Unit IV: Fault Tolerance, Recovery and Distributed Transaction

References:

1. G. Coulouris, J. Dollimore and T. Kindberg; **Distributed Systems Concepts and Design,4th Edition.**
2. Andrew S. Tanenbaum and Maarten van Steen**; Distributed Systems: Principles and Paradigms, 2nd Edition.**

Fault Tolerance, Recovery and Distributed Transaction

Faults

- A system **fails** when it cannot meet its promises (specifications)
- An **error** is part of a system state that may lead to a failure
- <u>A **fault** is the cause of the error</u>
- **Fault-Tolerance**: the system can provide services even in the presence of faults
- Faults can be:
  - Transient (appear once and disappear)
  - Intermittent (appear-disappear-reappear behavior)
    - A loose contact on a connector ➔intermittent fault
  - Permanent (appear and persist until repaired)

Fault Tolerance

- A DS should be fault-tolerant
  - Should be able to continue functioning in the presence of faults
- Fault tolerance is related to **dependability**
- Dependability Includes
  - Availability
  - Reliability
  - Safety
  - Maintainability

Availability & Reliability (1)

- **Availability**: A measurement of whether a system is ready to be used immediately
  - System is available at any given moment
- **Reliability**: A measurement of whether a system can run continuously without failure
  - System continues to function for a long period of time
- A system goes down 1ms/hr has an availability of more than 99.99%, but is unreliable
- A system that never crashes but is shut down for a week once every year is 100% reliable but only 98% available

Safety & Maintainability

- **Safety**: A measurement of <u>how safe failures are</u>
  - System fails, nothing serious happens
  - For instance, high degree of safety is required for systems controlling nuclear power plants
- **Maintainability**: A measurement of <u>how easy it is to repair a system</u>
  - A highly maintainable system may also show a high degree of availability
  - Failures can be detected and repaired automatically? Self-healing systems?

System reliability: Fault-Intolerance vs. Fault-Tolerance

- The fault intolerance (or fault-avoidance) approach improves system reliability by removing the source of failures (i.e., hardware and software faults) before normal operation begins

- The approach of fault-tolerance expect faults to be present during system operation, but employs design techniques which insure the continued correct execution of the computing process

Failure Models

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

Issues

- Process Deaths:
  - All resources allocated to a process must be recovered when a process dies
  - Kernel and remaining processes can notify other cooperating processes
  - Client-server systems: client (server) process needs to be informed that the corresponding server (client) process died
- Machine failure:
  - All processes running on that machine will die
  - Client-server systems: difficult to distinguish between a process and machine failure
  - Issue: detection by processes of other machines
- Network Failure:
  - Network may be partitioned into subnets
  - Machines from different subnets cannot communicate
  - Difficult for a process to distinguish between a machine and a communication link failure

Approaches to fault-tolerance

Approaches:

(a) Mask failures

(b) Well defined failure behavior

Mask failures:

- System continues to provide its specified function(s) in the presence of failures
- Example: voting protocols

Well defined failure behaviour:

- System exhibits a well define behaviour in the presence of failures
- It may or it may not perform its specified function(s), but facilitates actions suitable for fault recovery
- Example: commit protocols
  - A transaction made to a database is made visible only if successful and it commits
  - If it fails, transaction is undone

Redundancy:

- Method for achieving fault tolerance (multiple copies of hardware, processes, data, etc...)

Failure Masking

- Redundancy is key technique for hiding failures
- Redundancy types:

1. Information: add extra (control) information
    - Error-correction codes in messages
2. Time: perform an action persistently until it succeeds:
    - Transactions
3. Physical: add extra components (S/W & H/W)
    - Process replication, electronic circuits

Voting protocols

- **Principles:**
  - Data replicated at several sites to increase reliability
  - Each replica assigned a number of votes
  - To access a replica, a process must collect a majority of votes
- **Vote mechanism**:

  (1) Static voting:
  - Each replica has number of votes (in stable storage)
  - A process can access a replica for a read or write operation if it can collect a certain number of votes (*read* or *write quorum*)

  (2) Dynamic voting
  - Number of votes or the set of sites that form a quorum change with the state of system (due to site and communication failures)

  (2.1) Majority based approach:
  - Set of sites that can form a majority to allow access to replicated data of changes with the changing state of the system

  (2.2) Dynamic vote reassignment:
  - Number of votes assigned to a site changes dynamically

Failure resilient processes

- Resilient process: continues execution in the presence of failures with minimum disruption to the service provided (masks failures)
- Approaches for implementing resilient processes:
  - Backup processes and
  - Replicated execution

(1) Backup processes

- Each process made of a primary process and one or more backup processes
- Primary process execute, while the backup processes are inactive
- If primary process fails, a backup process takes over
- Primary process establishes checkpoints, such that backup process can restart

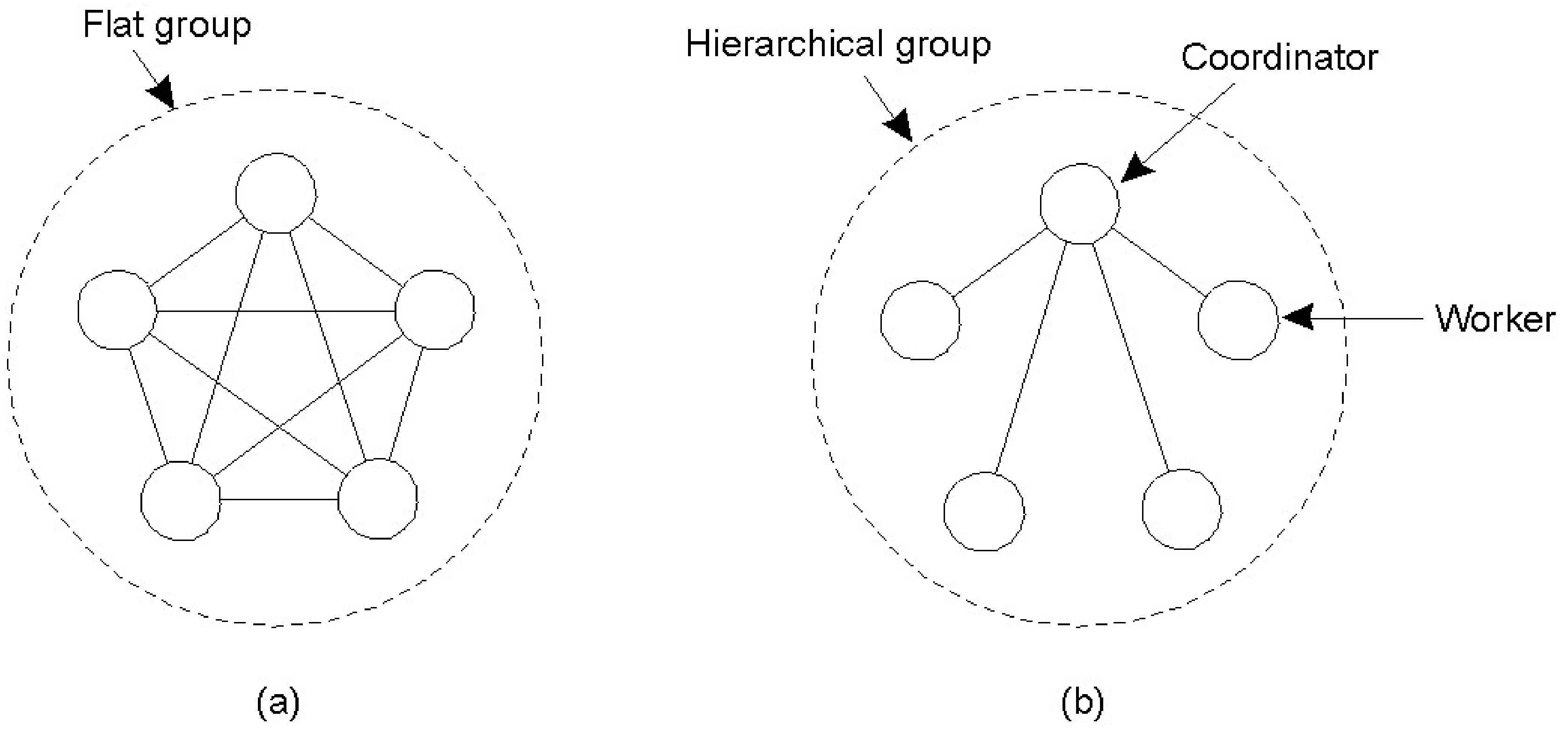
(2) Replicated execution

- Several processes execute same program concurrently
- Majority consensus (voting) of their results
- Increases both the reliability and availability of the process

Process Resilience

- Mask process failures by replication
- Organize process into groups, a message sent to a group is delivered to all members
- If a member fails, another should fill in

Flat Groups versus Hierarchical Groups

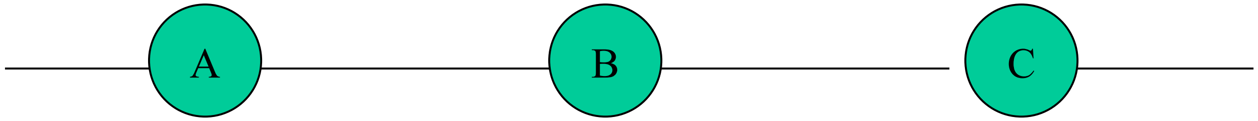


a) Communication in a flat group.
b) Communication in a simple hierarchical group

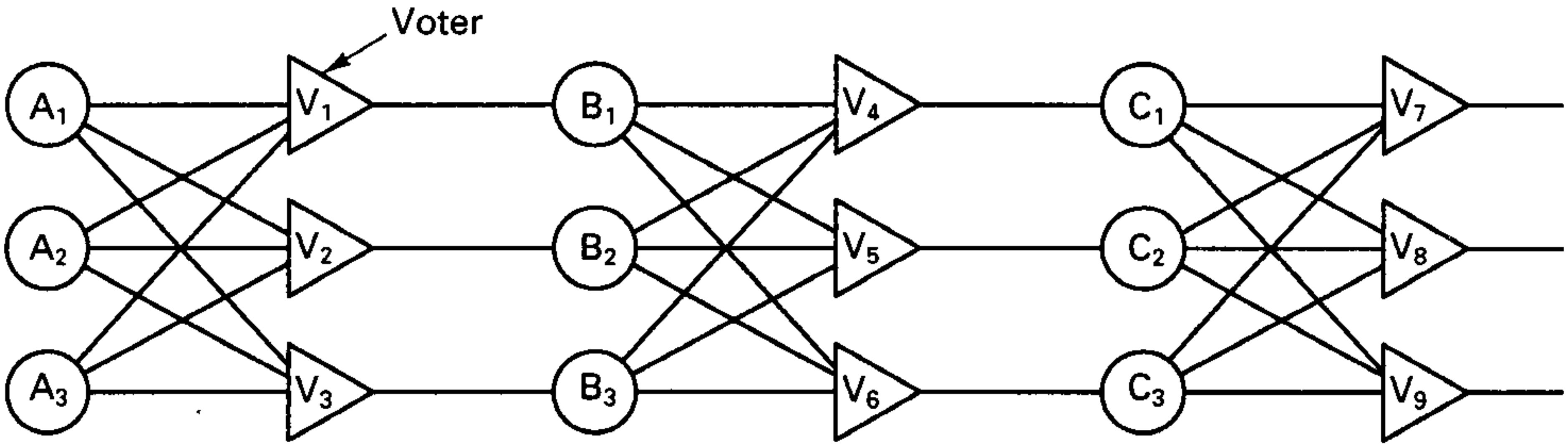
Fault Tolerance using active replication

- E.g. Aircraft (747 have four engines but can fly with three engines)
- Sports (multiple referees in case one misses an event)
- Can also referred as **State Machine Approach**.

