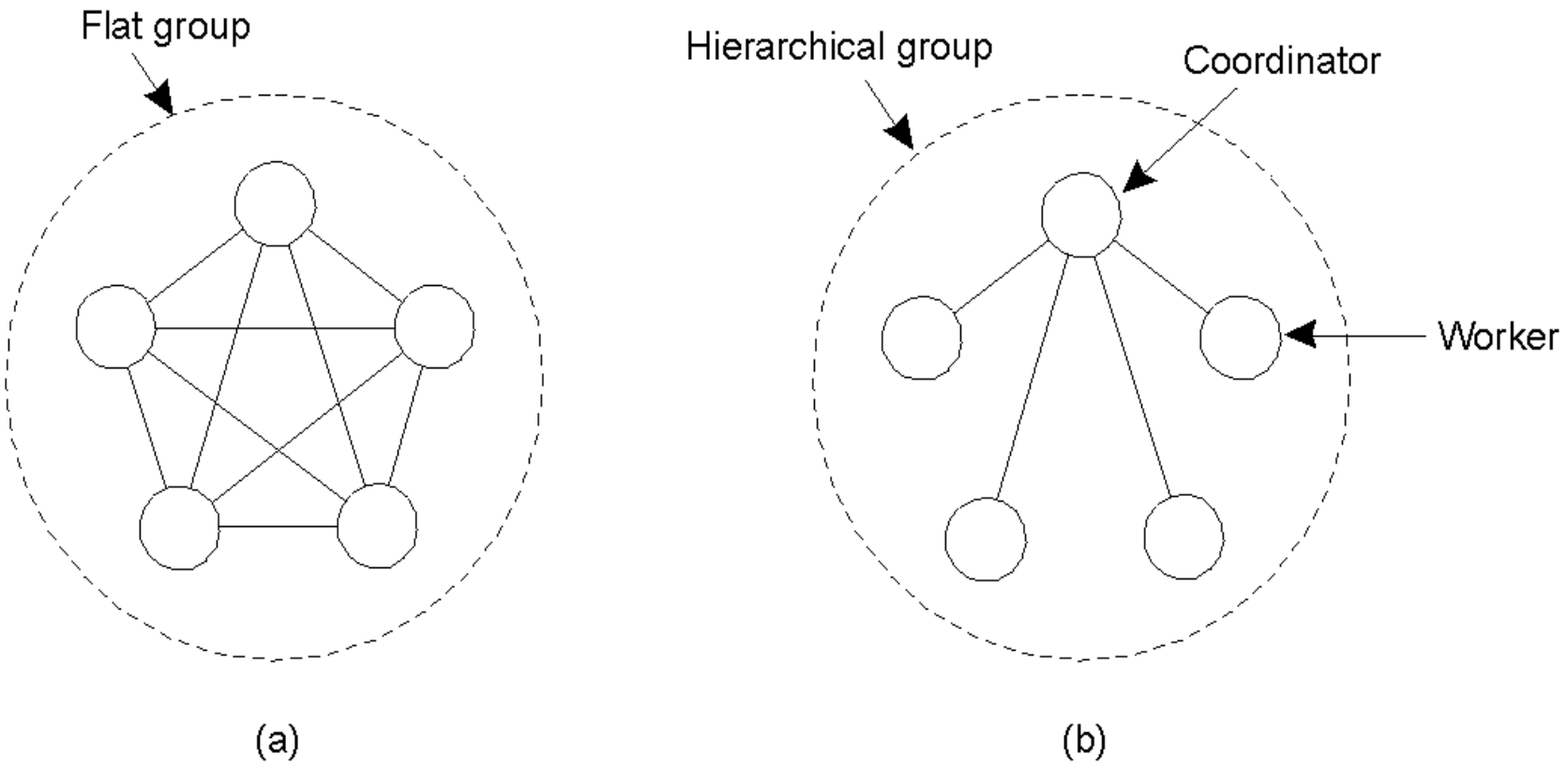
- Introduction
- An example: Client/server communication
- **Protection against process failures**
  - **Agreement in process groups**
- Reliable group communication
- Distributed commit
- Recovery techniques

# Process resilience

- Redundancy can be used to mask the presence of faulty processes, with redundant process groups
  - The work that should be done by a process is taken care by a group of processes
  - The healthy processes can continue to work when some of the others fail

# Process resilience

- Two possible organizations:



# Process resilience

- Using groups is easier said than done:
  - We must keep track of which processes constitute each group
  - Having a group coordinator is easy, but ….
- Distributed membership management (in flat groups) requires that join and leave announcements be (reliably) multicast
  - What happens if a process crashes? It won't tell everybody "I'm dying"
  - If too many processes crash or leave, the group will have to be rebuilt

# Process resilience

- But how large should a group be?
  - Let's consider a replicated-write system: information is stored by a set of replicated processes
    - If processes fail silently, then k+1 processes allow the system to be *k-fault-tolerant*
    - If failures are Byzantine, matters become worse: 2k+1 processes are required to achieve k-fault tolerance (to have a working voting mechanism)
  - In practice, we cannot be sure that no more than k processes will ever fail simultaneously
- This is fine, but in some cases we need even more

# Agreement in process groups

- A number of tasks may require that the members of a group agree on some decision before continuing
  - E.g., electing a coordinator, committing a transaction….
- Because we are dealing with faults, we want all non-faulty processes to reach an agreement
- This differs from the previous case because the decision is not taken by an external observer (i.e., the client who can decide based on a voting mechanism)

# The consensus problem

- A set of processes must agree on some value
  - The non-faulty ones must agree
- The value is subject to a validity condition: It cannot be any value
- More precisely, we have the following conditions
  - Each process starts with some initial value
  - All non-faulty processes have to reach a decision based on these initial values
  - The following properties must hold
    - Agreement: No two processes decide on different values
    - Validity: If all processes start with the same value v, then v is the only possible decision value
    - Termination: All non-faulty processes eventually decide

# Do you remember the coordinated attack?


- Generals decide separately whether or not to attack at dawn.
  - If both attack: success
  - If only one attacks: failure
- Messengers for communication, but they can be lost/captured/delayed
- Question: How do the generals ensure either success or no attack? What algorithm do they use?

# Consensus with process failures

- The coordinated attack problem shows that consensus is not possible in the presence of arbitrary communication failures
- Let us now analyze what happens when communication is reliable but processes are allowed to fail
  - Crash Failures (fail-stop or fail-silent)
  - Byzantine Failures

# Assumptions for crashing processes

- Let's review our assumptions:
  - We consider a synchronous system, in which all processes evolve in synchronous rounds
    - A message sent by a process is received within the same round by the recipient (or within a bounded number of rounds)
    - Processes may fail at any point, by stopping taking steps (fail-silent)
  - We want our processes to reach agreement according to the previous definition of consensus

# Consensus with process failures is possible

- The problem we just stated is, luckily, easier than the previous one
  - It turns out that the problem can be solved provided that the processes take at least *f+1* rounds with *f* being a bound on the number of failures
  - Moreover, it is sufficient that a single process be non-faulty
- Can you think of an algorithm?

# FloodSet algorithm

- Let us now consider a simple algorithm to solve the problem
- Let *v0* be a pre-specified default value
- Each process maintains a variable *W (subset of V)* initialized with its start value
- The following steps are repeated for f+1 rounds
  - Each process sends *W* to all other processes
  - It adds the received sets to *W*
- The decision is made after f+1 rounds
  - if $|W| = 1$: decide on *W*'s element
  - if $|W| > 1$: decide on *v0* (or use the same function to decide, e.g. max(W))
- Remember: each process may stop in the middle of the send operation

# Prooving FloodSet correcteness

- ***Lemma***

1. If no process fails during a particular round $r : 1 \leq r \leq f+1$, then $W_i(r) = W_j(r)$ for all $i$ and $j$ that are active after $r$ rounds
2. Suppose that $W_i(r) = W_j(r)$ for all $i$ and $j$ that are active after r rounds. Then for any round $r1 : r \leq r1 \leq f+1$, the same holds, that is, $W_i(r1) = W_j(r1)$ for all $i$ and $j$ that are active after r1 rounds
3. If processes $i$ and $j$ are both active after $f+1$ rounds, then $W_i = W_j$ at the end of round $f+1$

# Improving FloodSet

- The complexity of the FloodSet algorithm can be reduced by using an optimized version
  - Each process needs to know the entire set $W$ only if its cardinality is greater than 1
  - The improved algorithm broadcasts $W$ at the first round
  - In addition each process broadcasts $W$ again when it first learns of a new value

