# CHƯƠNG 8: FAULT TOLERANCE

TS. Trần Hải Anh

# Content



1. Introduction to fault tolerance
2. Process resilience
3. Reliable client-Server Communication
4. Reliable Group Communication
5. Distributed Commit
6. Recovery

# 1. Introduction to fault tolerance

1.1. Basic concept

1.2. Failure models

1.3. Failure masking by redundancy

# 1.1. Basic concept



- Being *fault tolerant* related to ***Dependable systems*** *which cover:*
  - Availability
  - Reliability
  - Safety
  - Maintainability
- ***Fail/Fault***
- ***Fault Tolerance***
- ***Transient Faults***
- ***Intermittent Faults***
- ***Permanent Faults***

# 1.2. Failure models



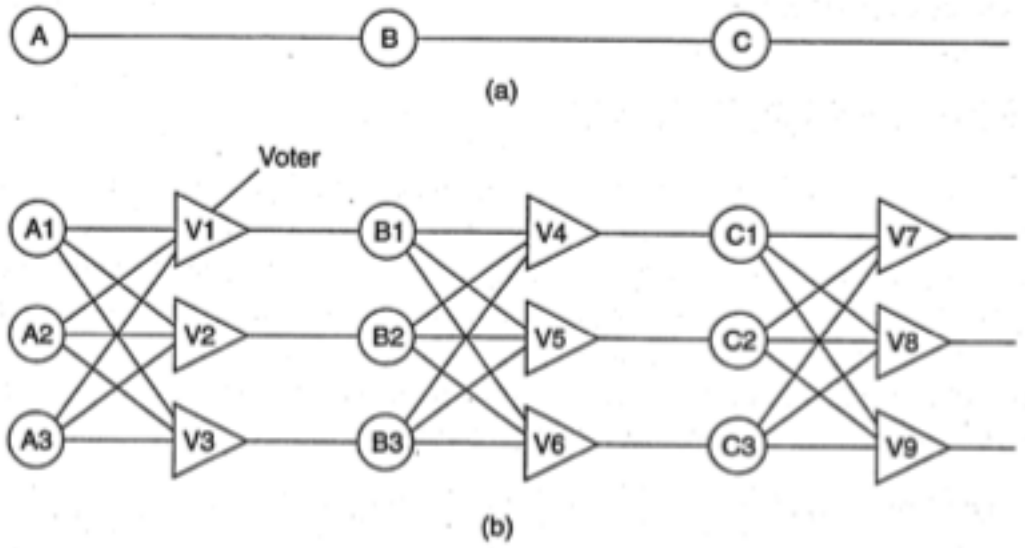
- Different types of failures

| Type of failure | Description |
|---|---|
| Crash failure | A server halts, but is working correctly until it halts |
| Omission failure | Aserver fails to respond to incoming requests |
| Receive omission | A server falls to receive incoming messages |
| Send omission | A server falls to send messages |
| Timing failure | A server's response lies outside the specified time interval |
| Response failure | A 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 | A server may produce arbitrary responses at arbitrary times |
| Fail-stop failure | A server stops producing output and its halting can be detected by other systems |
| Fail-silent failure | Another process may incorrectly conclude that a server has halted |
| Fail-safe | A server produces random output which is recognized by other processes as plain junk |

# 1.3. Failure masking by redundancy



- Three possible kinds for masking failure
  - ***Information redundancy***
  - ***Time redundancy***
  - ***Physical redundancy***
- Triple Modular Redundancy (***TMR***)



# 2. Process resilience



2.1. Design issues

2.2. Failure masking and replication

2.3. Agreement in faulty system

2.4. Failure detection

# 2.1. Design issues (1/3)

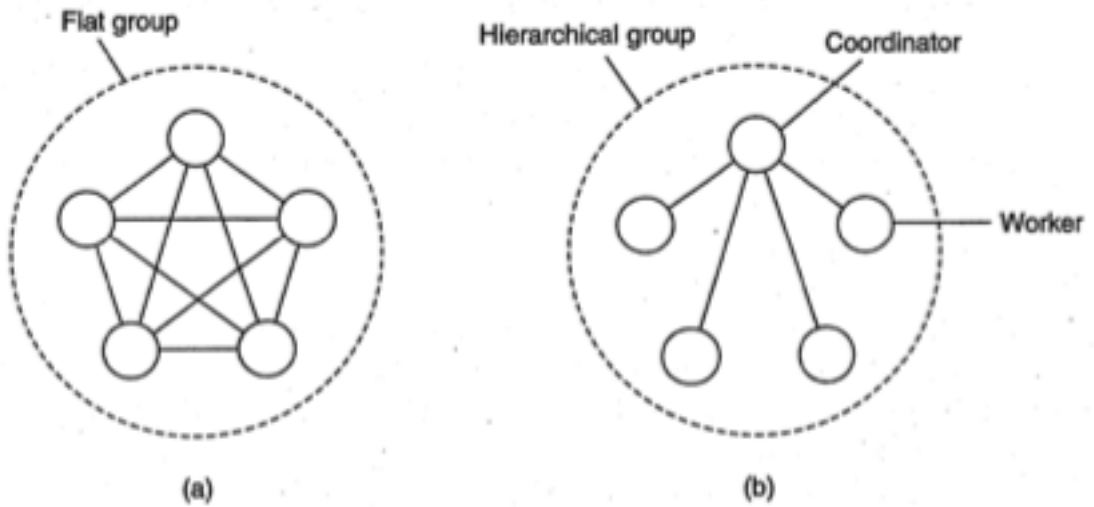


- ***Process group***
  - Key approach: organize several identical processes into a group
  - Key property: message is sent to the group itself and all members receive it
  - Dynamic: create, destroy, join or leave