Fault Tolerance using active replication



If one device is faulty then the final result will probably be wrong.



Fault Tolerance using active replication

- Each device is replicate three times.
- Each stage in the circuit is a triplicated voter.
- Each voter is a circuit that has three inputs and one output.
- If two or three of the inputs are same, the output is equal to that input.
- If all three inputs are different, the output is undefined.
- This kind of design is known as **TMR(Triple Modular Redundancy)**

- Suppose element A_2 fails. Each of the voters V_1, V_2 and V_3 gets two good input and one rogue input, and each of them outputs the correct value to the second stage.
- The effect of A_2 failing is completed masked, so that the inputs to B_1, B_2 and B_3 are exactly the same as they would have been had no fault occurred.
- If B_3 and C_1 are also faulty in addition to A_2, These effects also masked, so the three final outputs are still correct.

How much replication is needed?

- A system is said to be **k fault tolerant** if it can survive faults in *k* components and still meet its specifications.
- *K+1* processors can fault tolerant *k* fail-stop faults. If *k* of them fail, the one left can work. But need *2k+1* to tolerate *k* Byzantine faults because if *k* processors send out wrong replies, but there are still *k+1* processors giving the correct answer. By majority vote, a correct answer can still be obtained.

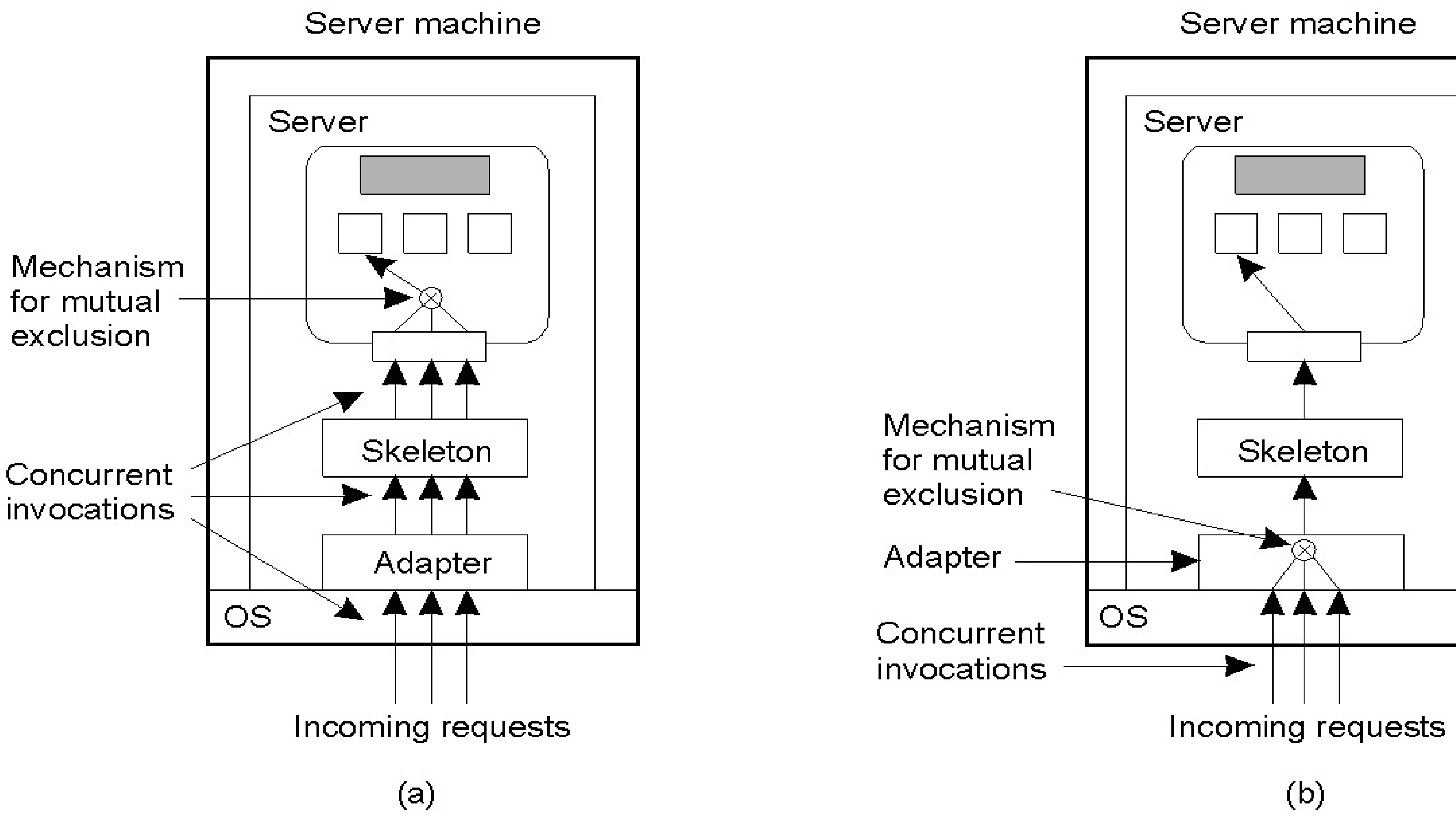
Reasons for Replication

- Data replication is a common technique in distributed systems. There are two reasons for data replication:
  - It increases the **reliability** of a system.
    - If one replica is unavailable or crashes, use another
    - Protect against corrupted data
  - It improves the **performance** of a system.
    - Scale with size of the distributed system (replicated Web servers)
    - Scale in geographically distributed systems (Web proxies)
- The key issue is the need to maintain **consistency** of replicated data.
  - If one copy is modified, others become inconsistent.

Object Replication

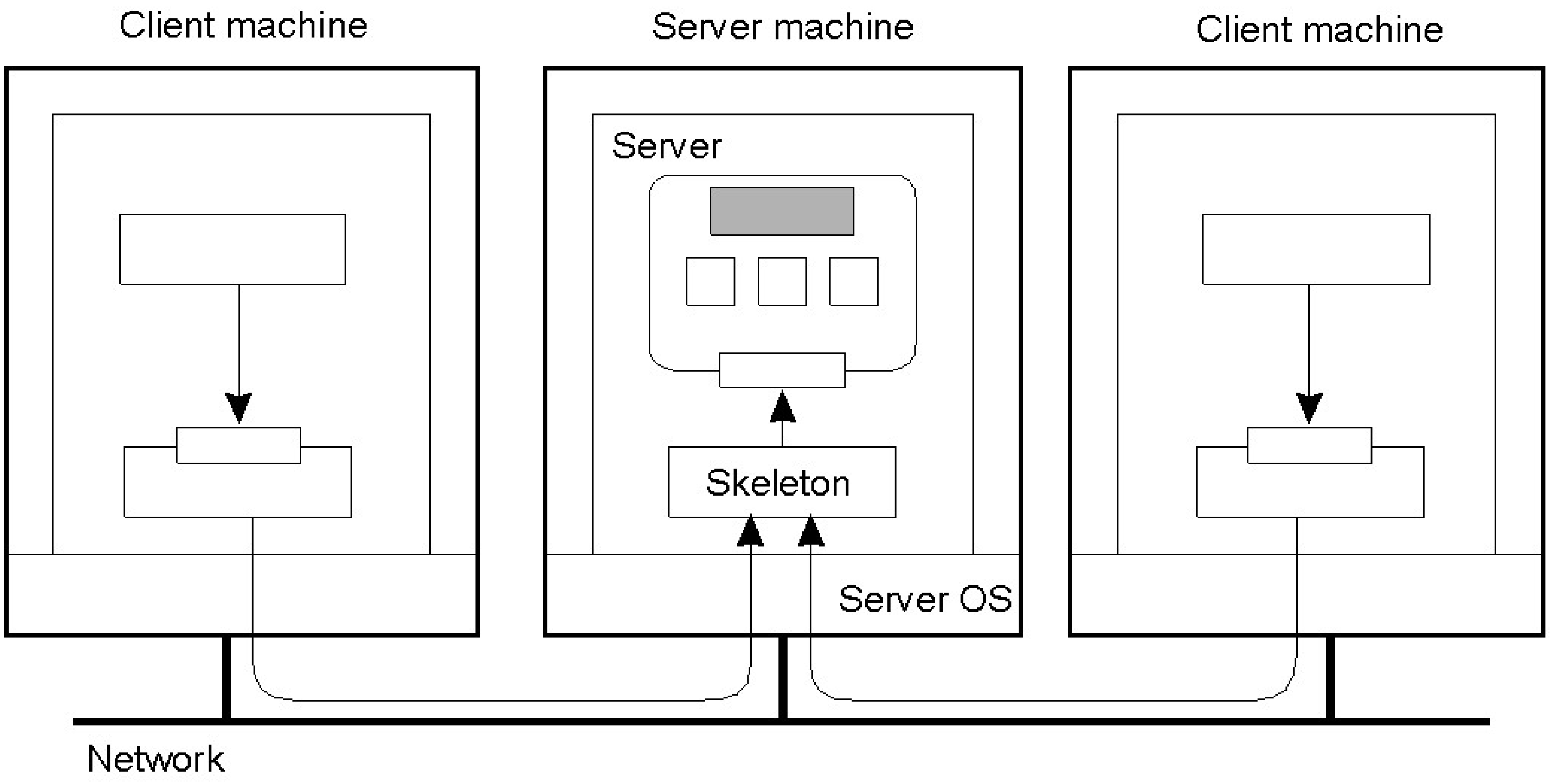
- There are two approaches for object sharing:
  - The object itself can handle concurrent invocation.
    - A Java object can be constructed as a monitor by declaring the object's methods to be synchronized.
  - The object is completely unprotected against concurrent invocations, but the server in which the object resides is made responsible for concurrency control.
    - In particular, use an appropriate object adapter.

Object Replication



a) A remote object capable of handling concurrent invocations on its own.
b) A remote object for which an object adapter is required to handle concurrent invocations

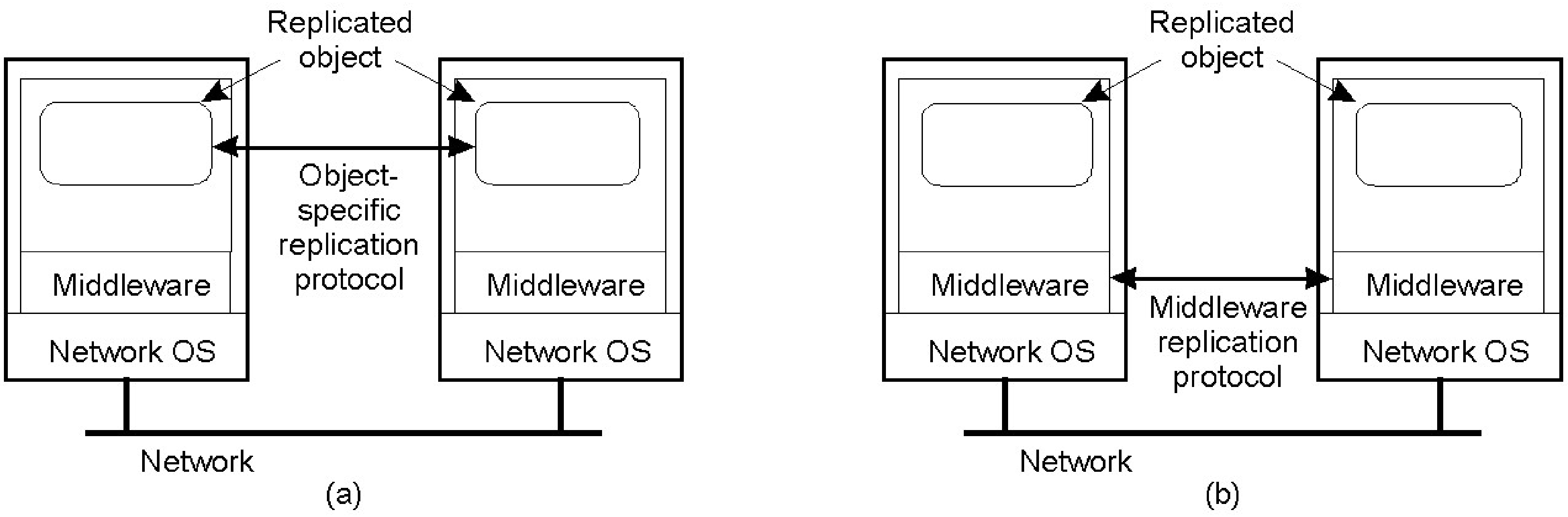
Object Replication



- Organization of a distributed remote object shared by two different clients.