# That was too easy

- Let us now make things more complex by introducing byzantine failures
  - Now processes can not only stop but also exhibit any arbitrary behavior, sending arbitrary messages, performing arbitrary state transitions and so on
- Our problem is now defined as follows
  - *Agreement*: No two **non-faulty** processes decide on different values
  - *Validity*: If all **non-faulty** processes start with the same value v, then v is the only possible decision value for all non-faulty processes
  - *Termination*: All non-faulty processes eventually decide
- The idea is that we cannot guarantee anything about faulty processes

# Going back to armies and generals

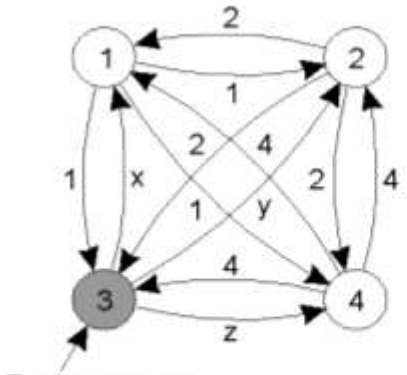
- The problem can be described in terms of armies and generals
  - This formulation is due to Lamport (1982)
  - Now we have *n* generals in the hills who have to reach a coordinated decision on whether to attack or retreat
  - They announce their troop strengths and decide based on the total number of troops
  - Communication is perfect, but some generals are traitors
    - Can we solve this problem?
    - How many rounds do we need?
    - How many non-faulty processes?

# Byzantine generals

- Our problem can be defined as follows
  - *Agreement*: No two loyal generals learn different troop strength values
  - *Validity*: If a loyal general announces v, then all loyal generals learn that his troop strength is v
  - *Termination*: All loyal generals eventually learn troop strength of all other loyal generals

# Byzantine generals: Lamport's alg. for 4 generals and 1 traitor

- Steps:
  - Send troop strength to others
  - Form a vector with received values
  - Send vector to others
  - Compute vector using majority for each vector position




(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)

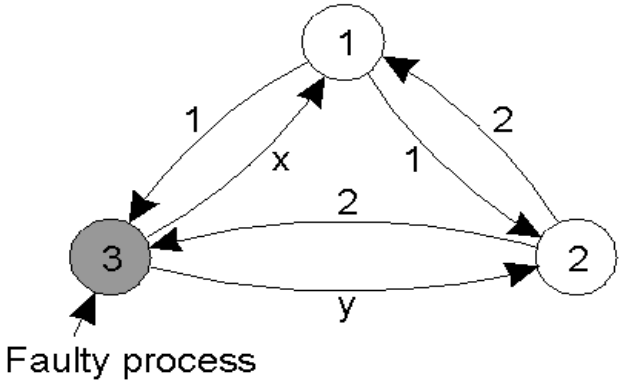
(b)

| 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)

# Byzantine generals: Soddisfacibility

- Lamport (1982) showed that if there are m traitors, 2m+1 loyal generals are needed for an agreement to be reached, for a total of 3m+1




(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)

- There is no way for 1 and 2 to determine a vector which is both correct and equal to that computed by the others

# If that wasn't hard enough

- So far we have considered synchronous systems
  - A real distributed system is inherently asynchronous
- Let us consider the problem of reaching an agreement beween *n* processes with *1* faulty process in an asynchronous system
  - Let's start with the "easy case": Fail-silent failures
- Can you find an algorithm?

# Asynchronous system with crash failures

- Fischer, Lynch, and Paterson proved that no solution exists: "Impossibility of Distributed Consensus with One Faulty Process"
  - The result was proved in the case of crash failures
  - What happens in the case of Byzantine failures?
    - A byzantine general can simulate a crashed general so if a solution existed for byzantine failures that solution would also work for crash failures

# Contents

- Introduction
- An example: Client/server communication
- Protection against process failures
  - Agreement in process groups
- **Reliable group communication**
- Distributed commit
- Recovery techniques

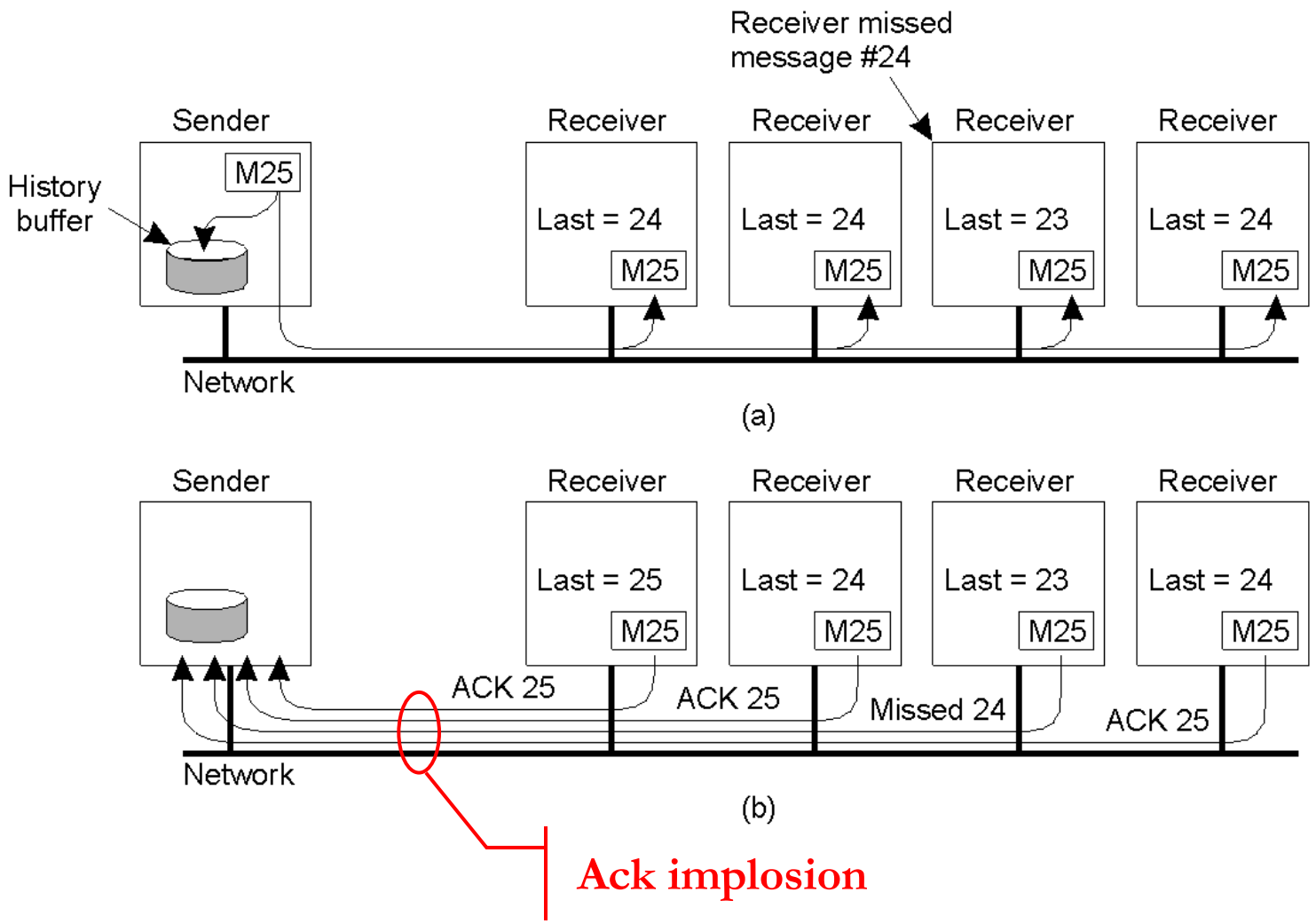
# Reliable group communication

- It is of critical importance if we want to exploit process resilience by replication
- Achieving reliable multicast through multiple reliable point-to-point channels may not be efficient or enough
- What happens if the sender crashes while sending?
- What if the group changes while a message is being sent?