# 2.1. Design issues (2/3)



- ***Flat Groups*** versus ***Hierarchical Groups***



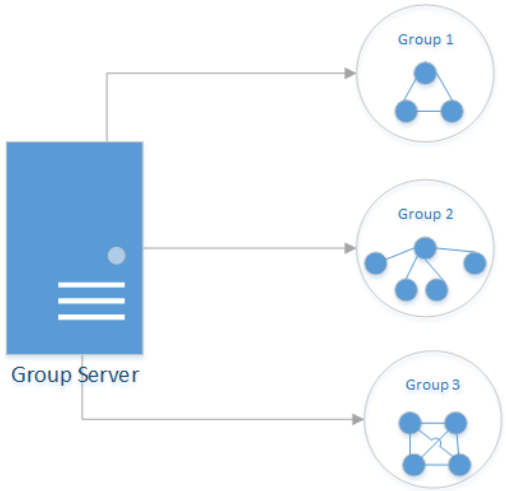

  - Comparison

| | Advantages | Disadvantages |
|---|---|---|
| **Flat Groups** | Symmetrical<br>No single point of failure<br>Group still continues while one of the processes crashes | Complicated decision making |
| **Hierarchical Groups** | Easy decision making | Loss of coordinator brings the group to halt |

# 2.1. Group membership(3/3)



- ***Group Server***




Approach
- Send request
- Maintain databases of all groups
- Maintain their memberships

Disadvantages
- A single point of failure

- ***Distributed way***

Approach - each member communicates directly to all others

Disadvantages
- Fail-stop semantics are not appropriate
- Leaving and joining must be synchronous with data messages being sent

- ***Membership issues***

What happens when multiple machines crash at the same time?

# 2.2. Failure masking and Replication



- ***Primary-based protocols***
- Used in form of primary-backup protocol
- Organize group of processes in hierarchy
- Backups execute election algorithm to choose a new primary

- ***Replicated-write protocols***
- Used in form of *active replication* or *quorum-based protocols*
- Organize a collection of identical processes into a flat group
- Called *'k fault tolerant'* if system can survive faults in k components.

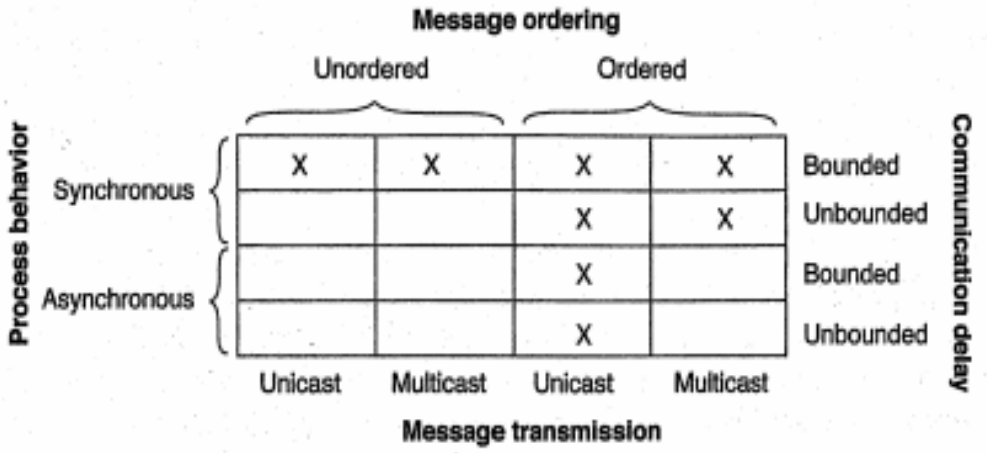
# 2.3. Agreement in Faulty systems (1/3)



- ***Different cases***

1. Synchronous versus asynchronous system
2. Communication delay is bounded or not
3. Message delivery is ordered or not
4. Message transmission is done through unicasting or multicasting

- ***Circumstances under which distributed agreement can be reached***



# 2.3. Agreement in Faulty systems (2/3)



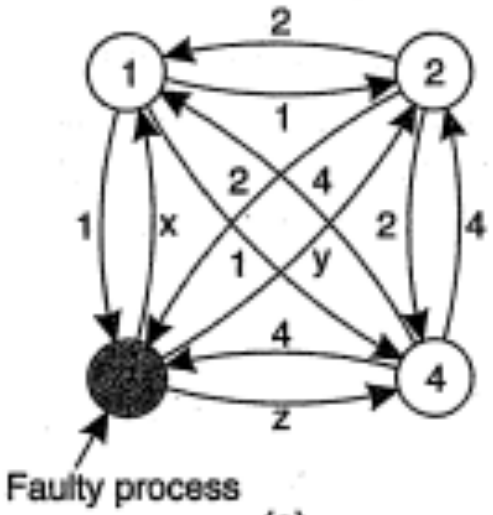
- ***Byzantine agreement***

Assuming $N$ processes, each process $i$ provides a value $v_i$

Goal: construct a vector $V$ of length $N$

If $i$ is nonfaulty then $V[i] = v_i$

- ***Example:*** $N = 4$ and $k = 1$




(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)

# 2.3. Agreement in Faulty systems (3/3)



- ***Lamport et al. (1982)*** proved that agreement can be achieved if
- $2k+1$ correctly process for total of $3k + 1$, with $k$ faulty processes

(or more than 2/3 correctly process with $2k+1$ nonfaulty processes)

- ***Fisher et al. (1985)*** proved that where messages is not delivered within a known and finite time -> No possible agreement if even only one process is faulty because arbitrarily slow processes are indistinguishable from crashed ones

# 2.4. Failure Detection



- Two mechanisms - ***Active process and Passive Process***
- ***Timeout mechanism*** is used to check whether a process has failed. Main disadvantages:
  - Possible wrong detection when simply stating failure due to unreliable networks. Thus, generate false positives and a perfectly healthy process could be removed from the membership list
  - Failure detection is plain crude, based only on the lack of a reply to a single message
- How to ***design*** a failure ***detection subsystem***?
  - Through gossiping
  - Through probe
  - Regular information exchange  with neighbors -> a member for which the availability information is old, will presumably have failed
- Failure detection ***subsystem ability***?
  - Distinguish network failures from node failures by letting nodes decide whether one of its neighbors has crashed
  - Inform nonfaulty processes about the failure detection using **FUSE** approach