Object Replication

- There are two approaches for object replication:
  - The application is responsible for replication (Fig (a)).
    - Application needs to handle consistency issues.
  - The system (middleware) handles replication (Fig(b)).
    - Consistency issues are handled by the middleware.
    - It simplifies application development but makes object-specific solutions harder.



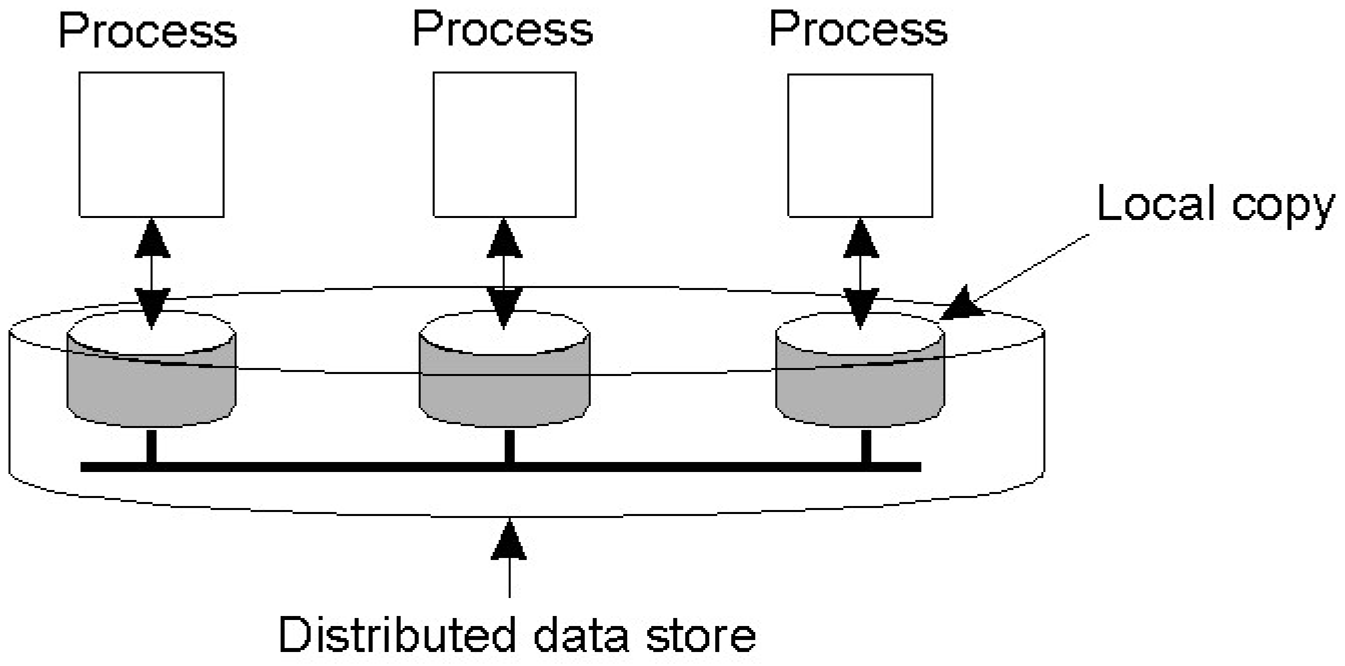
Replication and Scaling

- Replication and caching are used for system scalability.
- Multiple copies improve performance by reducing access latency but have higher network overheads of maintaining consistency.
  - Example: An object is replicated N times.
    - Consider the Read frequency R and the write frequency W
    - If $R << W$, high consistency overhead and wasted messages
    - Consistency maintenance is itself an issue
      - What semantics to provide?
      - Tight consistency requires globally synchronized clocks.

Replication and Scaling

- The solution is to loosen consistency requirements.
  - Variety of consistency semantics possible
- Consistency model (consistency semantics)
  - Contract between processes and the data store
    - If processes obey certain rules, data store will work correctly.
  - All models attempt to return the results of the last write for a read operation.
    - Differ in how last write is determined/defined

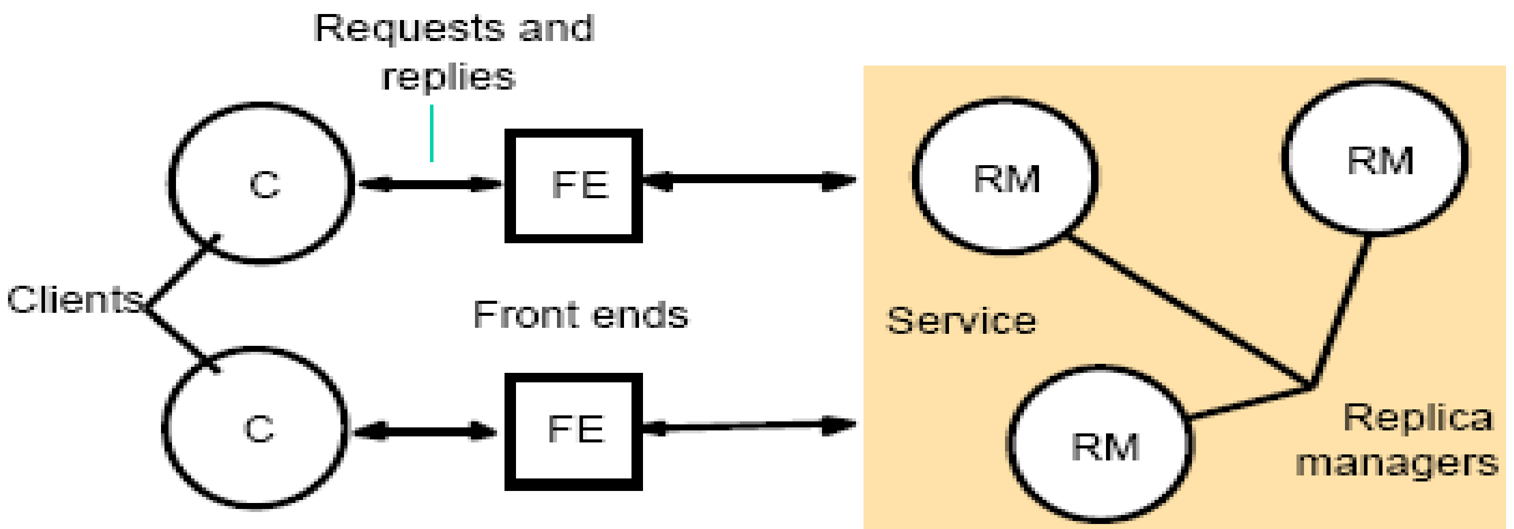
Data-Centric Consistency Models



- The general organization of a logical **data store**, physically distributed and replicated across multiple processes.

System Model

- Assume
  - Replica Manager applies operations to its replica
  - set of replica managers may be fixed or could change dynamically
  - requests are reads or writes (updates)



A basic architectural model for the management of replicated data

System Model-Replica manager

Five phases in performing a request

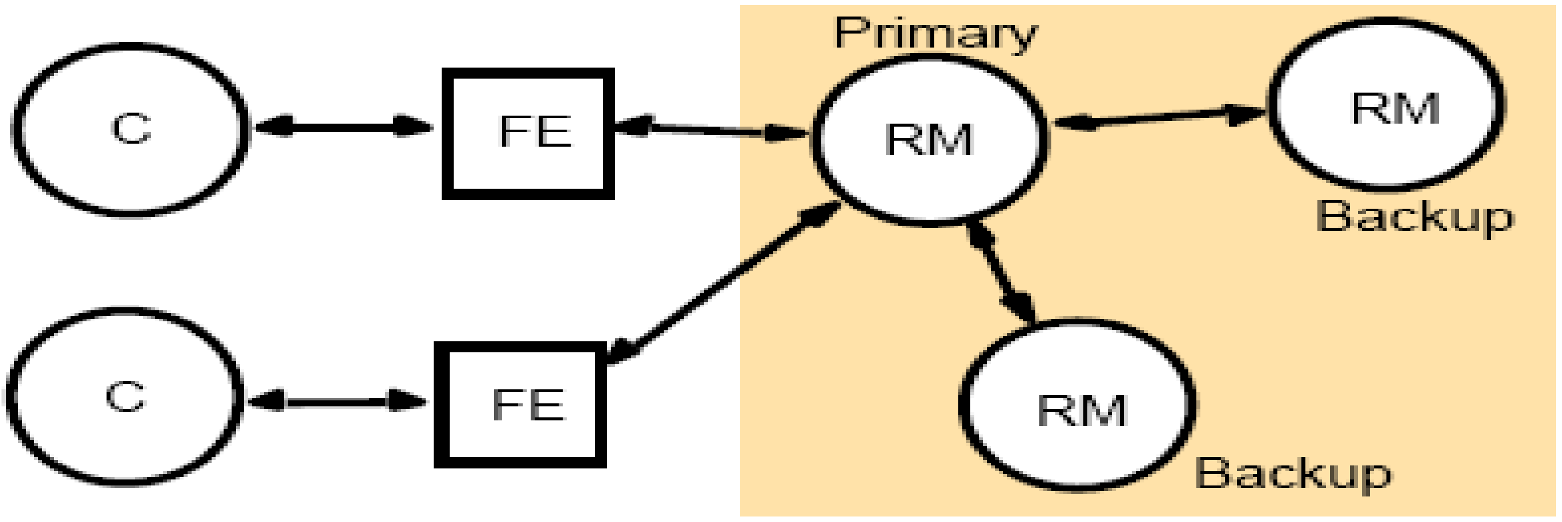
- Front end issues the request
  - Either sends to a single (primary) replica or multicasts to all replica managers
- Coordination
  - RMs coordinate in preparation for the execution of the request, i.e., agree if request is to be performed and the ordering of the request relative to others
  - FIFO ordering, Causal ordering, Total (sequential) ordering
- Execution
  - May be tentative (not fully agreed upon).
- Agreement
  - Reach consensus on effect of the request, e.g., agree to commit or abort in a transactional system
- Response: One or more RMs pass a response back to the front end.