# Reliable group communication

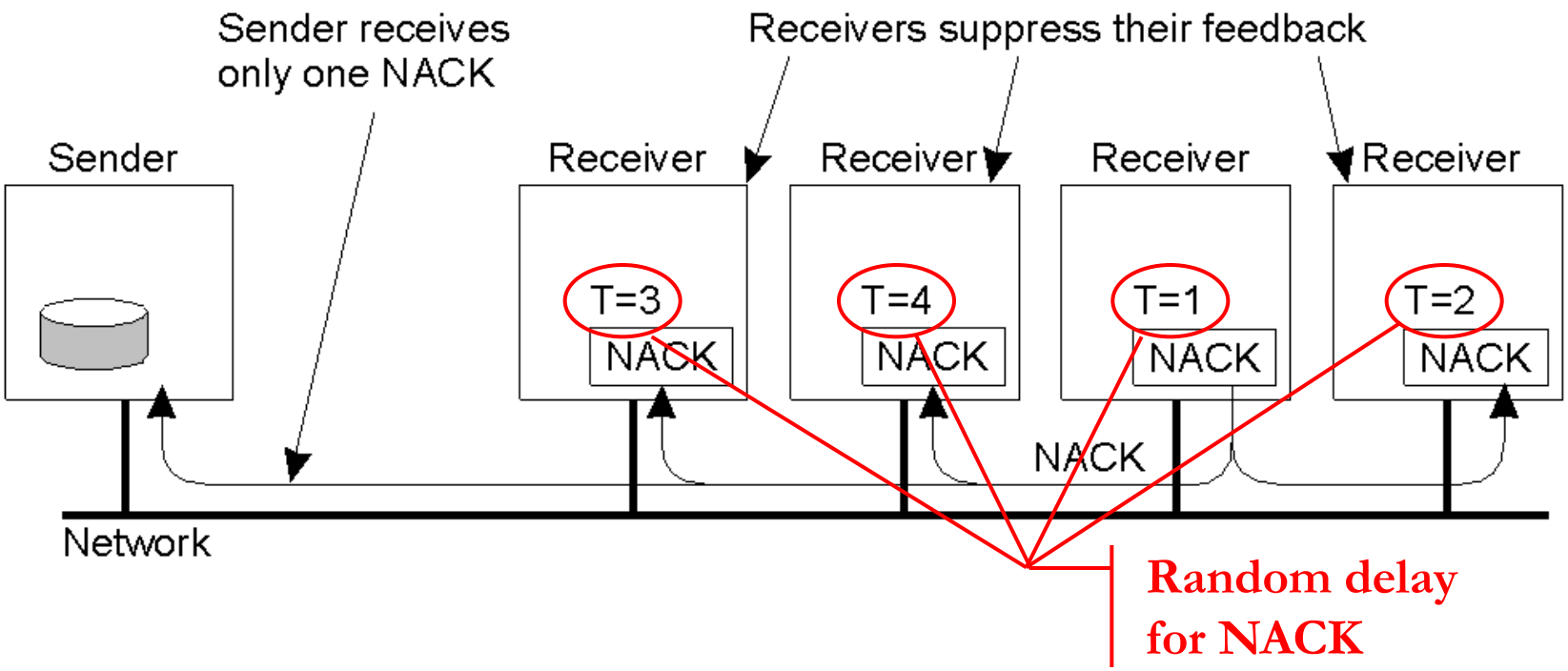
- Groups are fixed, and processes non-faulty
  - All group members should receive the multicast (not necessarily in the same order)
  - Easy to implement on top of unreliable multicast
    - Positive acknowledgements
      - Large number of ack messages
    - Negative acknowledgments
      - Fewer messages but sender has to cache messages
- Case of faulty processes (we will see it later)
  - All non-faulty group members should receive the message
  - There must be agreement about who is in the group

# Non-faulty processes: Basic approach



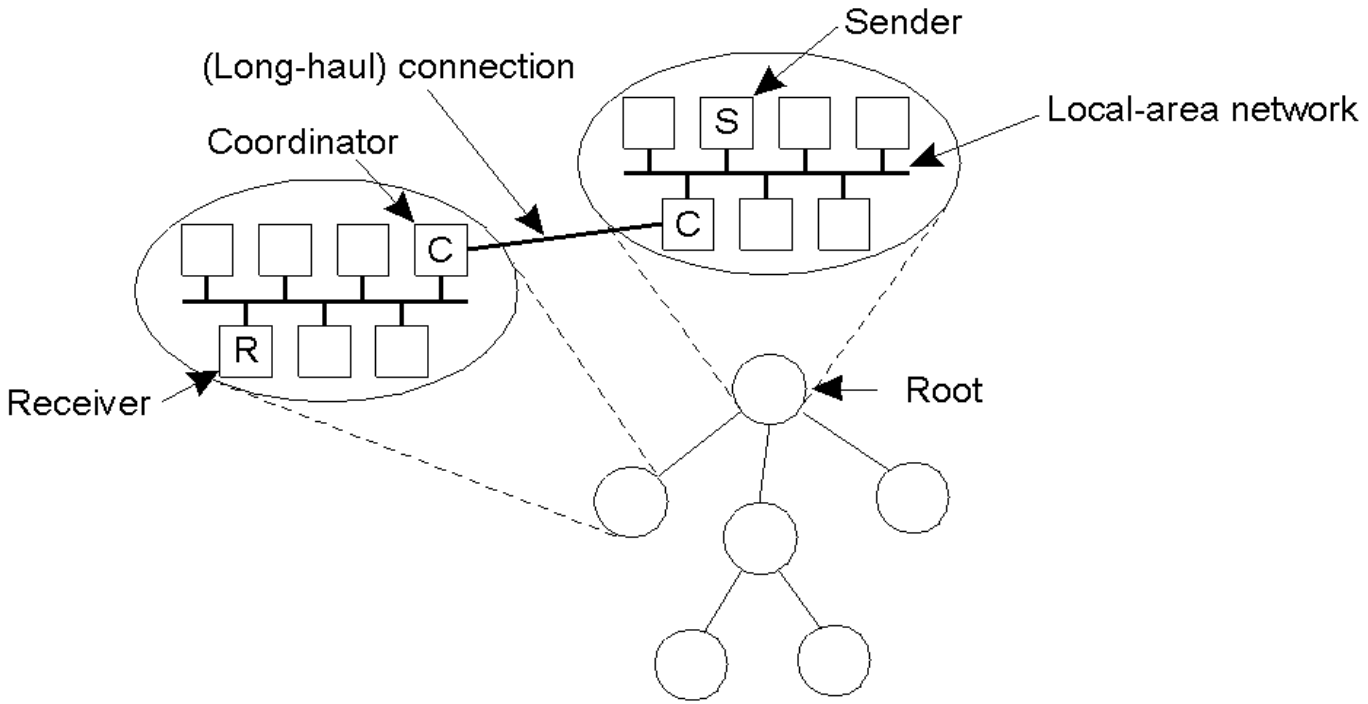
# Non-faulty processes: Scalable Reliable Multicast

- A first solution is non-hierarchical feedback control, implemented in SRM
  - Negative acknowledgements are multicast
    - Good to suppress replicated NACK but also bad since everyone must process NACKs



# Non-faulty processes: Hierarchical Feedback Control

- With Hierarchical Feedback control, receivers are organized in groups headed by a coordinator...
- ... and groups are organized in a tree routed at the sender



# Non-faulty processes: Hierarchical Feedback Control

- The coordinator can adopt any strategy within its group
- In addition it can request retransmissions to its parent coordinator
  - A coordinator can remove a message from its buffer if it has received an ack from
    - All the receivers in its group
    - All of its child coordinators
- The problem is that the hierarchy (tree) has to be constructed and maintained

# The case of faulty processes

- Things get harder if processes can fail or join and leave the groups during communication
- What is usually needed is that a message be delivered either to all the members of a group or to none, and that the order of messages be the same at all receivers
- This is known as the *atomic multicast problem*
- Consider an update in a replicated database
  - Everything is fine if all the (non faulty) replicas receive the same commands in the same order