16

# 3. Reliable Client-Server Communication

3.1. Point-to-Point Communication

3.2. RPC Semantics in the Presence of Failures

# 3.1. Point-to-Point Communication



- Point-to-point communication is established by using **reliable transport protocols**
  - TCP masks omission failures by using acknowledgments and retransmissions -> failure is hidden from TCP client

  - Crash failures cannot be masked because TCP connection is broken

    -> client is informed through exception raised

    -> Let the distributed system automatically set up a new connection

# 3.2. RPC Semantics in the Presence of Failures (1/5)



- **RPC (Remote Procedure Calls)** hides communication by remote procedure calls
- **Failures occur** when:
  - Client is unable to locate the server
  - Request message from the client to the server is lost
  - Server crashes after receiving a request
  - Reply message from the server to the client is lost
  - Client crashes after sending a request

# 3.2. RPC Semantics in the Presence of Failures (2/5)



- Client is unable to locate the server, e.g. the client cannot locate a suitable server, or all servers are down…

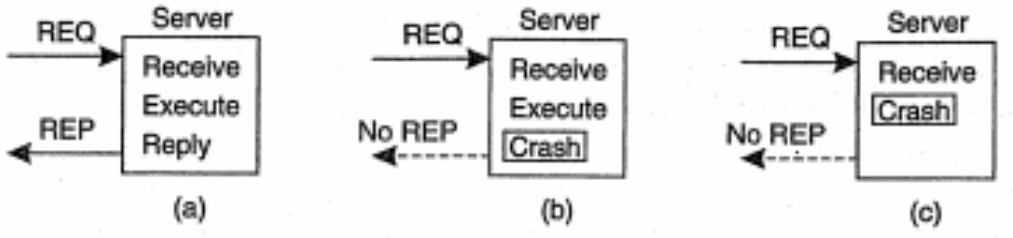
  -> Solution: raise **Exception**

  Drawbacks:
  - not every language has exceptions or signals.
  - Exception destroys the transparency
- Lost request Messages, detected by setting a timer
  - Timer expires before a reply or ack -> resend message
  - True loss -> no difference between retransmission and original
  - So many messages lost -> client gives up and concludes that the server is down, which is back to "Cannot locate server"
  - No message lost: let the server to detect and deal with retransmission

# 3.2. RPC Semantics in the Presence of Failures (3/5)



- Server Crashes




(a) Normal Case (b) Crash after execution (c) Crash before execution

**Difficult to distinguish between (b) and (c)**

- (b) the system has to report failure back to the client
- (c) need to retransmit the request

**3 philosophies for servers:**

- At least once semantics
- At most once semantics
- Exactly once semantics

**4 strategies for the client**

- Client decide to never reissue a request
- Client decide to always reissue a request
- Client decide to reissue a request only when no acknowledgment received
- Client decide to reissue a request only when receiving acknowledgment

# 3.2. RPC Semantics in the Presence of Failures (4/5)



- Server Crashes (next)

**8 considerable combinations but none is satisfactory**

- 3 events: M (send message), P (print text), C (crash)
- **6 orderings** **All possible combinations**

1. *M -> P -> C*
2. *M -> C (-> P)*
3. *P -> M -> C*
4. *P -> C –(> M)*
5. *C (-> P -> M)*
6. *C (-> M -> P)*

| 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 |

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

**Conclusion**

- The possibility of server crashes changes the nature of RPC and distinguishes single-processor systems from distributed systems
- In former case, a server crash also implies a client crash

# 3.2. RPC Semantics in the Presence of Failures (5/5)



- Lost Reply Messages
  - **Solution: rely on a timer** set by client's operating system

Difficulty -> The client is not really sure why there was no answer: lost or slow?

- **Idempotent request:** asking for the first 1024 bytes of a file has no side effects and executing as often as necessary without any harm
- **Assign sequence number:** server keeps track of the most recently received sequence number from each client and refuse to carry out any request a second time

- Client crashes
  - **Solution: activate computation called "orphan"**

Difficulty:

- Waste CPU cycles
- Lock files or tie up valuable resources
- Confusion if the client reboots and does RPC again

- **Alternative solutions:**
  - Orphan extermination
  - Reincarnation
  - Gentle Reincarnation
  - Expiration

# 4. Reliable Group Communication

4.1. Basic Reliable – Multicasting Schemes

4.2. Scalability in Reliable Multicasting

4.3. Atomic Multicast

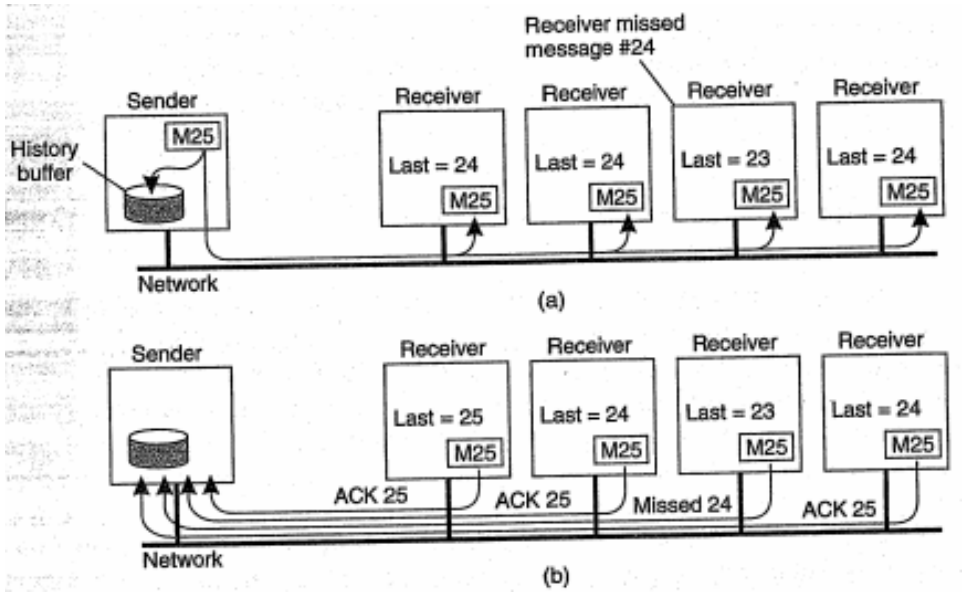
# 4.1. Basic Reliable – Multicasting Schemes