Replication technique for fault tolerance (service provision technique)

1. Passive Replication
2. Active Replication

- Both the technique provide the fault tolerance. In passive replication there is only one server (called primary) that processes client requests. In active replication each client request is processed by all the replicas

The Passive (Primary-Backup) Model



Front ends only communicate with primary

•PRM(Primary Replica Manager) informs to all BRM(Backup Replica Manager) and provides the updated data BRM.

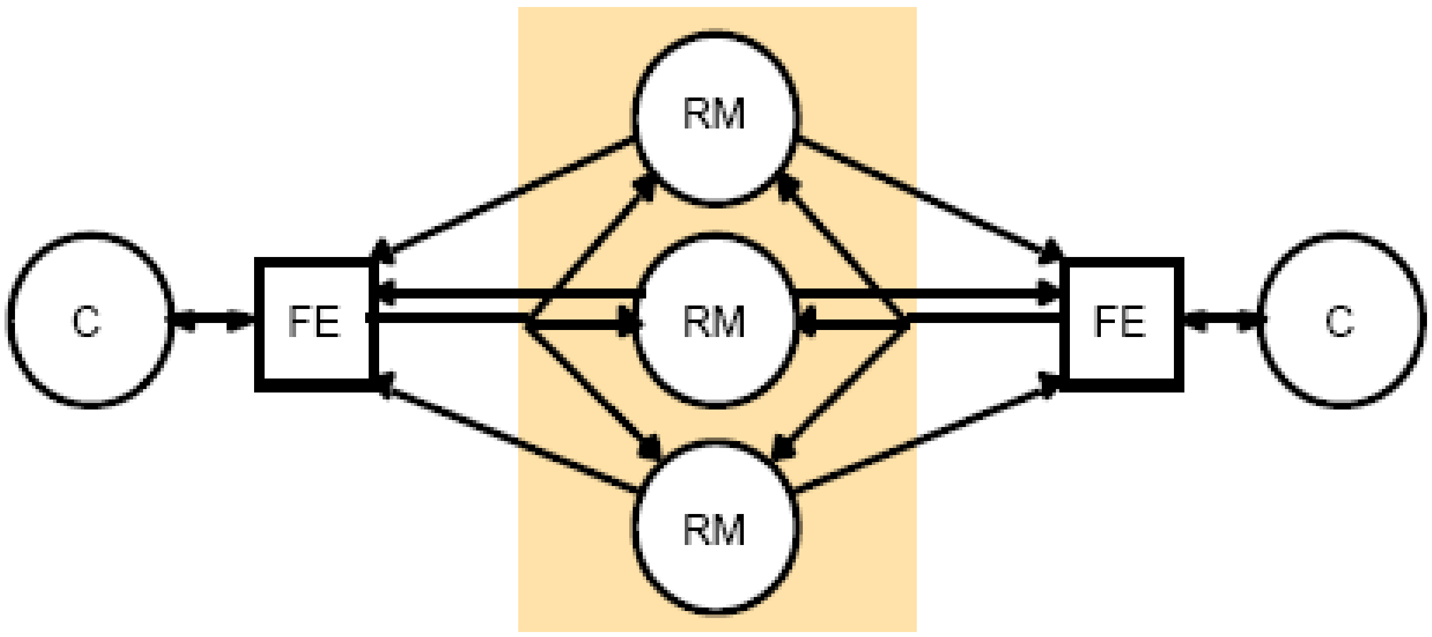
•Only Primary Replica manager is directly connected to front end

Passive (Primary-Backup) Replication

- Request: Front end issue client request with unique request ID to PRM(primary replica manager).
- Coordination: primary RM takes updates in FIFO. It is just preparation for execution. It maintains the order of request for execution. It may maintains the request in FIFO, causal and total ordering etc.
- Execution: primary RM executes the update and stores the response
- Agreement: primary sends the update to all backups. The backups send acknowledgements
- Response: primary responds to the FE, which hands the response back to the client
- If primary fails,
  - It should be replaced with a unique backup
  - RMs that survive must agree upon which operations had been performed when the replacement primary takes over.

Active Replication Using Multicast

- Active replication
  - FE multicasts request to each replica



Active Replication Using Multicast

- Sequence of Events:
- Request: FE with unique identifier of client request multicast request to all RM in a totally order fashion.
- Co-ordination: The Group command system deliver request to every RM in same(total) order.
- Execution: Every RM execute request which are delivered in same (total-> if a process issue m then m' then m->m') order, correct RM process the request identically. Since the request is send in same order.
- Agreement: This phase not required, every RM process request separately.
- Response: Each RM send response to FE the no of replicas that the front end collects depends up on failure assumption and on the multicast algorithm. If , for example, the goal is to tolerate only crash failure and the multicast satisfies uniform agreement and ordering properties, then the FE passes first response to client discarding others response.

  If request = n then ,

  Response is <= n.
- Drawbacks: Unnecessary response flow.

Fault tolerance using primary backup

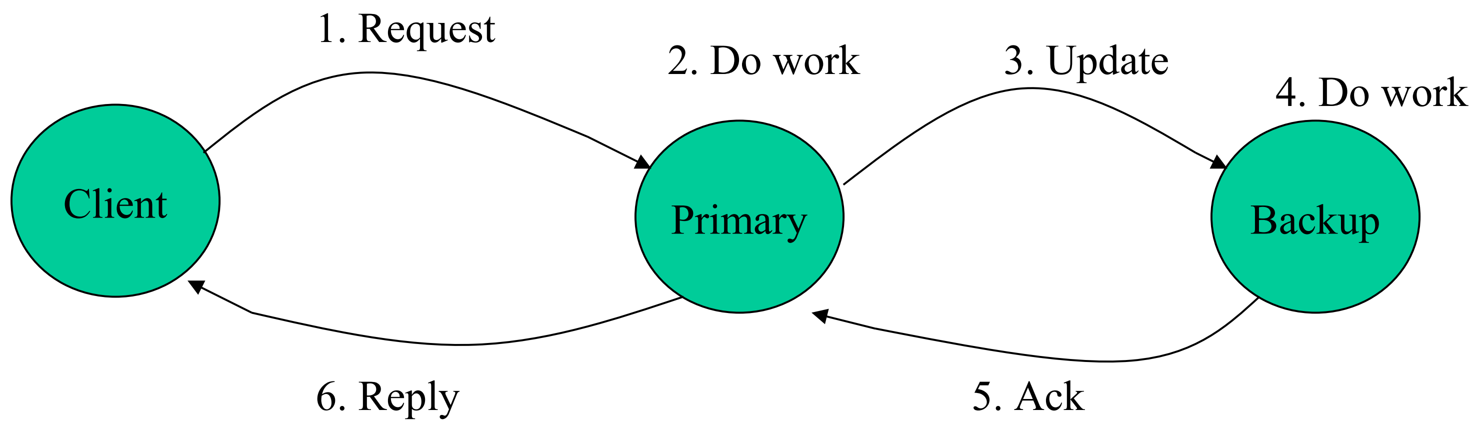
- With a primary backup approach, one server (the primary) does all the work.
- When the server fails, the backup takes over.
- A backup may periodically ping the primary with *"are you alive"* messages. If it fails to get an acknowledgement, then the backup may assume that the primary failed and it will take over the functions of the primary.
- If the system is asynchronous, there are no upper bounds on a timeout value for the pings. This is a problem.

Advantage: relatively easy to design since requests do not have to go multicast to a group of machines and there are no decisions to be made on who takes over.

Disadvantage: another backup is needed immediately.

- Backup servers work poorly with Byzantine faults, since the backup may not be able to detect that the primary has actually failed.

Fault Tolerance using primary backup



A simple primary backup protocol on write operation.

If the primary crashes

Before doing work (step 2):

•No harm is done

After doing the work but before sending the update :

•work will be done a second time by backup.

After step 4 before step 6:

•work may end up being done three times (once by primary, once by the backup as a result of step 3 and once after the backup becomes the primary)

The Failover

Recovery from a primary failure may be time-consuming and/or complex depending on the needs for continuous operation and application recovery.

- **Cold Failover:**
  - Cold failover entails application restart on the backup machine. When a backup machine takes over, it starts all the applications that were previously running on the primary system. Of course, any work that the primary may have done is now lost.
- **Warm failover**:
  - applications periodically write checkpoint files onto stable storage that is shared with the backup system. When the backup system takes over, it reads the checkpoint files to bring the applications to the state of the last checkpoint.
- **Hot failover**:
  - applications on the backup run in lockstep synchrony with applications on the primary, taking the same inputs as on the primary. When the backup takes over, it is in the exact state that the primary was in when it failed.

Handling of Processor Faults

- **Backward recovery** – checkpoints.
- In the check pointing method, two undesirable situations can occur:
- ***Lost message***
  - The state of process P_i indicates that it has sent a message *m* to process P_j. P_j has no record of receiving this message.
- ***Orphan message***
  - The state of process P_j is such that it has received a message *m* from the process P_i but the state of the process P_i is such that it has never sent the message *m* to P_j.

Checkpoints

- A **strongly consistent set** of checkpoints consist of a set of local checkpoints such that there is no orphan or lost message.
- A **consistent set** of checkpoints consists of a set of local checkpoints such that there is no orphan message.
- **Synchronous checkpointing:**
  - A processor P_i needs to take a checkpoint only if there is another process P_j that has taken a checkpoint that includes the receipt of a message from P_i and P_i has not recorded the sending of this message.
  - In this way no orphan message will be generated.
- **Asynchronous checkpointing**
  - Each process takes its checkpoints independently without any coordination.
- **Hybrid checkpointing**
  - Synchronous checkpoints are established in a longer period while asynchronous checkpoints are used in a shorter period. That is, within a synchronous period there are several asynchronous periods.

Recovery Mechanism

- Once a failure has occurred, it is essential that the process where the failure happened ***recovers*** to a correct state.
- Recovery from an error is ***fundamental*** to fault tolerance.
- Two main forms of recovery:

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.
    - Examples
        - Reliable communication through packet retransmission
2. **Forward Recovery**: bring the system into a correct state, from which it can then continue to execute.
    - Examples:
        - Reliable communication via erasure correction, such as an (n, k) block erasure code

Forward and Backward Recovery

- **Major disadvantage of Backward Recovery**:
  - Checkpointing can be very expensive (especially when errors are very rare).
  - [Despite the cost, backward recovery is implemented more often. The "logging" of information can be thought of as a type of checkpointing.].
- **Major disadvantage of Forward Recovery**:
  - In order to work, all potential errors need to be accounted for *up-front*.
  - When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

Stable Storage

- In order to store checkpoints and logs, information needs to be stored safely - not just able to survive crashes, but also able to survive hardware faults
- RAID (of the mirroring or parity checking variety) is the typical example of stable storage