# Close synchrony

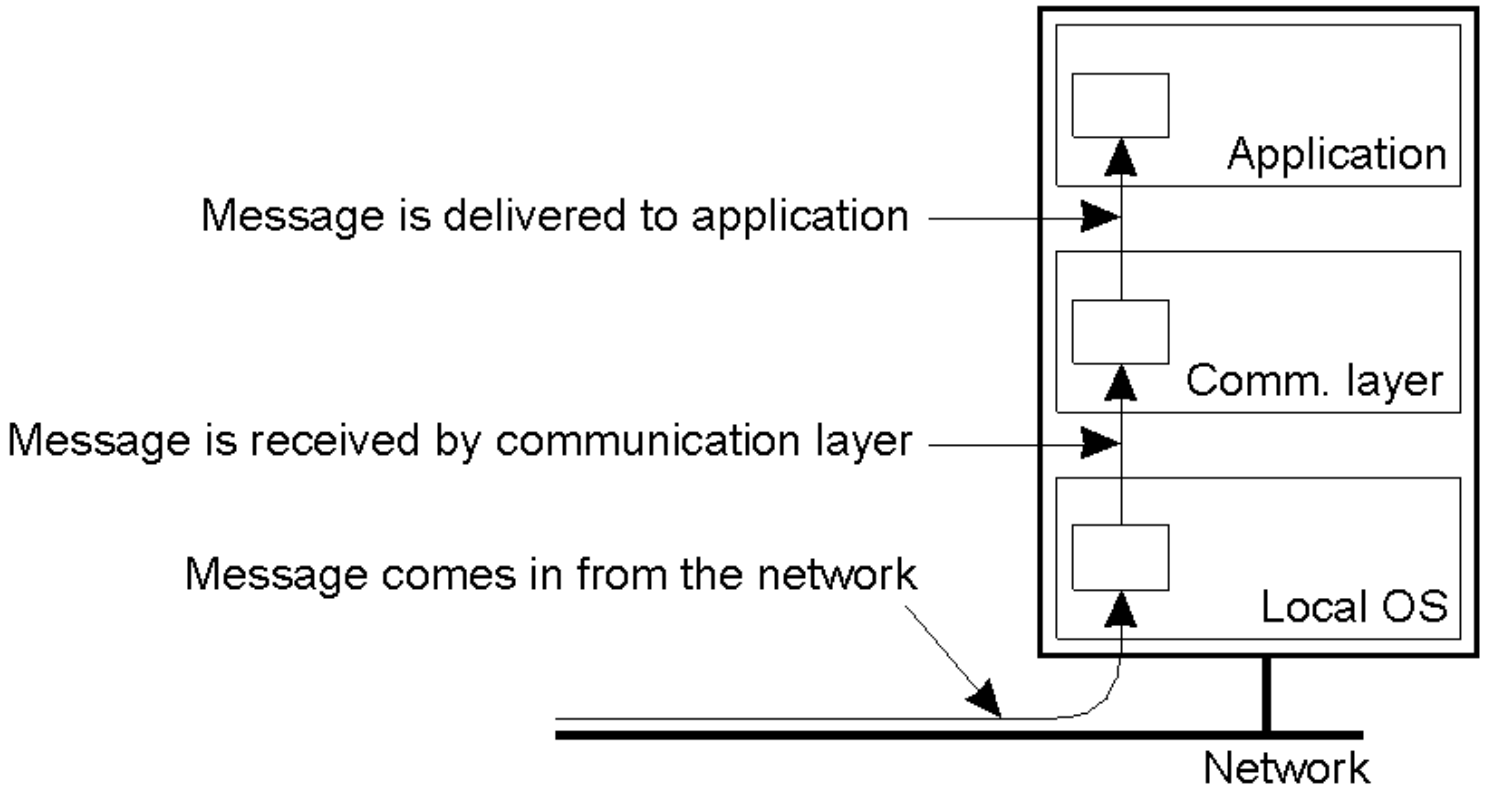
- Ideally we would like:
  - Any two processes that receive the same multicast messages or observe the same group membership changes to see the corresponding events in the same order
  - A multicast to a process group to be delivered to its full membership. The send and delivery events should be considered to occur as a single, instantaneous event
- Unfortunately, close synchrony cannot be achieved in the presence of failures

# Towards virtual synchrony

- A mechanism to detect failures is needed
  - Remember the Fischer, Lynch, and Paterson result
- Even if we can detect failures correctly, we cannot know whether a failed process has received and processed a message
- Our new (weaker) model becomes
  - Crashed processes are purged from the group and have to join again, reconciling their state
  - Messages from a correct process are processed by all correct processes
  - Messages from a failing process are processed either by all correct members or by none
  - Only relevant messages are received in a "specific" order
    - More on this later
- Let's see how we can implement this kind of group communication

# Towards virtual synchrony

- To our multicast primitive it is useful to adopt the following model which distinguishes between *receiving* and *delivering* a

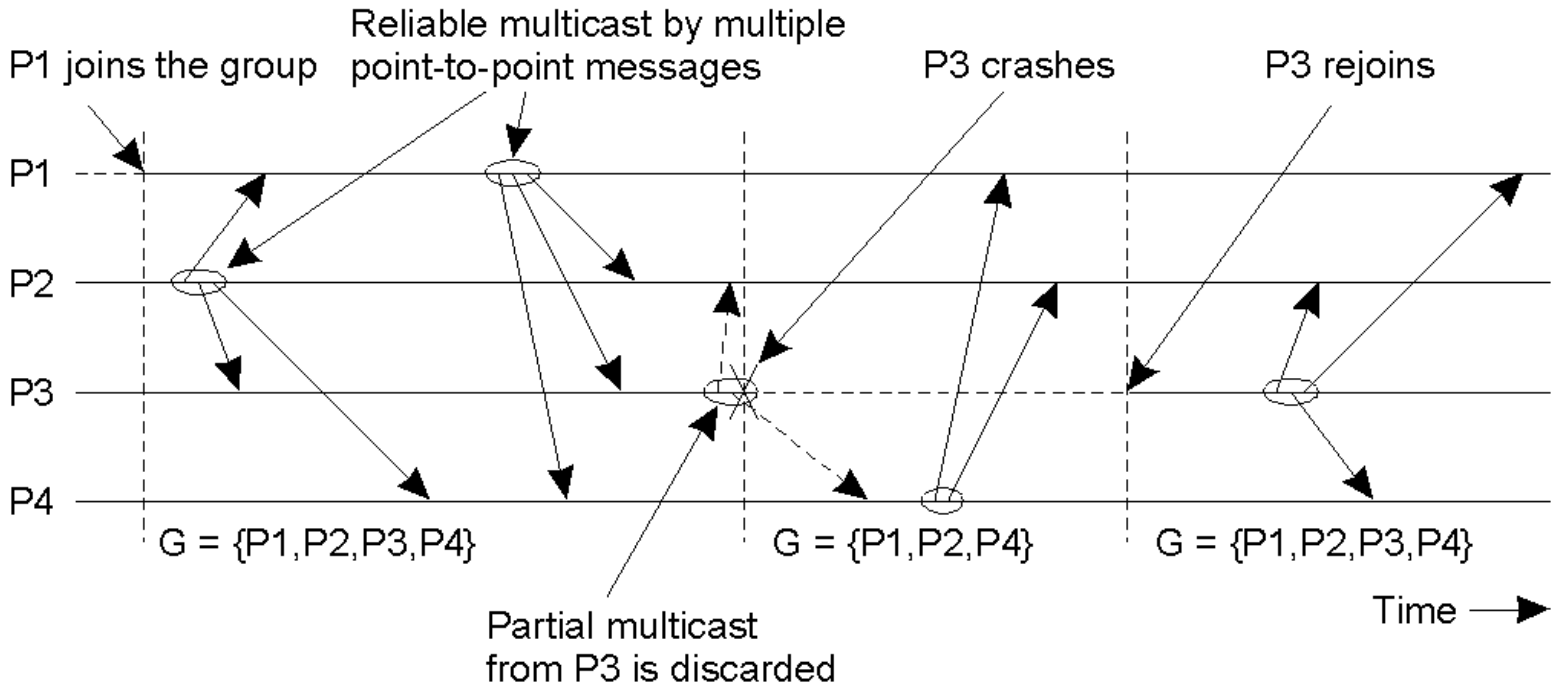


# Virtual synchrony

- A *group view* is the set of processes to which a message should be delivered as seen by the sender at sending time
- The minimal ordering requirement is that
  - Group view changes be delivered in a consistent order with respect to other multicasts and with respect to each other
- This, together with the previous requirements, leads to a form of reliable multicast which is said to be *virtually synchronous* [Birman and Joseph, 1987]

# Virtual synchrony

- The messages sent by P3 "while" crashing are discarded

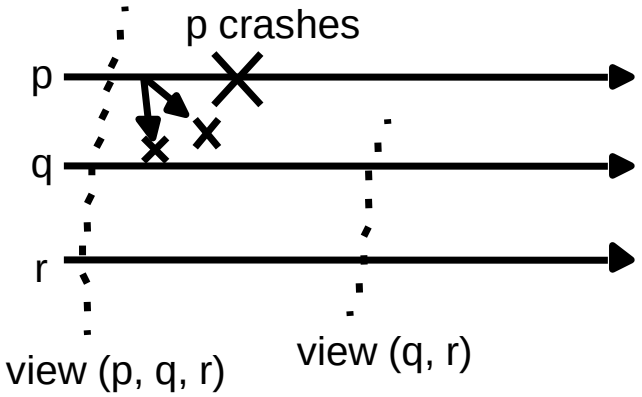


# Virtual synchrony

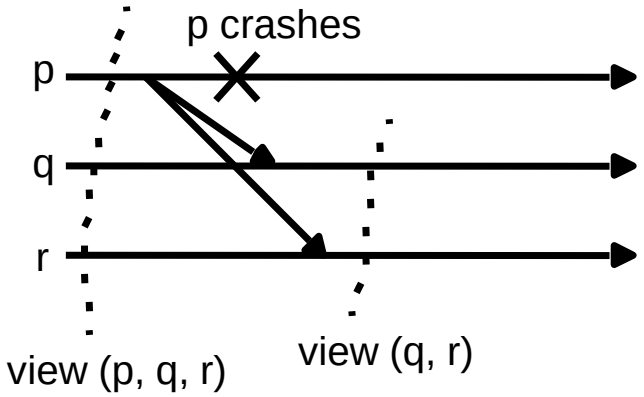
- We say that a *view change* occurs when a process joins or leaves the group, possibly crashing
- All multicast must take place between view changes
  - We can see view changes as another form of multicast messages (those announcing changes in the group membership)
  - We must guarantee that messages are always delivered before or after a view change
  - If the view change is the result of the sender of a message m leaving the message is either delivered to all group members before the view change is announced or it is dropped
- Multicasts take place in epochs separated by group membership changes