- **Multicasting** means that a message sent to a process group, should be delivered to each member of that group
- **In presence of faulty process**: multicasting is reliable when all nonfaulty group members receive the message
- Solution to reliable multicasting when all receivers are known and assumed not to fail

(a) Message Transmission

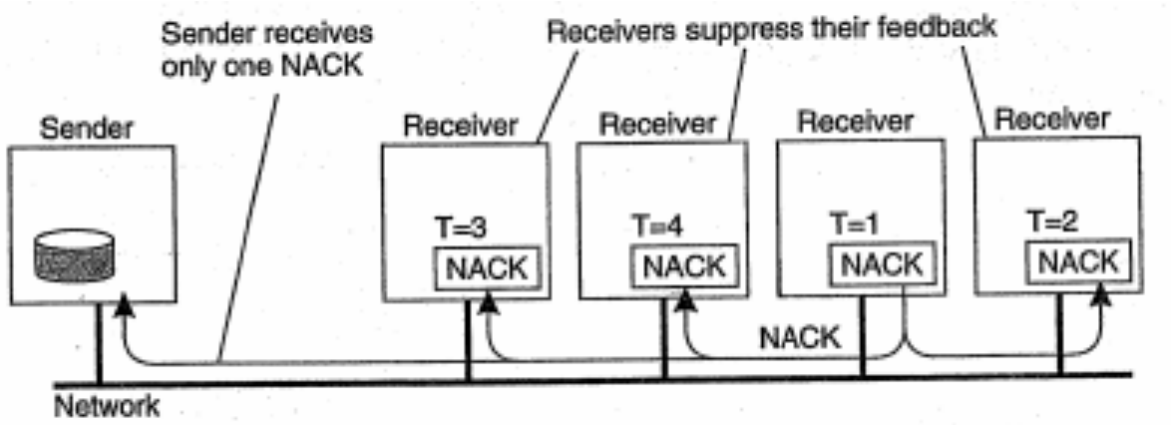
(b) Reporting feedback



# 4.2. Scalability in Reliable Multicasting (1/2)



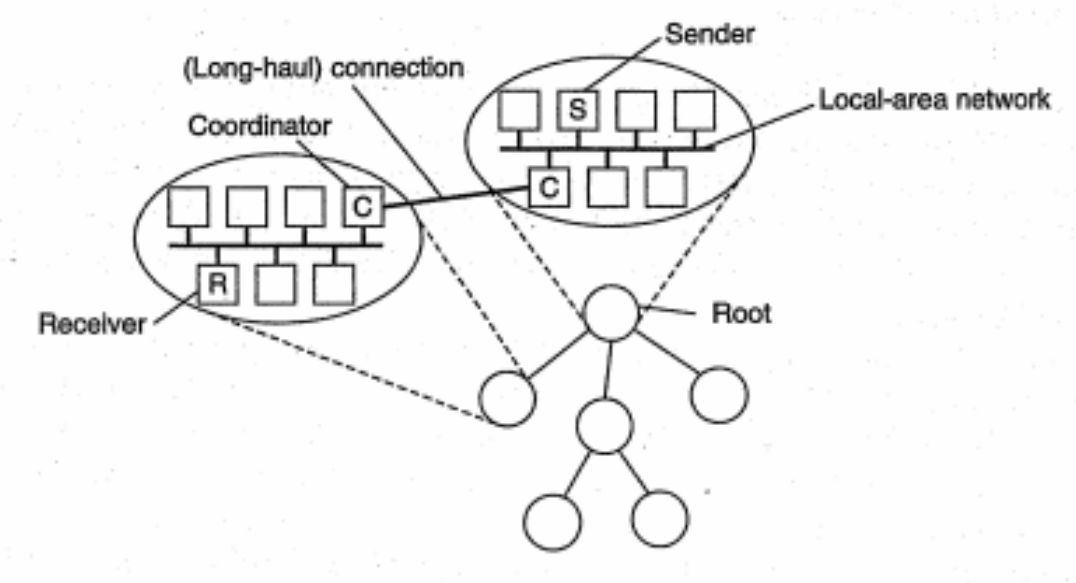
- **Problem of reliable multicast scheme** it that cannot support large numbers of receivers
- **Nonhierarchical feedback control**
  - Key: reduce the number of feedback messages returned
  - Model: feedback suppression which underlies the scalable reliable multicasting (SRM)
  - In SRM, receiver reports when missing message and multicasts its feedback to the rest of the group. Other group members will suppress its own feedback.



# 4.2. Scalability in Reliable Multicasting (2/2)



- **Hierarchical feedback control**
  - Achieving scalability for very large groups of receivers requires adopting hierarchical approaches
  - Each local coordinator forwards the message to its children and later handles retransmission requests