Recovery – Stable Storage

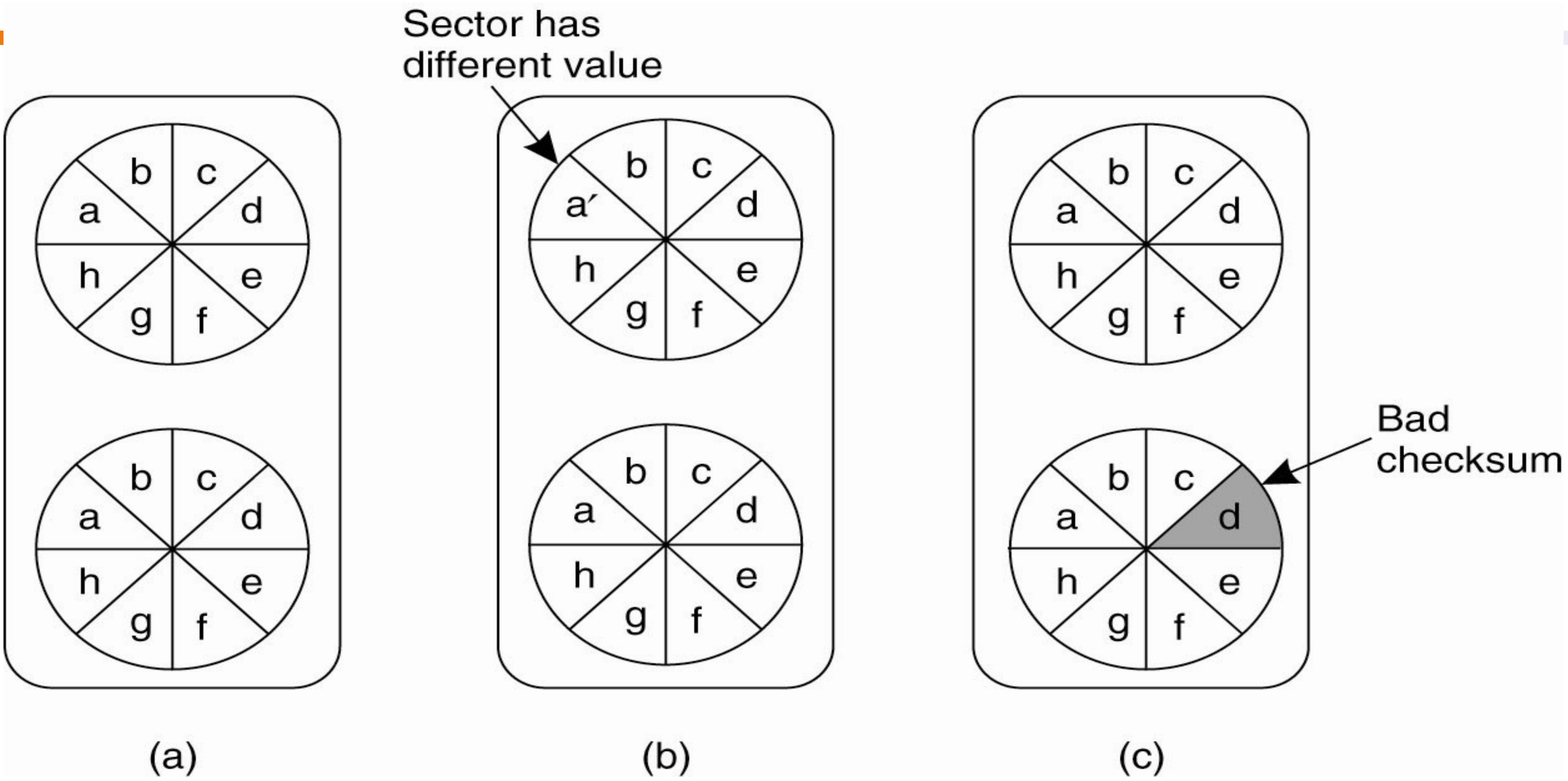


Figure 8-23. (a) Stable storage.
(b) Crash after drive 1 is updated. (c) Bad spot.

Determining Global States

- **The Global State of a distributed computation is**
  - *the set of local states of all individual processes involved in the computation*

    +

  - *the states of the communication channels*

Obvious First Solution…

- Synchronize clocks of all processes and ask all processes to record their states at known time t

- Problems?
  - Time synchronization possible only approximately
    - distributed banking applications: no approximations!

  - Does not record the state of messages in the channels

Global State

- We cannot determine the exact global state of the system, but we can record a snapshot of it

- **Distributed Snapshot**: a state the system might have been in [Chandy and Lamport]

A naïve snapshot algorithm

- Processes record their state at *any* arbitrary point
- A designated process collects these states

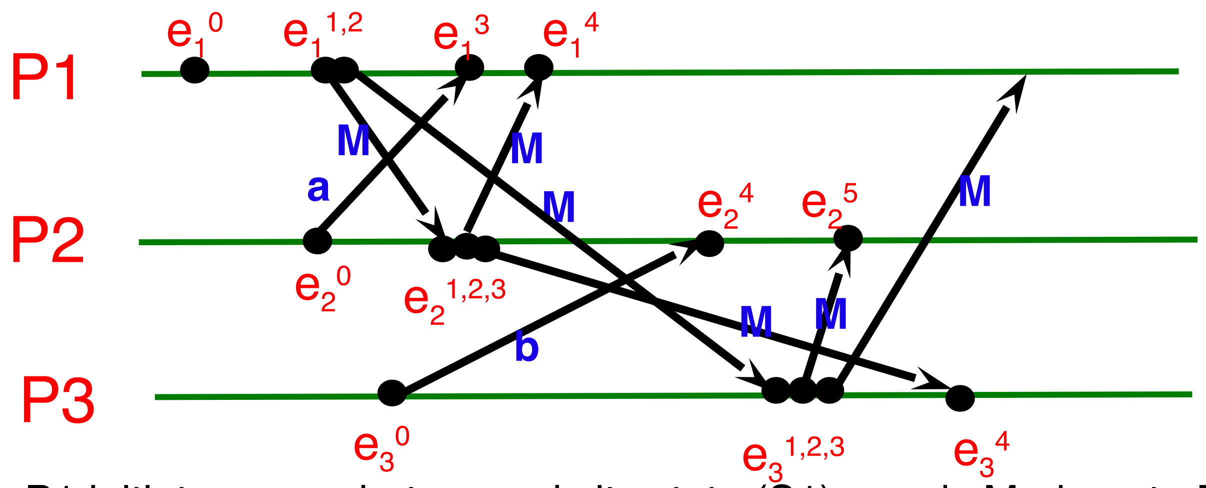
The "Snapshot" Algorithm

- **Records a set of process and channel states such that the combination is a consistent GS.**
- ***Assumptions:***
  - **No failure, all messages arrive intact, exactly once**
  - **Communication channels are unidirectional and FIFO-ordered**
  - **There is a comm. path between any two processes**
  - **Any process may initiate the snapshot (sends Marker)**
  - **Snapshot does not interfere with normal execution**
  - **Each process records its state and the state of its incoming channels**

The "Snapshot" Algorithm (2)

- 1. **Marker sending rule for initiator process P_0**
  - After P_0 has recorded its state
    - for each outgoing channel C, sends a marker on C
- 2. **Marker receiving rule for a process P_k, on receipt of a marker over channel C**
  - if P_k has not yet recorded its state
    - records P_k's state
    - records the state of C as "empty"
    - turns on recording of messages over other incoming channels
    - for each outgoing channel C, sends a marker on C
  - else
    - records the state of C as all the messages received over C since P_k saved its state

Snapshot Example



1- P1 initiates snapshot: records its state (S1); sends Markers to P2 & P3; turns on recording for channels C21 and C31

2- P2 receives Marker over C12, records its state (S2), sets state(C12) = {} sends Marker to P1 & P3; turns on recording for channel C32

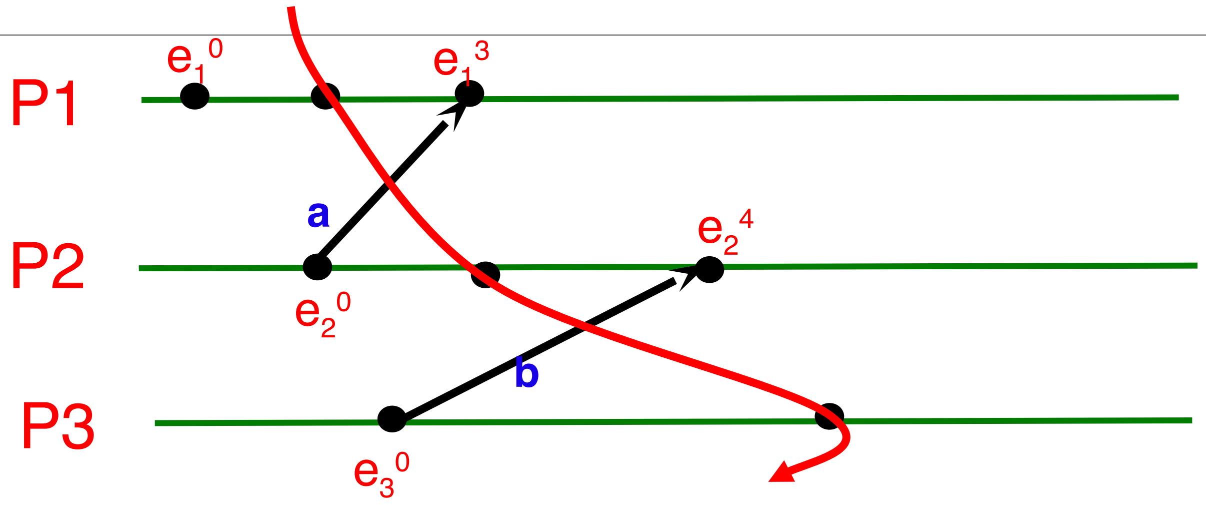
3- P1 receives Marker over C21, sets state(C21) = {a}

4- P3 receives Marker over C13, records its state (S3), sets state(C13) = {} sends Marker to P1 & P2; turns on recording for channel C23

5- P2 receives Marker over C32, sets state(C32) = {b}

6- P3 receives Marker over C23, sets state(C23) = {}

Snapshot Example

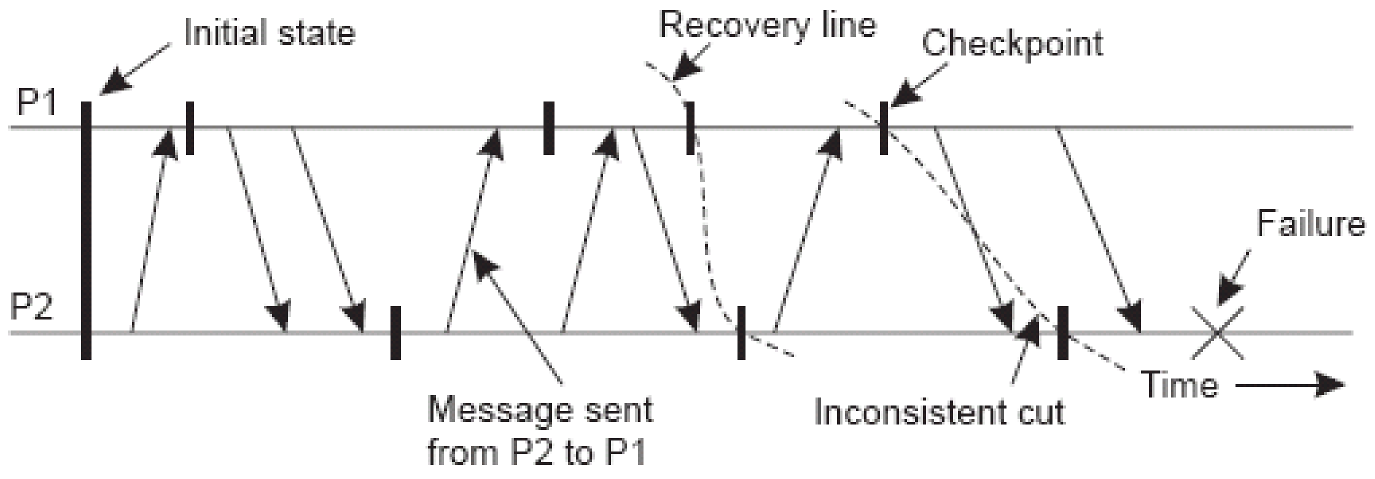


Distributed Snapshot Algorithm

- When a process finishes local snapshot, it collects its local state (S and C) and sends it to the initiator of the distributed snapshot
- The initiator can then analyze the state
- One algorithm for distributed global snapshots, but it's not particularly efficient for large systems