# Virtual synchrony

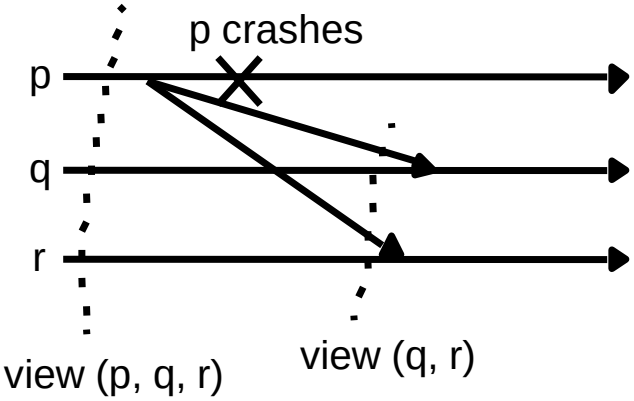
a (allowed).



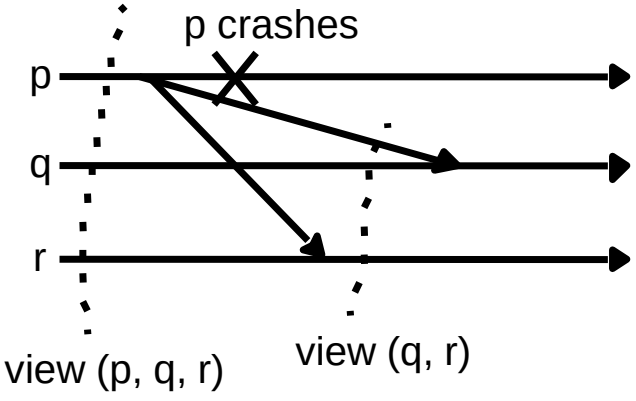

b (allowed).




c (disallowed).




d (disallowed).



# Message ordering

- Retaining the virtual synchrony property, we can identify different orderings for the multicast messages
  - Unordered multicasts
  - FIFO-ordered multicasts
  - Causally-ordered multicasts
- In addition, the above orderings can be combined with a *total ordering* requirement
  - Whatever ordering is chosen, messages must be delivered to every group member in the same order
    - Virtual synchrony includes this property in its own definition

# Message ordering - FIFO

| 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 | |

- Messages from the same sender are received in the same order by all receivers

# Atomic multicast

- *Atomic multicast* is defined as a virtually synchronous reliable multicast offering totally-ordered delivery of messages

| 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 |

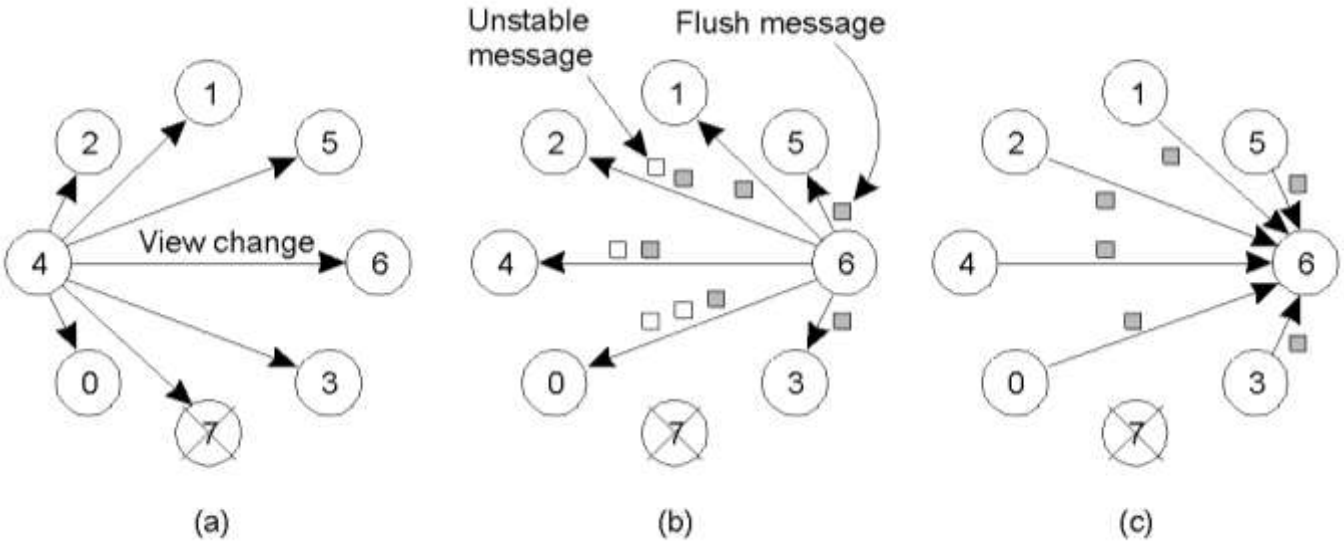
# An implementation of VS

- Virtual synchrony is implemented in ISIS, a fault tolerant distributed system [Birman et al. 1991], making use of
  - Reliable and FIFO point-to-point channels
- In practice, although each transmission is guaranteed to succeed, there are no guarantees that all group members receive it; the sender may fail before completing its job
- ISIS must make sure that messages sent to a group view are all delivered before the view changes
  - Every process keeps a message *m* until it is sure that all the others have received it. Once this happens, *m* is said to be *stable*
  - We assume that processes are notified when messages become stable and that they keep them in a buffer until that time

# An implementation of VS

- The basic idea is as follows
  - We assume that processes are notified of view changes by some (possibly distributed component)
  - When a process receives a view change message, stop sending new messages until the new view is installed, instead it multicasts all pending unstable messages to the non faulty members of the old view, marks them as stable and multicasts a flush message
  - Eventually all the non faulty members of the old view will receive the view change and do the same (duplicates are discarded)
  - Each process installs the new view as soon as it has received a flush message from each other process in the new view. Now it can restart sending new messages
- If we assume that the group membership does not change during the execution of the protocol above we have that at the end all non faulty members of the old view receive the same set of messages before installing the new view
  - The protocol can be extended to account the case of group changes while in progress
- This orders view changes with respect to messages

# An implementation of VS




- Process 4 notices that process 7 has crashed, sends a view change
- Process 6 sends out all its unstable messages, followed by a flush message
- Process 6 installs the new view when it has received a flush message from everyone else

# Contents

- Introduction
- An example: Client/server communication
- Protection against process failures
  - Agreement in process groups
- Reliable group communication
- **Distributed commit**
- Recovery techniques