  - Main problem: construction of the dynamic is not easy

# 4.3. Atomic Multicast (1/6)



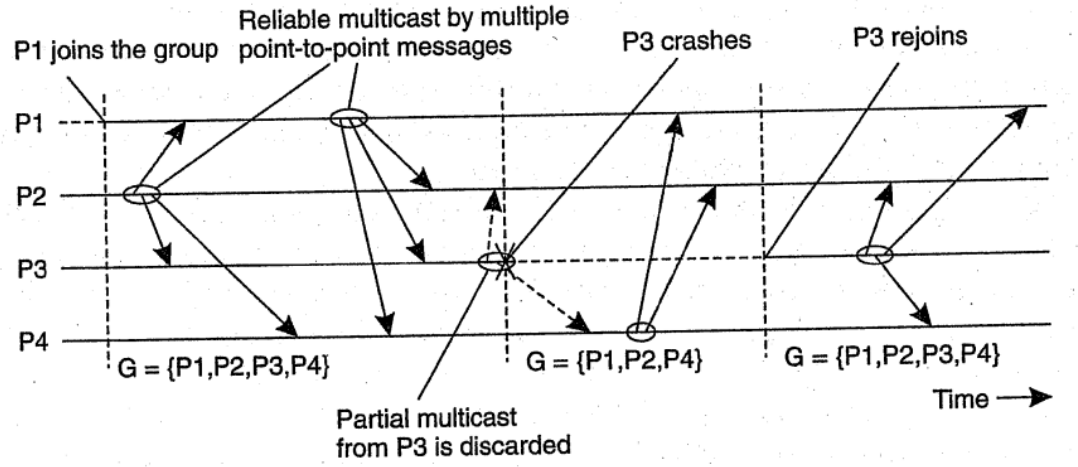
- **Atomic multicast:**
  - Guarantee that a message is delivered to either all processes or to non at all.
  - All messages are delivered in the same order to all processes
  - In non-atomic multicast, when there are multiple updates and a replica crashes, it is difficult to locate operations missing and the order these operations are to be performed
  - In atomic multicast, when replica crashes, it ensures that nonfaulty processes maintain a consistent view of the database and force reconciliation when a replica recovers and rejoins the group

# 4.3. Atomic Multicast (2/6)



- **Virtual Synchrony**

To distinguish between receiving and delivering message, adopt **distributed system model which consists of communication layer**




- Multicast message ***m*** **is associated with a list of processes** to which it should be delivered, named **group view**
- Each process on that list has the same view.
- Message m, group view G. While the multicast is taking place, another process joins or leaves the group -> **View change –** multicast a message *vc* announcing the joining or leaving of a process -> two multicast messages in transit: m and vc

# 4.3. Atomic Multicast (4/6)



- **Message Ordering**
  - Unordered multicasts

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

Sample of three communicating processes in the same group -> the ordering of events per process is shown along the vertical axis

  - FIFO-ordered multicasts

| Process P1 | Process P2 | Process P3 | Process P3 |
|---|---|---|---|
| sends m1 | receives m1 | receives m3 | receives m3 |
| sends m2 | receives m3 | receives m1 | receives m4 |
| | receives m2 | receives m2 | |
| | receives m4 | receives m4 | |

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

  - Causally-ordered multicasts
  - Totally-ordered multicasts

# 4.3. Atomic Multicast (5/6)

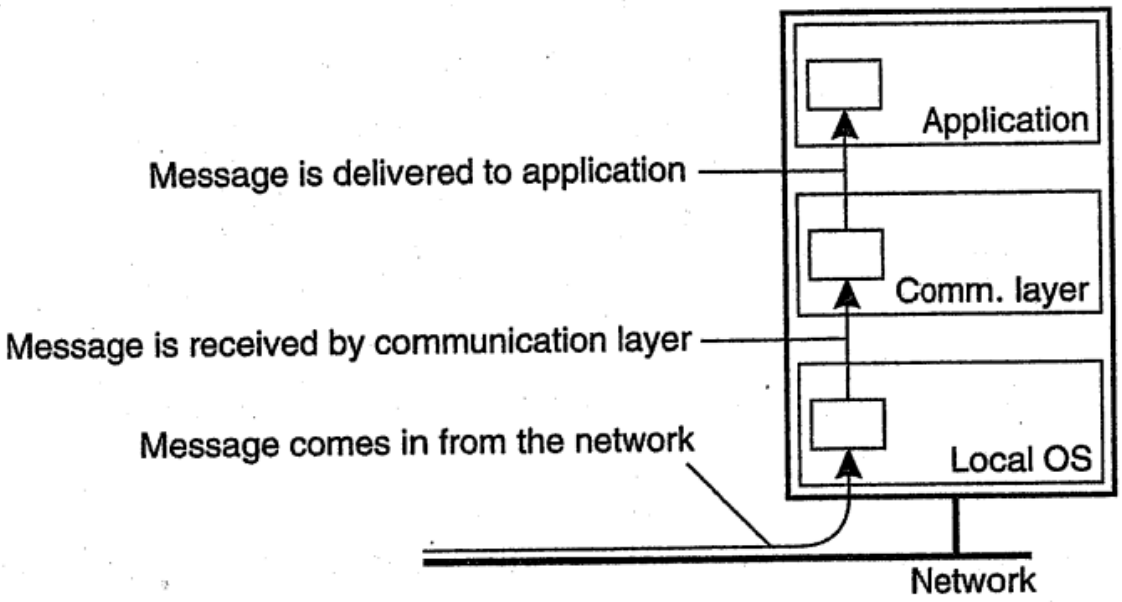


- **Implementing Virtual Synchrony**

Goal: Guarantee that all messages sent to view G are delivered to all nonfaulty processes in G before the view change.

Solution: Let every process in G keep m until it knows for sure that all members in G have received it.