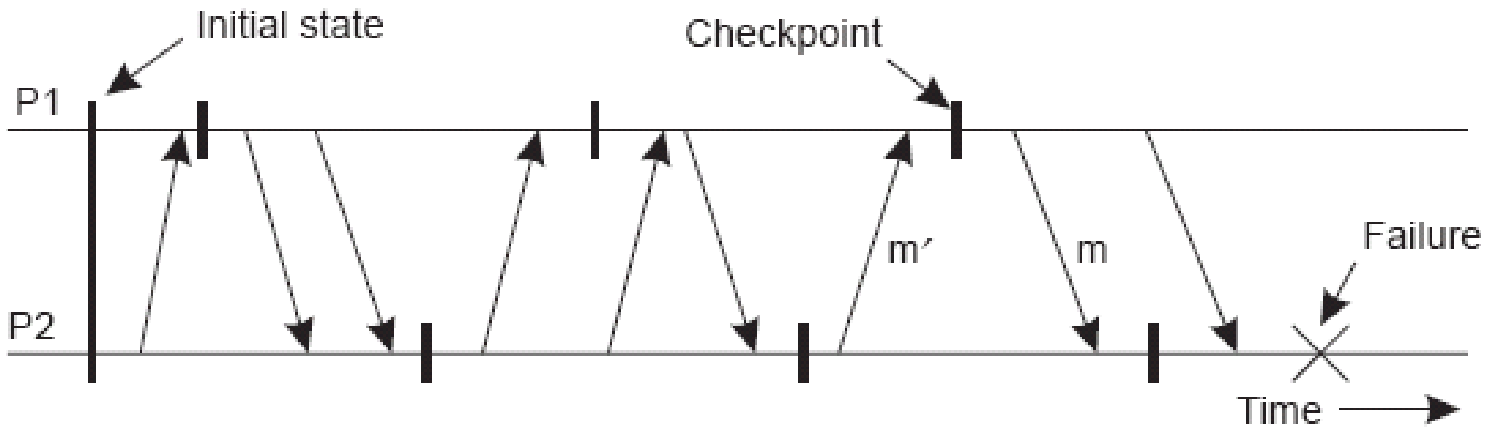
Checkpointing

- The most recent distributed snapshot in a system is also called the *recovery line*



Independent Checkpointing

- It is often difficult to find a recovery line in a system where every process just records its local state every so often - a *domino effect* or cascading rollback can result:



Coordinated Checkpointing

- To solve this problem, systems can implement *coordinated checkpointing*
- We’ve discussed one algorithm for distributed global snapshots, but it’s not particularly efficient for large systems
- Another way to do it is to use a two-phase blocking protocol (with some coordinator) to get every process to checkpoint its local state “simultaneously”

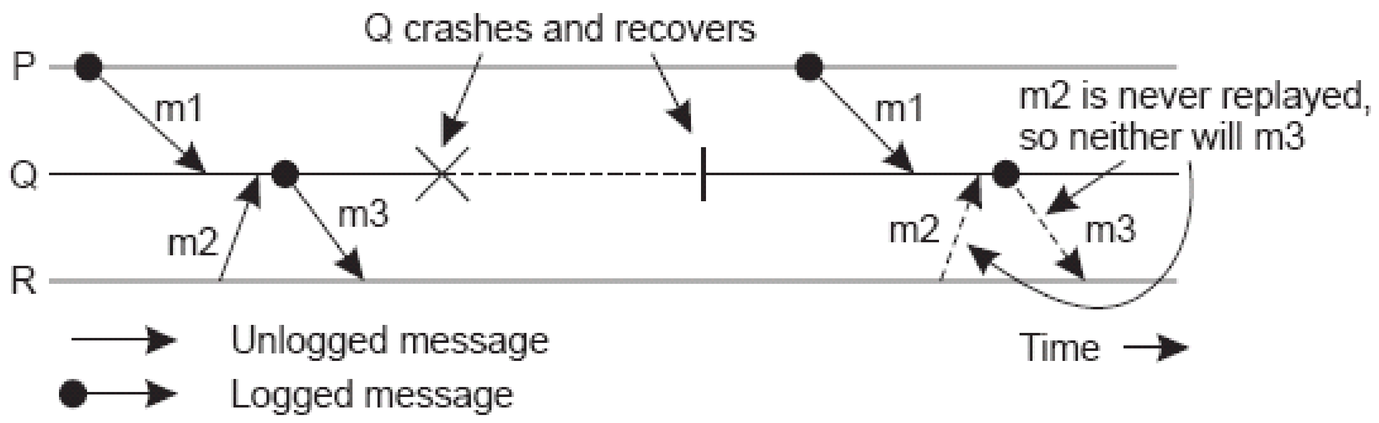
Coordinated Checkpointing

- Make sure that processes are synchronized when doing the checkpoint
- Two-phase blocking protocol
  1. Coordinator multicasts *CHECKPOINT_REQUEST*
  2. Processes take local checkpoint
     - Delay further sends
     - Acknowledge to coordinator
     - Send state
  3. Coordinator multicasts CHECKPOINT_DONE

Message Logging

- Check pointing is expensive - message logging allows the occurrences between checkpoints to be *replayed*, so that checkpoints don't need to happen as frequently
- We need to choose when to log messages
- Message-logging schemes can be characterized as *pessimistic* or *optimistic* by how they deal with *orphan processes*
  - An orphan process is one that survives the crash of another process but has an inconsistent state after the other process recovers

Message Logging



- An example of an incorrect replay of messages
- We assume that each message *m* has a header containing all the information necessary to retransmit *m* (sender, receiver, sequence no., etc.)

Agreement

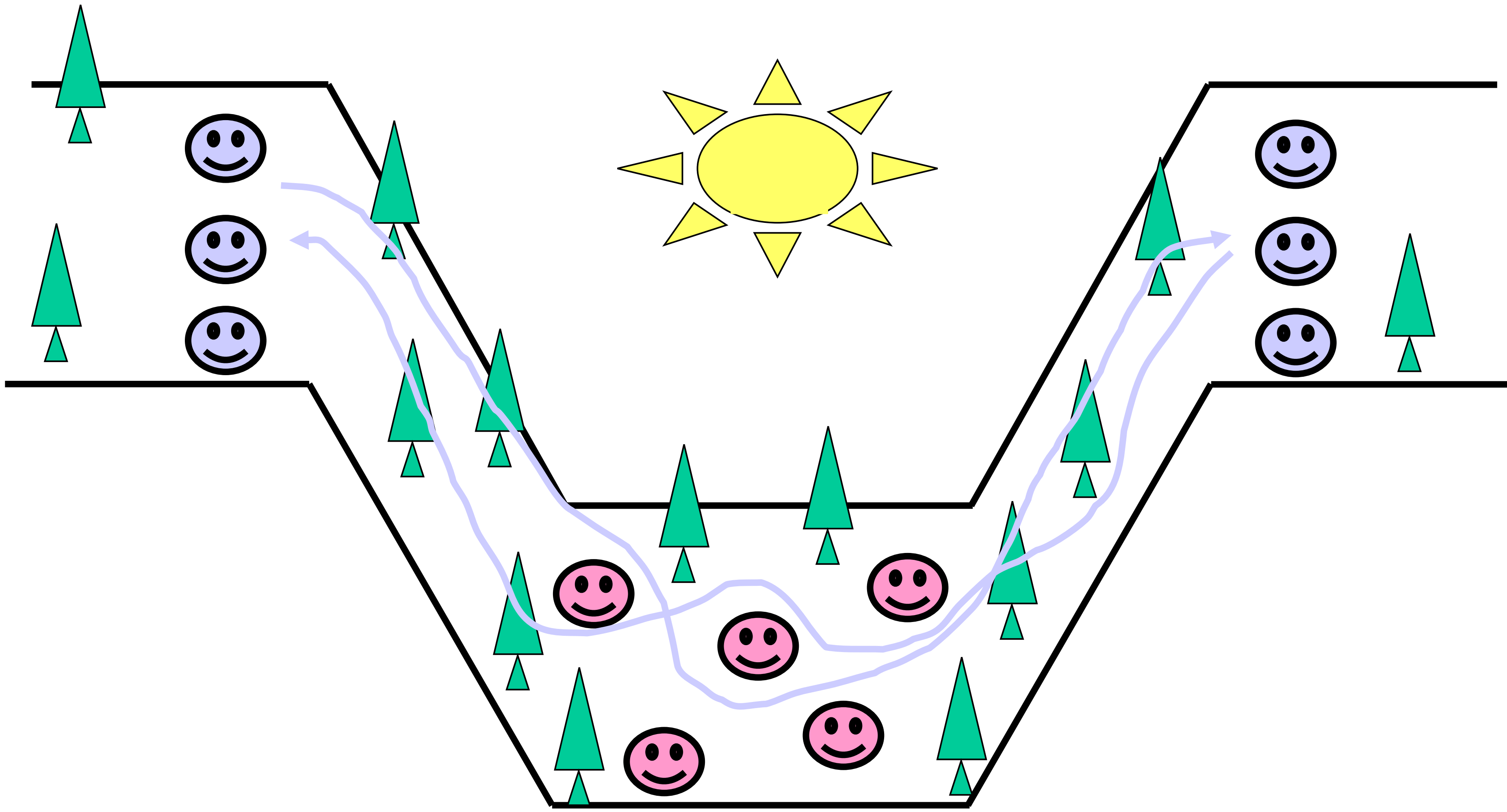
- Need agreement in DS:
  - Leader, commit, synchronize
- **Distributed Agreement algorithm**: all non-faulty processes achieve consensus in a finite number of steps
  - Perfect processes, faulty channels: **two-army**
  - Faulty processes, perfect channels: **Byzantine generals**

Agreement in Faulty Systems

Two-army problem

- Two blue armies must reach agreement to attack a red army. If one blue army attacks by itself it will be slaughtered. They can only communicate using an unreliable channel: sending a messenger who is subject to capture by the red army.
- They can never reach an agreement on attacking.
- Now assume the communication is perfect but the processors are not. The classical problem is called the **Byzantine generals problem.** **N** generals and **M** of them are traitors. Can they reach an agreement?

Two-Army Problem



Two-Army Problem

- In this example, Enemy Red Army has 5000 troops. Blue Army has two separate gatherings, Blue (1) and Blue (2), each of 3000 troops. Alone Blue will loose, together as a coordinated attack Blue can win. Communications is by unreliable channel (send a messenger who may be captured by red army so may not arrive.

Impossible Consensus

- Agreement is **<u>impossible</u>** in **asynchronous** DS, even if only one process fails [Fischer et al.]