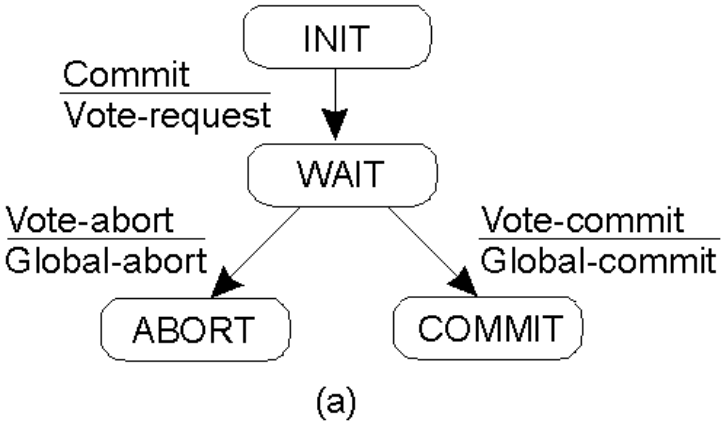
# Commit protocols

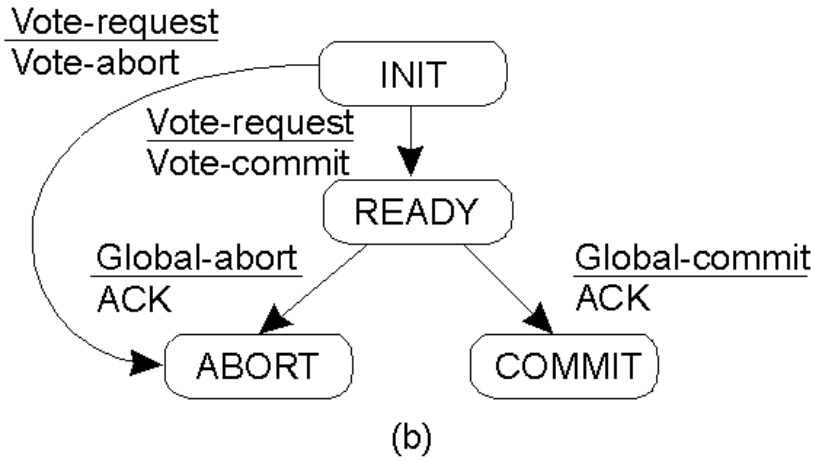
- A case of agreement you have certainly considered is that of commit protocols
  - What are the assumption at the basis of a commit protocol?
  - What guarantees does a commit protocol offer?
  - **Agreement:** No two processes decide on different values
  - **Validity**
    - If all processes start with 1 then 1 is the only possible decision value
    - If any process starts with 0, then 0 is the only possible decision value
  - **Termination**
    - If there are no faults, all processes eventually decide (weak)
    - **or**
    - All non-faulty processes eventually decide (strong)
  - Consensus and atomic commitment look similar...
    - Does the Fischer, Lynch, and Paterson result apply here?

# Two- and three-phase commit

- Two-phase commit is a blocking protocol, as such it satisfies the weak termination condition
- This allows it to reach termination in less that f+1 rounds
- Three-phase commit is non-blocking. It satisfies the strong termination condition but may require a large number of rounds to terminate
  - The good thing is that with no failures only 3 rounds are required

# Two-phase commit




a) Coordinator (aka Transaction Manager)
b) Replica (aka Resource Manager)

# Two-phase commit: Possible failures

- A participant fails
  - After a timeout the coordinator can assume abort decision by participant
- The coordinator fails
  - Participant blocked waiting for vote request → It can decide to abort
  - Participant blocked waiting for global decision (state READY) → It cannot decide on its own, must wait for the coordinator to recover
  - Or it can request retransmission of the decision from another participant
    - Who may have received a reply from the coordinator
    - Or can be in INIT state, which means the coordinator has crashed before completing the starting phase → the participant may safely abort
    - What if everybody is in the same READY situation? Nothing can be decided until the coordinator recovers (*blocking protocol*)