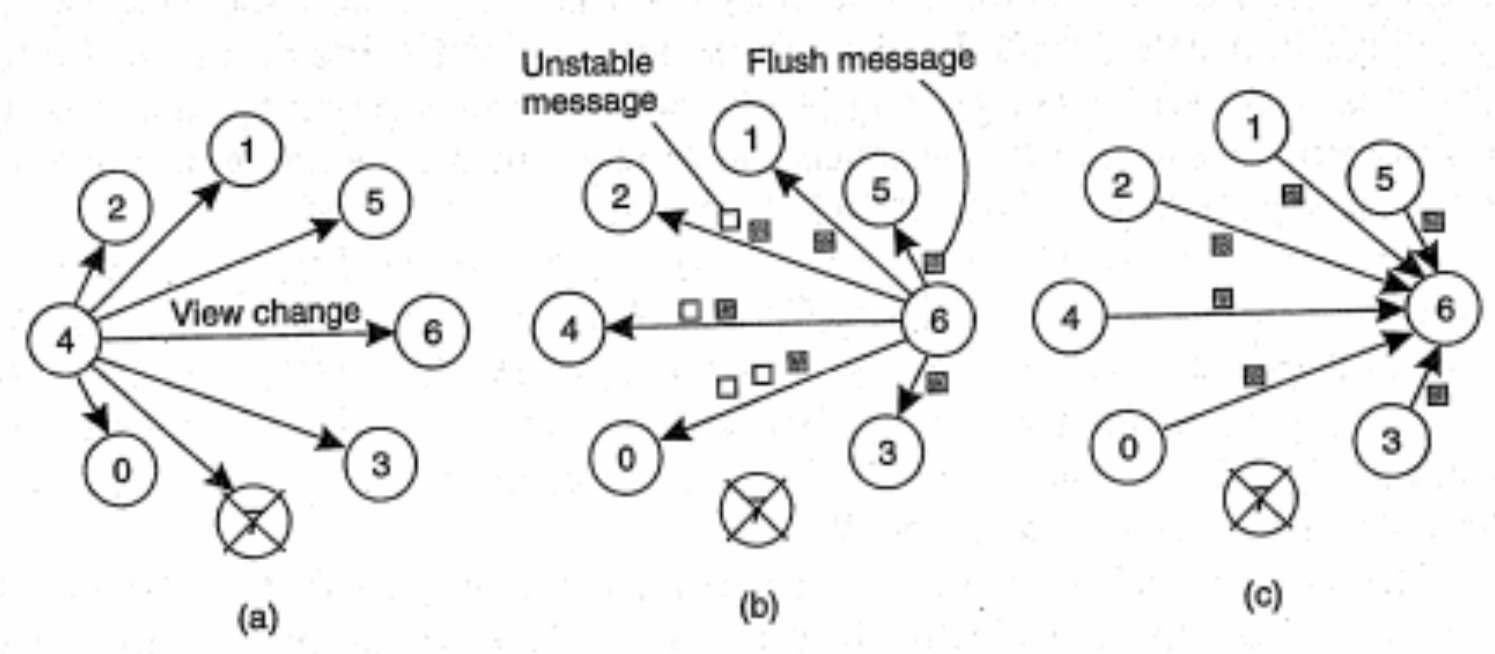
Stable message



# 4.3. Atomic Multicast (6/6)



- **Implementing Virtual Synchrony**
- Illustration of selecting stable message
  a) Process 4 notices that process 7 has crashed and sends a view change
  b) Process 6 sends out all its unstable messages and subsequently marks it as being stable, followed by a flush message
  c) Process 6 installs the new view when it has received a flush message from everyone else



# 5. Distributed Commit

5.1. Two-Phase Commit

5.2. Three-Phase Commit

# About Distributed Commit



- Distributed commit involves **having an operation being performed by each member of a process group, or non at all**
  - Reliable multicasting: Operation = message delivery
  - Distributed transactions: Operation = transaction commit at the single site that takes part in the transaction
- Distributed commit is established by means of **coordinator**
- **One-phase commit protocol:** a simple scheme where a coordinator tells all other processes (called participants) whether or not to perform the operation in question.
- **Sophisticated schemes**: Two-phase commit or Three-phase commit

# 5.1. Two-Phase Commit - 2PC (1/5)



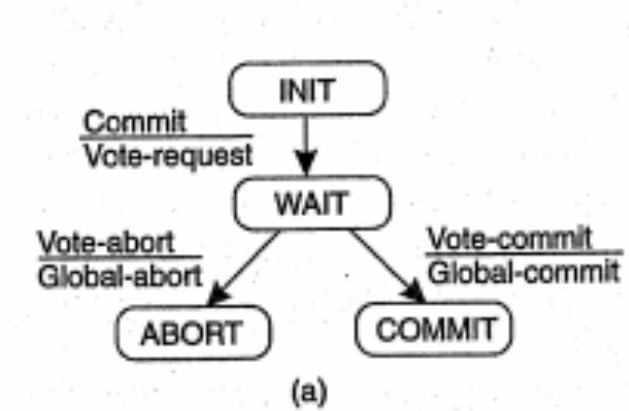
- Protocol consists **two phase**:

**Phase 1**

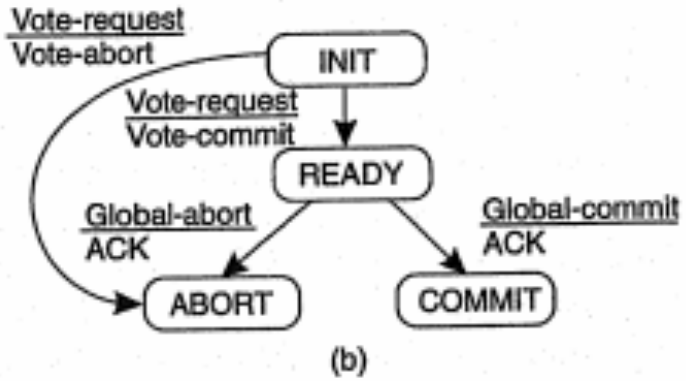
- ✓ Coordinator sends a VOTE_REQUEST message to all participants
- ✓ After receiving, participant returns VOTE_COMMIT or VOTE_ABORT message to the coordinator

**Phase 2**

- ✓ Coordinator collects all votes and send GLOBAL_COMMIT message or GLOBAL_ABORT message to participants
- ✓ Each participant that voted for a commit waits for the final reaction to commit or not the transaction




**Coordinator**




**Participant**

# 5.1. Two-Phase Commit - 2PC (2/5)



- **Participant Solution:**
  - use timeout mechanism or let a participant P contact
  - Let a participant P contact another participant Q and decide what it should do. If P is in READY status, here are various options

| State of Q | Action by P |
|---|---|
| COMMIT | Make transition to COMMIT |
| ABORT | Make transition to ABORT |
| INIT | Make transition to ABORT |
| READY | Contact another participant |

# 5.1. Two-Phase Commit - 2PC (3/5)



- Sample of actions taken in place by the 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;
}
```

# 5.1. Two-Phase Commit - 2PC (4/5)



- Each participant should be prepared to accept requests for a global decision from other participants

**Actions for handling decision requests:** /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}
```