- Asynchronous DS:
  - messages cannot be guaranteed to be delivered within a known, finite time
  - cannot distinguish a slow process from a crashed one

Possible Consensus

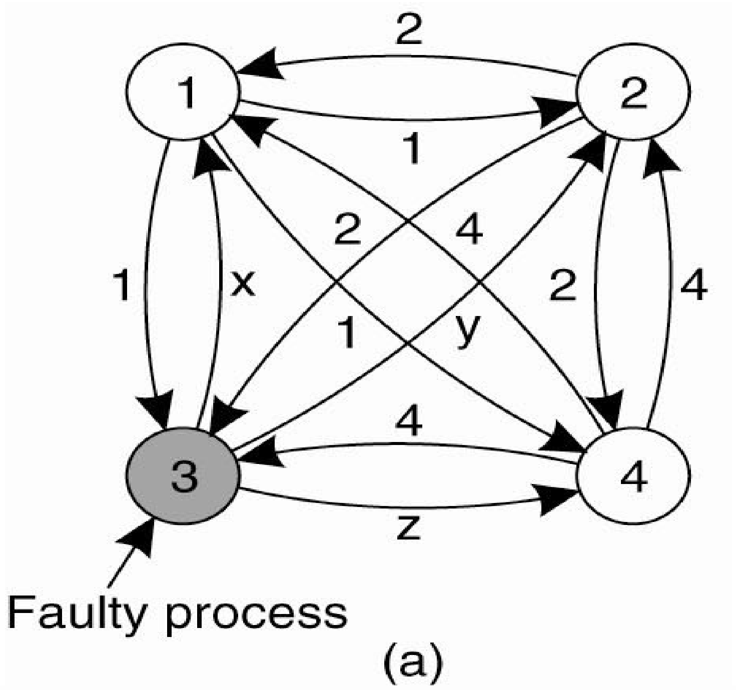
- Agreement is <u>**possible**</u> in **synchronous** DS [e.g., Lamport et al.]
  - Messages can be guaranteed to be delivered within a known, finite time.
  - Byzantine Generals Problem

- A synchronous DS: can distinguish a slow process from a crashed one

Lamport's Algorithm

- **Step 1**: Every non faulty process *i* sends v_i to every other process using reliable unicasting. Faulty process can send anything.
- **Step 2**: The result of the announcements of step 1 are collected together in the form of vectors.
- **Step 3:** Every process passing its vector from to every other process. (process may gets n-1 vectors).
- **Step 4:** Each process examines the *ith* element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. Of no value has majority, the corresponding element of the result vector is marked UNKNOWN.

Lamport's algorithm



(a)

N = 4(processor)
M = 1 (traitor)

The Byzantine agreement problem for three non faulty and one faulty process. (a) Each process sends their value to the others.
(b) The vectors that each process assembles based on (a).
(c) The vectors that each process receives in step 3.

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

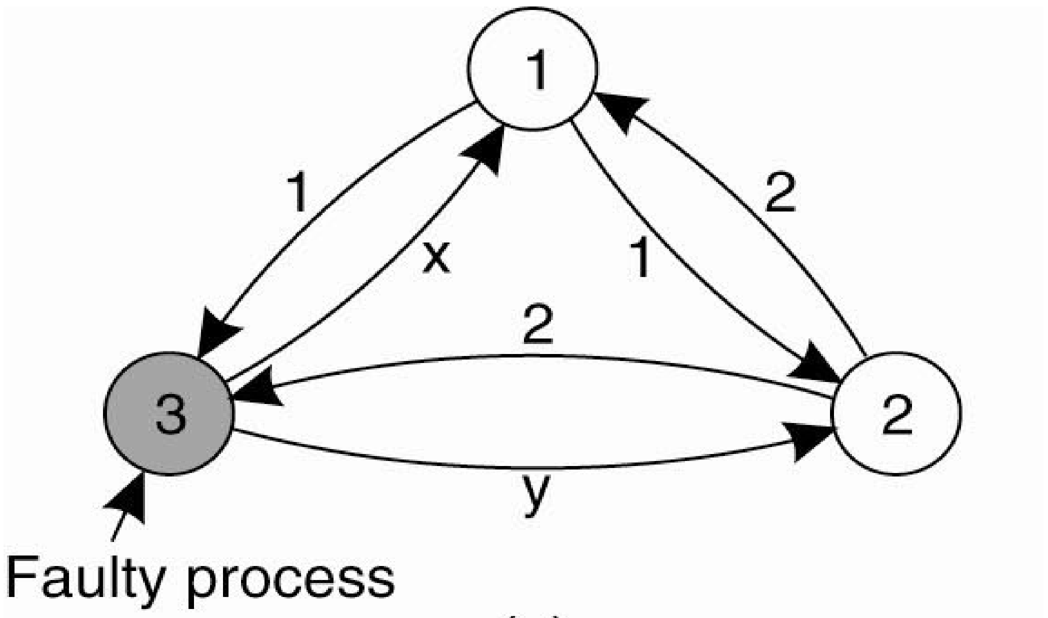
(b)

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)

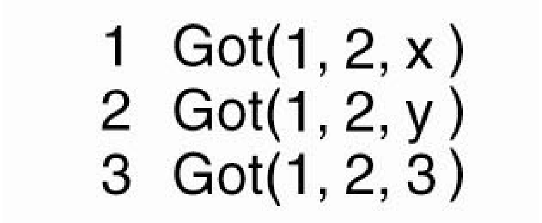
(c)

Result

- If there are m faulty processors, agreement can be achieved only if 2m+1 correct processors are present, for a total of 3m+1.
- Suppose n=3, m=1. Agreement cannot be reached.

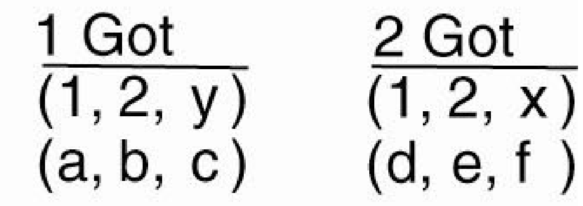


(a)



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)

With two correct process and one faulty process. No majority. Cannot reach an agreement.

Agreement under different models

- Turek and Shasha considered the following parameters for the agreement problem.

1. The system can be synchronous (A=1) or asynchronous (A=0).
2. Communication delay can be either bounded (B=1) or unbounded (B=0).
3. Messages can be either ordered (C=1) or unordered (C=0).
4. The transmission mechanism can be either point-to-point (D=0) or broadcast (D=1).

A Karnaugh map for the agreement problem

AB \ CD	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	1	1	1	1
10	0	0	1	1

Circumstances under which distributed agreement can be reached.

A Karnaugh map for the agreement problem

- Minimizing the Boolean function we have the following expression for the conditions under which consensus is possible:

  AB+AC+CD = True
- (AB=1): Processors are synchronous and communication delay is bounded.
- (AC=1): Processors are synchronous and messages are ordered.
- (CD=1): Messages are ordered and the transmission mechanism is broadcast.

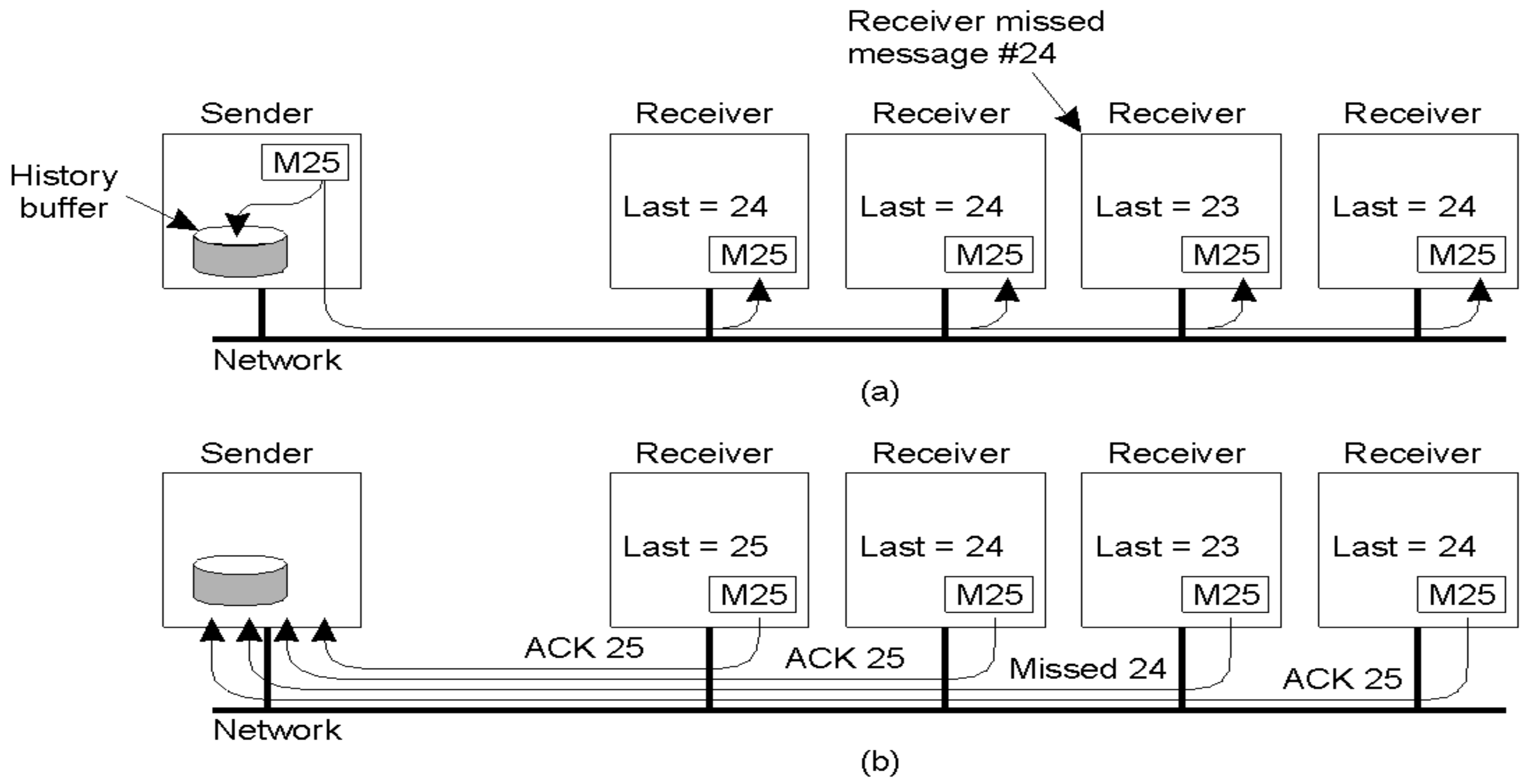
Reliable Group Communication

- When a group is static and processes do not fail
- Reliable communication = deliver the message to all group members
  - Any order delivery
  - Ordered delivery

Reliable-Multicasting

- If the number of processes is small, achieving reliability through multiple reliable point-to-point channels is a simple and often straight forward solution.
- A message that is sent to a process group should be delivered to each member of that group.
  - What happens if during communication a process joins the group?
  - Should that process also receive the message?
  - We should also determine what happens if a (sending) process crashes during communication.
- Multicasting is considered to be reliable when it can be guaranteed that all non faulty group member receive the message.
- If we assume that processes do not fail, and processes do not join or leave the group while communication is going on, reliable multicasting simply means that every message should be delivered to each current group member.
- In the simplest case, there is no requirement that all group members receive messages in the same order, but sometimes this feature is needed.