# Two-phase commit

- **Actions by the coordinator**

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

# Two-phase commit

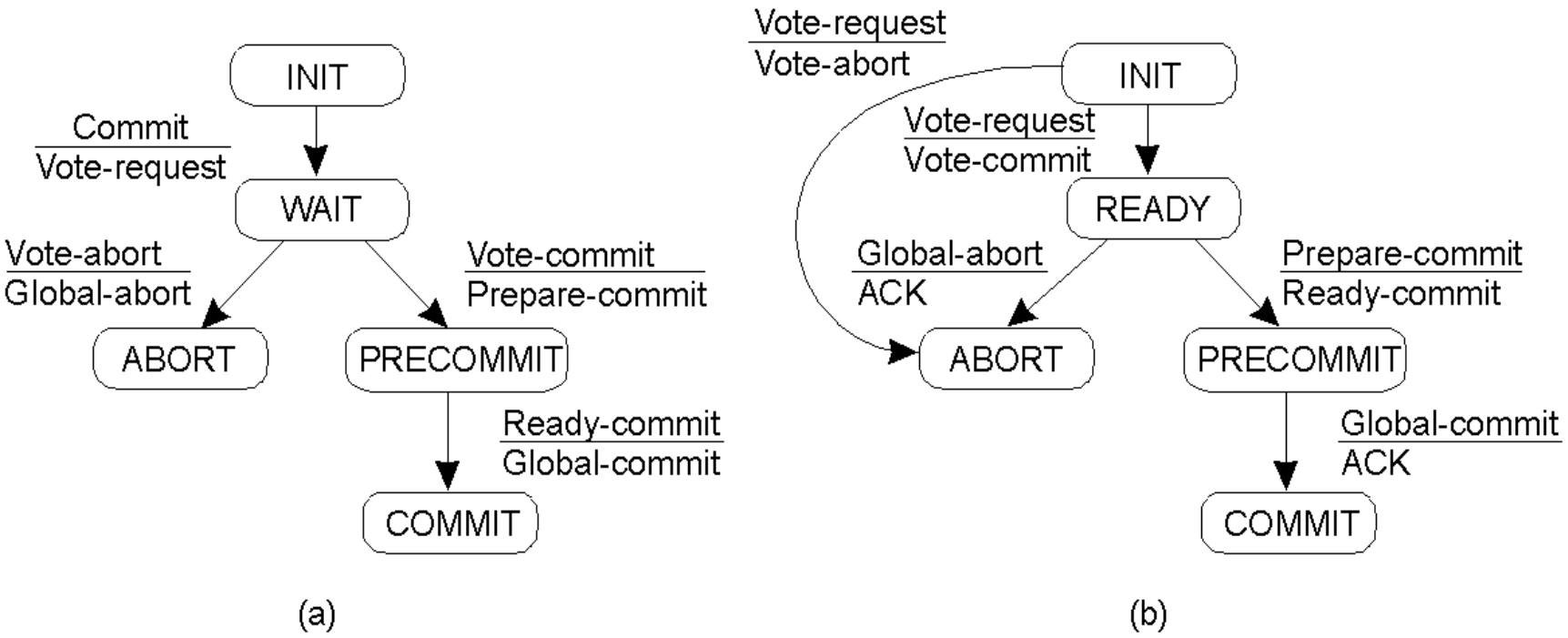
- **Actions by a participant**

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {  write VOTE_ABORT to local log;  exit;  }
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT  write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT  write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send  VOTE ABORT to coordinator;
}
```

# Three-phase commit

- Attempt to solve the problems of two-phase commit by adding another phase to the protocol
  - No state leading directly to COMMIT and ABORT
  - No state where final decision is impossible can lead to COMMIT
- Above conditions are satisfied *iif* protocol is non-blocking (Skeen and Stonebraker, 1983)

# Three-phase commit




a) Coordinator (aka Transaction Manager)
b) Replica (aka Resource Manager)

# Three-phase commit: Possible failures

- **A participant fails**
  - Coordinator blocked waiting for vote → it can assume abort decision by participant
  - Coordinator blocked in state PRECOMMIT → it can safely commit and tell the failed participant to commit when it recovers
- **The coordinator fails**
  - Participant blocked waiting for vote request → It can decide to Abort
  - Participant blocked waiting for global decision → It contacts other participant:
    - ABORT (at least one) → ABORT ;
    - COMMIT (at least one) → COMMIT ;
    - INIT (at least one) → ABORT;
    - PRECOMMIT (majority) → COMMIT;
    - READY (majority) → ABORT
  - Alternatively it is possible to elect new coordinator
  - No two participants can be one in PRECOMMIT and the other in INIT

# Contents

- Introduction
- An example: Client/server communication
- Protection against process failures
  - Agreement in process groups
- Reliable group communication
- Distributed commit
- **Recovery techniques**

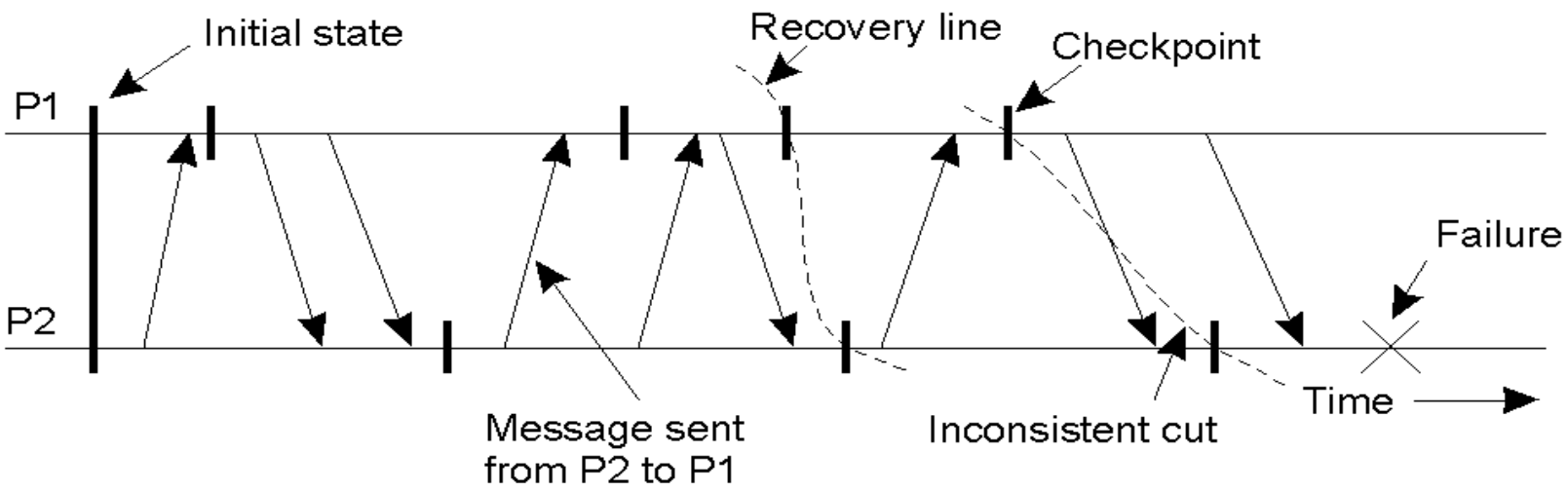
# Recovery: backward vs. forward

- When processes resume working after a failure, they have to be taken back to a correct state
- Backward recovery
  - The system is brought back to a previously saved correct state
- Forward recovery
  - The system is brought into a new correct state from which execution can be resumed
- Example: Implementing reliable communication
  - Retransmission: Backward recovery
  - Erasure correction (i.e.., error correction codes): Forward recovery

# Checkpointing and logging

- Recovering a previous state is only possible if that state can be retrieved
  - Checkpointing consists in periodically saving the distributed state of the system to stable storage
    - It's an expensive operation
  - With Logging, events (messages) are recorded to stable storage so that they can be replayed when recovering from a failure

# Checkpointing: Recovery line

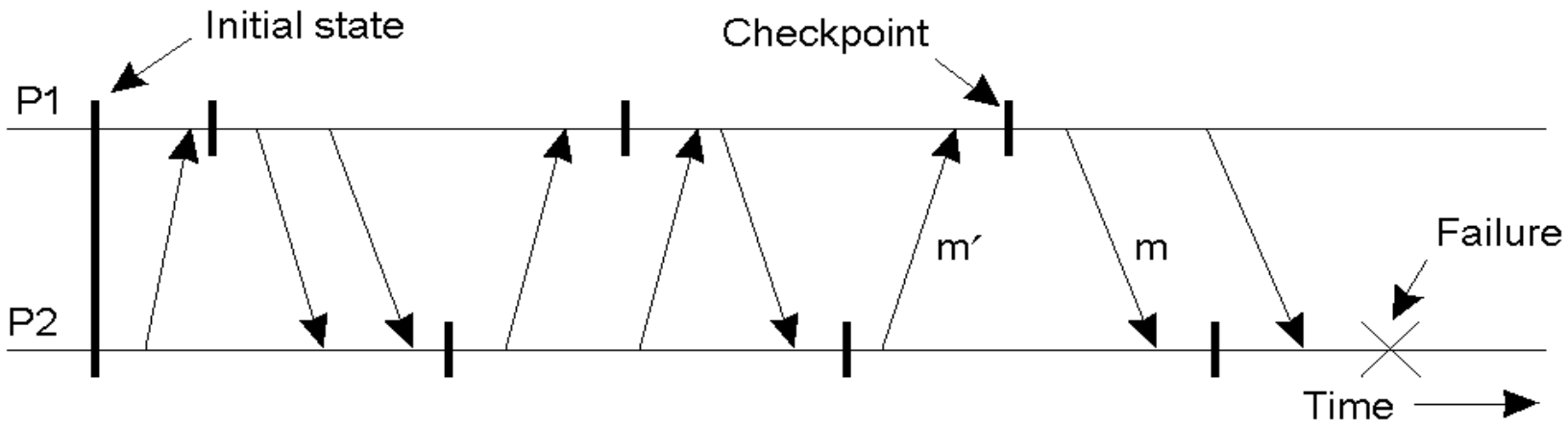



- We need to find a (possibly the most recent) consistent cut

# Independent checkpointing

- The most trivial solution is to have each process independently record its own state
  - We may need to roll back to previous checkpoint until we find a set of checkpoints (one per process) which result in a consistent cut

    → *Domino effect*

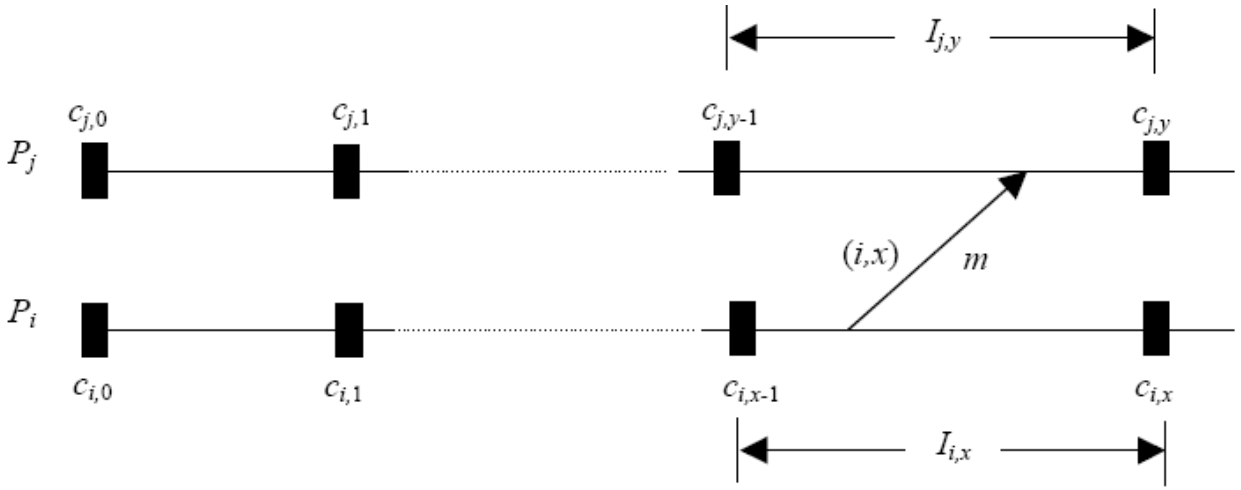


# Independent checkpointing

- Not only does it lead to the domino effect, but it is also not trivial to implement
  - Interval between two checkpoints is tagged
  - Each message exchanged by the processes must record (be labelled with) a reference to the interval
  - This way the receiver can record the dependency between receiving and sending intervals with the rest of the recovery information
  - Intuitively, if process *A* rolls back to a checkpoint before interval *i*, all processes must be rolled back to an interval that does not depend on *i*
    - Messages sent during *i* must not have been received yet

# Independent checkpointing

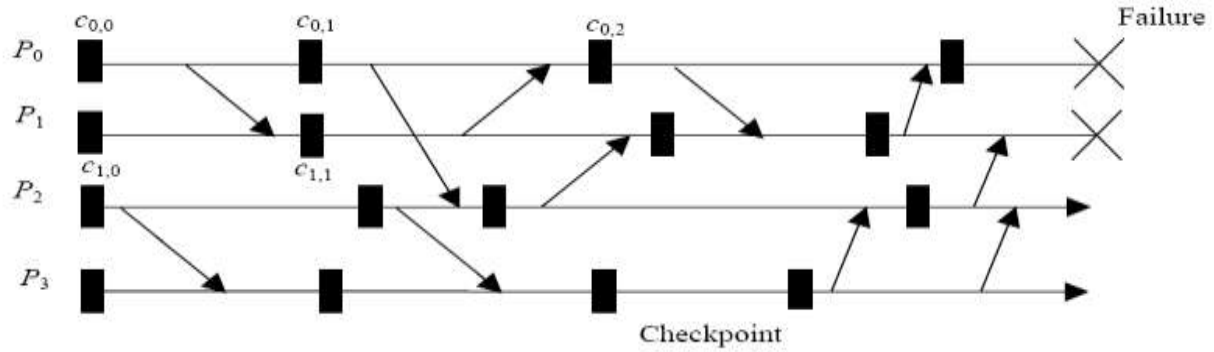
- Let $c_{i,x}$ be the $x_{th}$ checkpoint taken by process $P_i$ and $I_{i,x}$ be the interval between checkpoints $c_{i,x-1}$ and $c_{i,x}$
- When a process sends a message to another, it piggybacks information about it's current checkpoint interval
- The receiver uses this information to record that it's own interval is dependent on the sender's.
  - $I_{i,x}$ -> $I_{j,y}$



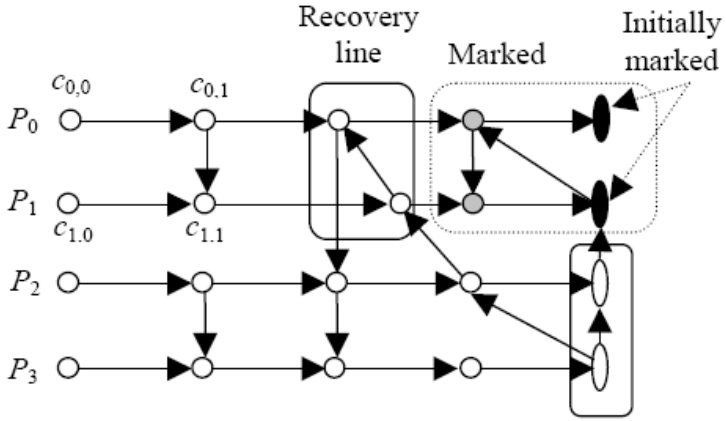
# Checkpoint dependencies

- When a failure occurs, the recovering process broadcasts a dependency request to collect dependency information.
  - All processes stop and send their information
  - The recovering process computes the recovery line and sends the corresponding rollback request
- The recovery line can be computed using two approaches
  - Rollback dependency graph
  - Checkpoint graph

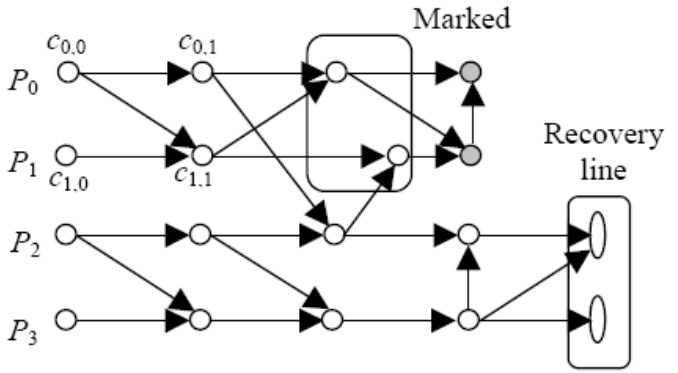