# 5.1. Two-Phase Commit - 2PC (5/5)



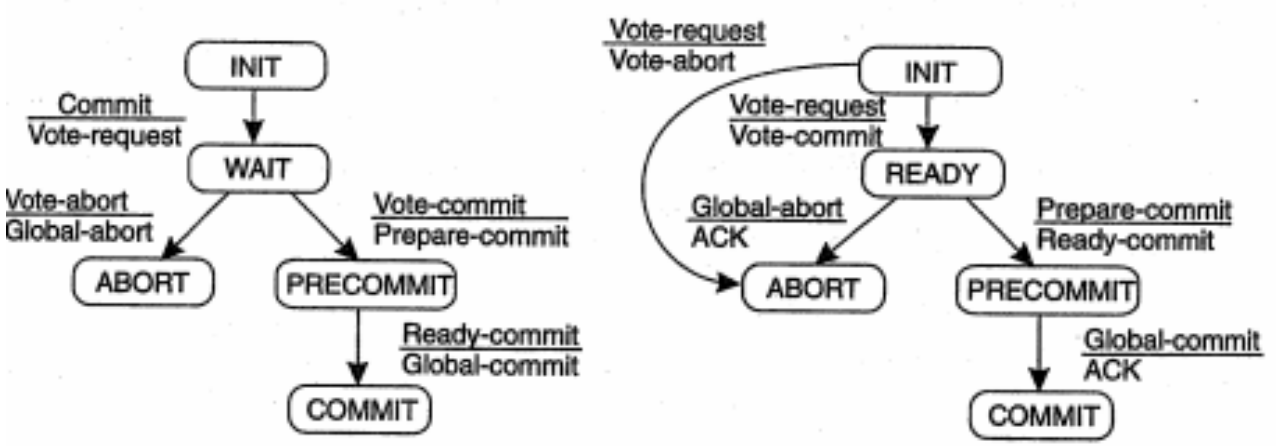
- **Coordinator solution**
  - Keep track of current state
  - Sample of actions taken in place by the coordinator:

```
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;
}
```

# 5.2. Three-Phase Commit (1/2)



- Two-phase problem: when the coordinator has crashed, participants may not be able make final decision
- Three-phase commit protocol (3PC) avoids blocking processes in when fail-stop crashes.
- Principle:
  - There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state
  - There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made
- Illustration



# 5.2. Three-Phase Commit (2/2)



- Actions taken by Participant in different cases

| State of Participant P | State of Participant Q | State of all other participants | Action |
|---|---|---|---|
| INT | | | VOTE_ABORT |
| READY | INT | | VOTE_ABORT |
| READY | READY | READY | VOTE_ABORT |
| READY | PRECOMMIT | PRECOMMIT | VOTE_COMMIT |
| PRECOMMIT | READY | READY | VOTE_ABORT |
| PRECOMMIT | PRECOMMIT | PRECOMMIT | VOTE_COMMIT |
| PRECOMMIT | COMMIT | COMMIT | VOTE_COMMIT |

- Actions taken by Coordinator in different cases

| State of Coordinator | Action |
|---|---|
| WAIT | GLOBAL_ABORT |
| PRECOMMIT | GLOBAL_COMMIT |

- Main difference with 2PC: if any participant is in READY state, no crashed process will recover to a state other than INT, ABORT or PRECOMMIT

# 6. Recovery



6.1. Introduction

6.2. Checkpointing

# 6.1. Introduction (1/2)



- **Backward recovery**: bring the system into a previously correct state.
  - Necessary to record the system's state, called checkpoint
  - Generally applied for recovering from failures in distributed systems
  - E.g. Reliable communication through packet retransmission
  - Drawback:
    - reduce performance
    - no guarantees that recovery has taken place
    - some states can never be rolled back to.
    - checkpoint could penalize performance and is cosly
  - Solution for checkpoint: combine with message logging or use receiver-based logging
- **Forward recovery**: bring the system in a correct new state from which it can continue to execute
  - E.g. Erasure correction- a missing packet is constructed from other; successfully delivered packets

# 6.1. Introduction (2/2)

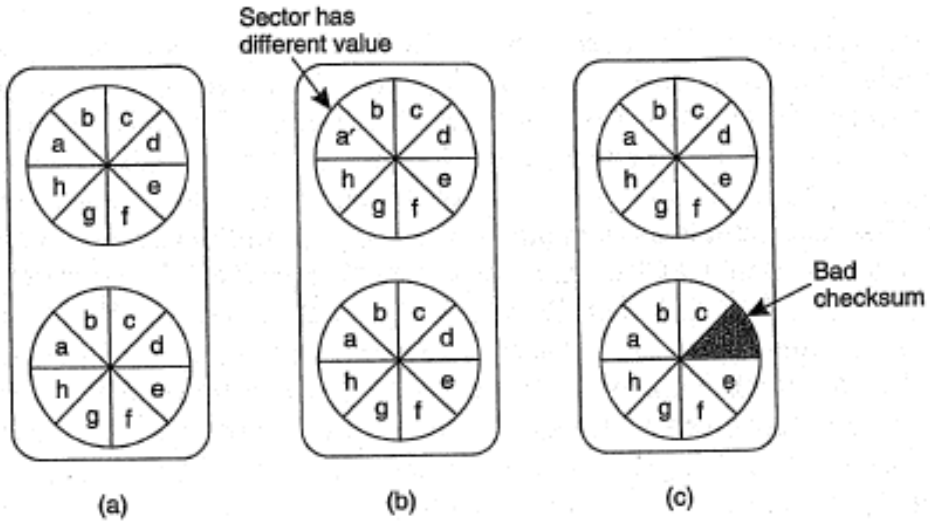


- **Stable Storage**
  - Information needed to enable recovery is safely stored in case of process crashes, site failures or various storage media failures
  - Three categories of storage: RAM memory, disk storage and stable storage
  - Sample of stable storage implementing with a pair of ordinary disk