# Computing the recovery line




- Let's consider the situation in the figure; we obtain the following




Rollback-dependency graph




Checkpoint-dependency graph

# Computing the recovery line

- In the rollback-dependency graph we draw an arrow between two checkpoints $c_{i,x}$ and $c_{j,y}$ if either
  - i=j and y=x+1
  - i≠j and a message is sent from $I_{i,x}$ to $I_{j,y}$
- The recovery line is computed by marking the nodes corresponding to the failure states and then marking all those which are reachable from one of them
  - These marked nodes are no good. Each process rolls back to the last unmarked checkpoint
- An equivalent approach can be used on the checkpoint graph. Take the latest checkpoints that do not have dependencies among them

# Coordinated checkpointing

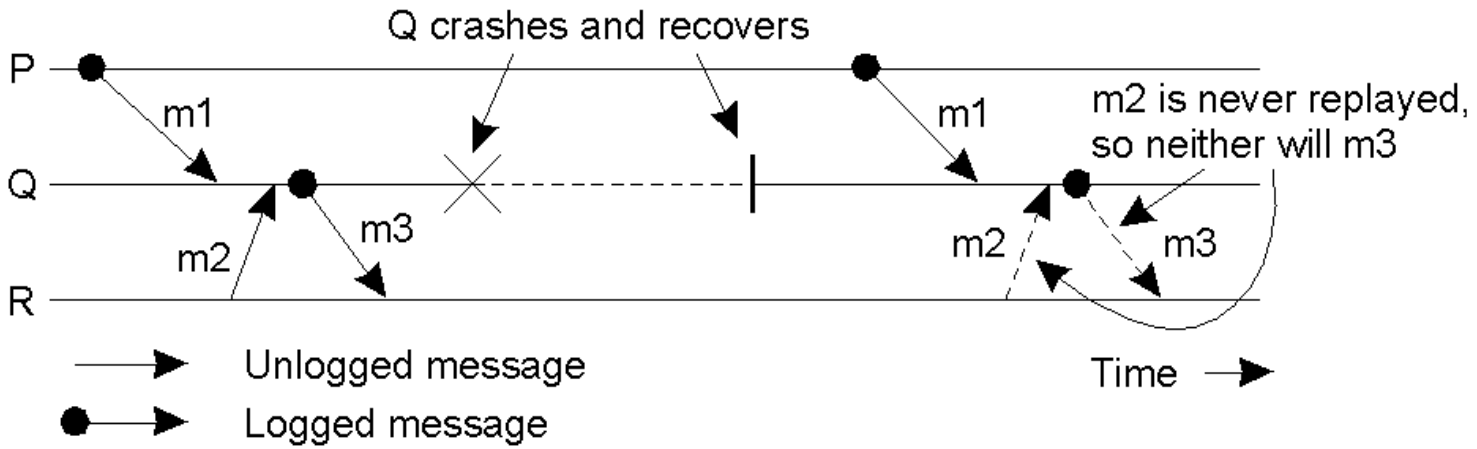
- The complexity of the previous algorithm suggests that independent checkpointing is not so convenient
- The solution is to coordinate checkpoints
  - No useless checkpoints are ever taken
- A simple solution is the following:
  - Coordinator sends CHKP-REQ
  - Receivers take checkpoint and queue other outgoing messages (delivered by applications)
  - When done send ACK to coordinator
  - When all done coordinator sends CHKP-DONE
- An improvement: Incremental snapshot
  - Only request checkpoints to processes that depend on recovery of the coordinator, i.e., those processes that have received a message causally dependent (directly and indirectly) from one sent by the coordinator after the last checkpoint

# Coordinated checkpointing

- Can you think of another algorithm?
  - A global snapshot algorithm can be used to take a coordinated checkpoint
    - The advantage is that processes are not blocked while the checkpoint is being taken
    - The drawback is the increased complexity of the algorithm
- A further alternative which tries to share the benefits of independent and coordinated checkpointing is communication-induced checkpointing
  - Processes take checkpoints independently but piggyback information on messages to allow the other processes to determine whether they should also take a checkpoint

# Logging schemes

- We can start from a checkpoint and replay the actions that are stored in a log file
- The method work if the system is piecewise deterministic: execution proceeds deterministically between the receipt of two messages
- Logging must be done carefully:



# Logging schemes

- Each message's header contains all necessary information to replay the messages
- A message is stable if it can no longer be lost
  - It has been written to stable storage
- For each unstable message *m* define
  - *DEP(m):* Processes that depend on the delivery of *m*, i.e., those to which *m* has been delivered or to which *m'* dependent on *m* has been delivered
  - *COPY(m):* Processes that have a copy of *m*, but not yet in stable storage. They are those processes that hand over a copy of *m* that could be used to replay it

# Logging schemes

- We can then characterize an orphan process in the following way
  - Let *Q* be one of the surviving processes after one or more crashes
  - *Q* is orphan if there exists *m* such that *Q* is in *DEP(m)* and all processes in *COPY(m)* have crashed
    - There is no way to replay *m* since it has been lost
- If each process in *COPY(m)* has crashed then no surviving processes must be left in *DEP(m)*
  - This can be obtained by having all processes in *DEP(m)* be also in *COPY(m)*
  - When a process become dependent on a copy of *m*, it keeps *m*

# Pessimistic vs Optimistic Logging

- *Pessimistic*: Ensure that any unstable message *m* is delivered to at most one process
  - The receiver will always be in *COPY(m)*, unless it discards the message
  - The receiver cannot send any other message until it writes *m* to stable storage
  - If communication is assumed to be reliable then logging should ideally be performed before message delivery
- *Optimistic*: Messages are logged asynchronously, with the assumption that they will be logged before any faults occur
  - If all processes in *COPY(m)* crash, then all processes in *DEP(m)* are rolled back to a state where they are no longer in *DEP(m)*
  - This is similar to the rollback technique used in uncoordinated checkpointing

# Pessimistic vs Optimistic Logging

- Pessimistic logging protocols ensure that no orphan processes are ever created
- Optimistic logging protocols allow orphans to be created and force them to roll back until they are no longer orphans
- A variant of pessimistic logging requires messages to be logged by the sender
- Logging was originally designed to allow the use of uncoordinated checkpointing. However, it has been shown that combining coordinated checkpointing with logging yields better performance