(a) Stable storage

(b) Crash after drive 1 is updated

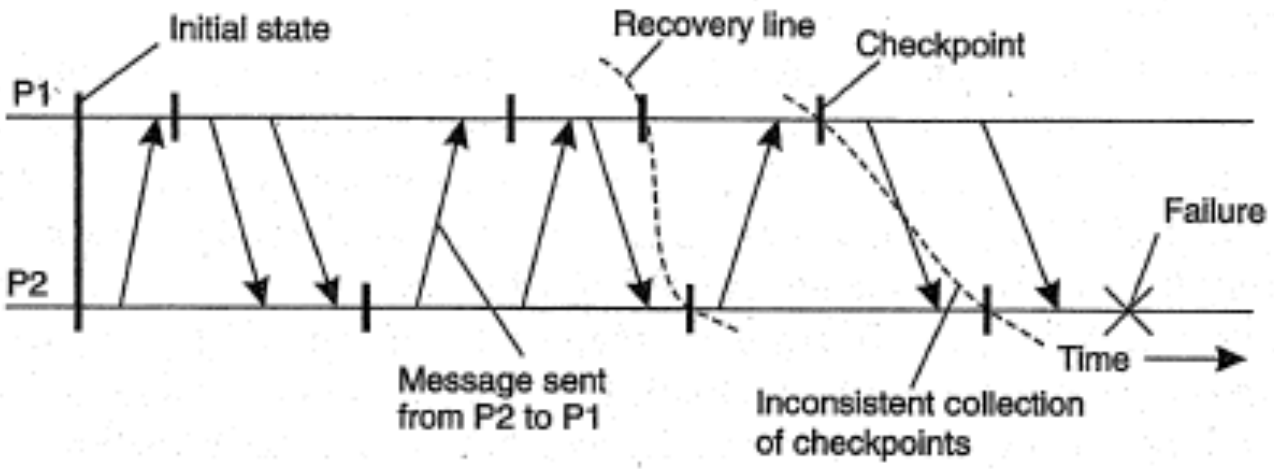
(c) Bad spot



# 6.2. Checkpointing (1/3)



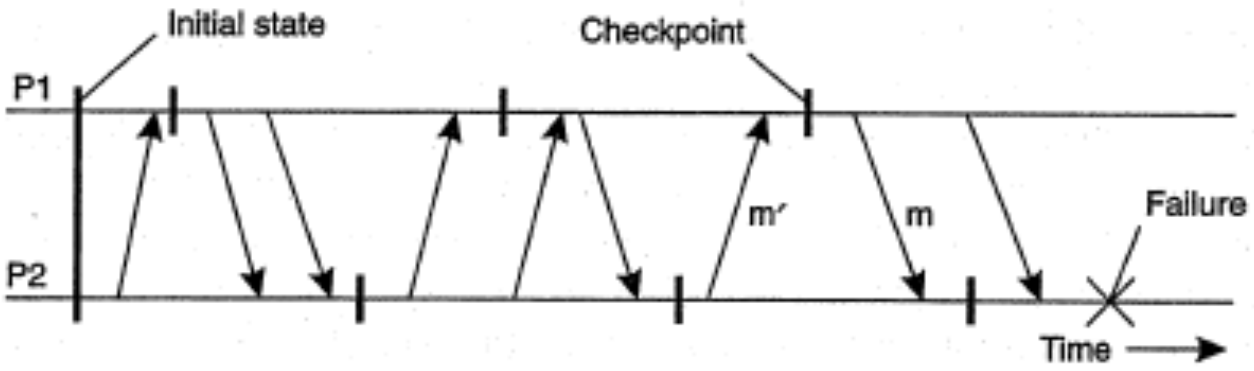
- **Distributed snapshot:** record a consistent global state.
  - If a process P records the receipt of a message, then there should also be a process Q that has recorded the sending of that message.
- **Recovery line**: recover to the most recent distributed snapshot



# 6.2. Checkpointing (2/3)



- **Independent Checkpointing**
  - Domino effect: process to find a recovery line via cascaded rollback




  - **Independent checkpointing**: processes take local checkpoints independent of each other.
  - **Disadvantages**: Introduction of performance problem, need of periodical cleaning for local storage, difficult problem in computing the recovery line

# 6.3. Message Logging (1/3)



- **Idea:** if the transmission of messages can be replayed, we can still reach a globally consistent state but without having to restore that state from stable storage
- **Solution:** take a checkpointed state as a starting point, all messages sent will be retransmitted and handled accordingly
- **Assumption:** *piecewise deterministic model*, the execution of each process is assumed to take place as a series of intervals in which events take place
- **Alvisi & Marzullo**: many existing message-logging schemes can be easily characterized if we concentrate on how they deal with orphan processes
- **Orphan process** is a process that survives the crash of another process, but whose state is inconsistent with the crashed process after its recovery

# 6.3. Message Logging (2/3)



- **Characterizing Message – Logging Schemes**
  - Each message m is considered to have a **header containing all information** to retransmit m and to handle it
  - A **stable message** is used for recovery by replying their transmission
  - Each message m leads to a set DEP(m) of processes that depend on the delivery of m
  - If another message m' is dependent on the delivery of m, and m' has been delivered to a process Q, then Q will also be contained in DEP(m)
  - The set COPY(m) consists of those processes that have a copy of m, but not in their local stable storage. When Q delivers m, it becomes a member of COPY(m)

# 6.3. Message Logging (3/3)



- **Characterizing Message – Logging Schemes (next)**
  - Suppose that Q is one of the surviving processes after a crash in COPY(m) -> **Q is an orphan process** which is dependent on m, but cannot replay m's transmission
  - To avoid orphan processes -> ensure that if each process in COPY(m) crashed, no surviving process is left in DEP(m)
  - **Pessimistic logging protocols** ensure that each nonstable message m is delivered to at most one process.
  - **Optimistic logging protocol**: any orphan process in DEP(m) is rolled back to a state in which it no longer belongs to DEP(m)

# 6.4. Recovery-Oriented Computing



- **Approach: start over again**
- **Solution 1: reboot part of a system**
  - Delete all instances of the identified components with threads and restart the associated requests.
  - Solution requires that components are largely decoupled and no dependencies between components.

- **Solution 2: apply checkpointing and recovery techniques**
  - Give more buffer space to programs, clear memory before allocated, changing the ordering of message delivery
  - Tackle software failures

# Q&A