Basic Reliable-Multicasting Schemes



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

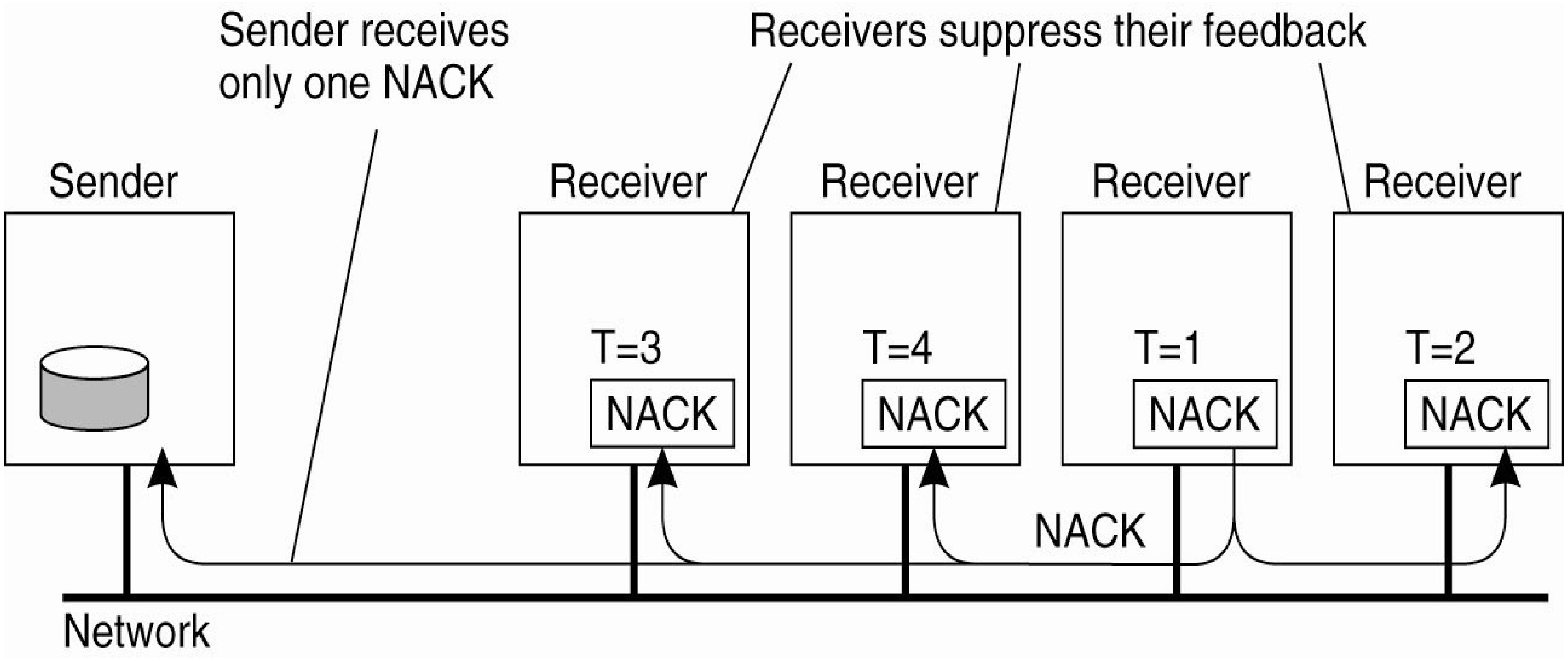
a) Message transmission

b) Reporting feedback

SRM: Scalable Reliable Multicasting

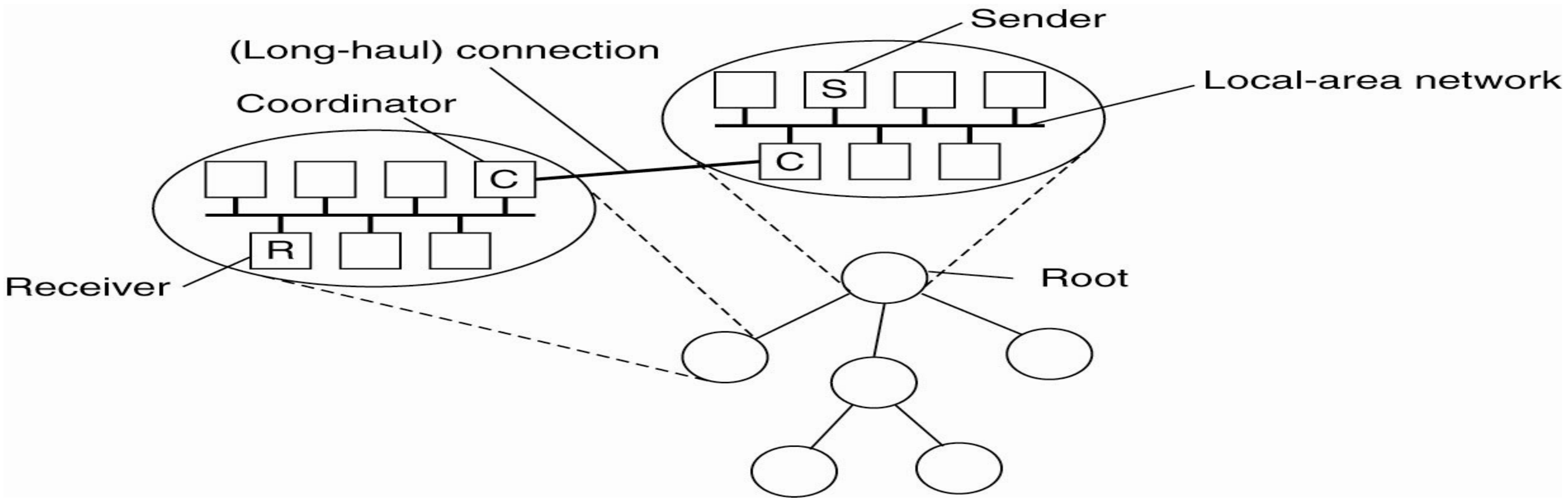
- In Reliable Multicast: If there are N receivers, the sender must be prepared to accept at least N acknowledgments (more load on network)
- In SRM: Receivers *never* acknowledge successful delivery.
- **Only missing messages are reported. (Problem:** Sender have to keep a message in its history buffer forever)
- NACKs are multicast to all group members.
- This allows other members to suppress their feedback, if necessary.
- To avoid "retransmission clashes", each member is required to wait a random delay prior to NACKing.

Nonhierarchical Feedback Control



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

Hierarchical Feedback Control



- The essence of hierarchical reliable multicasting.
- Each local coordinator forwards the message to its children and later handles retransmission requests.
- Each subgroup appoints a local coordinator, which is responsible for handling retransmission requests of receivers contained in its subgroup.

Hierarchical Feedback Control(2)

- The local coordinator will thus have its own history buffer.
- If the coordinator itself has missed a message m, it asks the coordinator of the parent subgroup to retransmit m.
- In a scheme based on acknowledgments, a local coordinator sends an acknowledgment to its parent if it has received the message.
- If a coordinator has received acknowledgments for message m from all members in its subgroup, as well as from its children, it can remove m from its history buffer.
- The main problem with hierarchical solutions is the construction of the tree.

Atomic Multicast

- All messages are delivered in the same order to “all” processes
- **Group view**: the view on the set of processes contained in the group
- **Virtual synchronous multicast**: a message m multicast to a group view G is delivered to all non-faulty processes in G
  - If sender fails “before” m reaches a non-faulty process, none of the processes deliver m.
- If the sender of the message crashes during the multicast, the message may either be delivered to all remaining processes, or ignored by each of them.
- A reliable multicast with this property is said to be virtually synchronous

Virtual Synchrony System Model

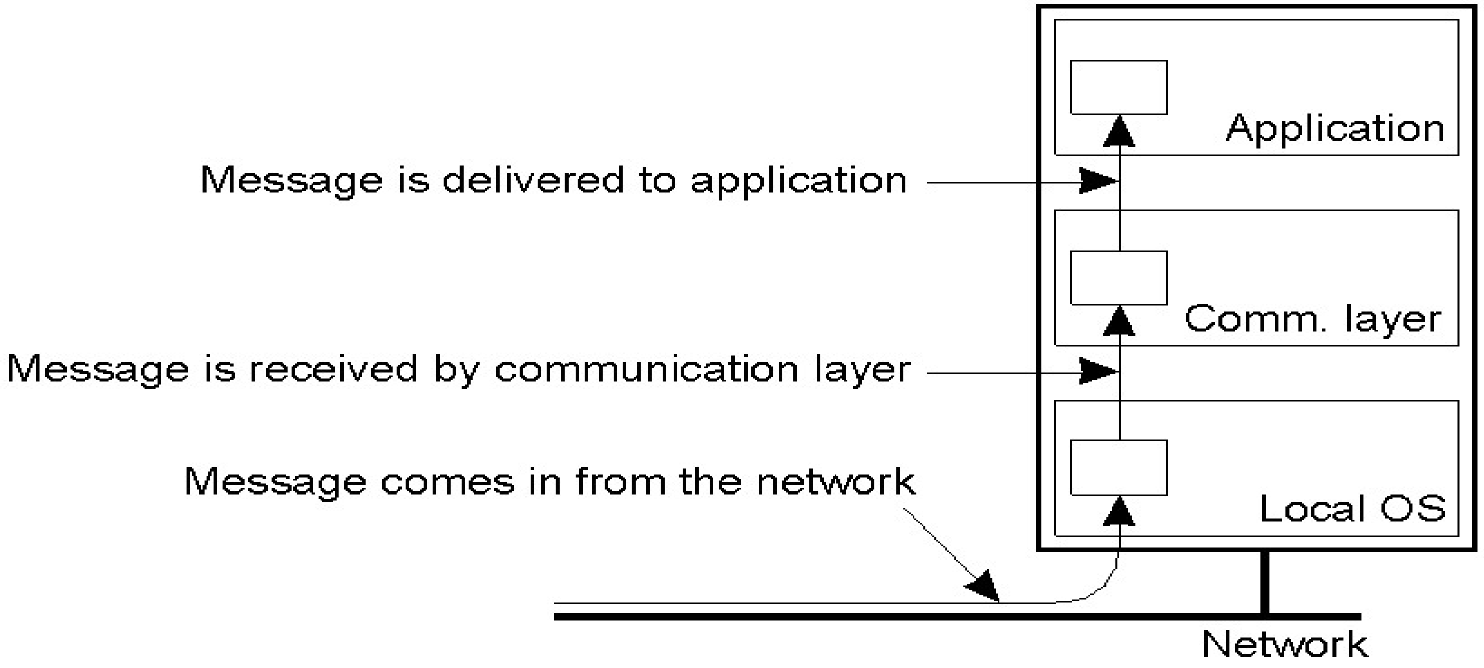

Application
Message is delivered to application
Comm. layer
Message is received by communication layer
Message comes in from the network
Local OS
Network

Fig. The logical organization of a distributed system to distinguish between message receipt and message delivery

- A received message is locally buffered in the communication layer until it can be delivered to the application that is logically placed at a higher layer.

Virtual Synchrony (2)

- Assume that while the multicast is taking place, another process joins or leaves the group. (need announcement of change in group membership to all processes in G).
- A view change takes place by multicasting a message ***vc*** announcing the joining or leaving of a process.
- We now have two multicast messages simultaneously in transit: ***m*** and ***vc***.
- Need to guarantee is that ***m*** is either delivered to all processes in G before each one of them is delivered message ***vc***, or ***m*** is not delivered at all.

Virtual Synchrony (3)

- Question: if ***m*** is not delivered to any process, how can we speak of a reliable multicast protocol?
  - In principle. there is only one case in which delivery of ***m*** is allowed to fail:
    - when the group membership change is the result of the sender of ***m*** crashing.
    - In that case, either all members of G should hear the abort of the new member, or none.
    - Alternatively, ***m*** may be ignored by each member, which corresponds to the situation that the sender crashed before ***m*** was sent.

Virtual Synchrony (3)

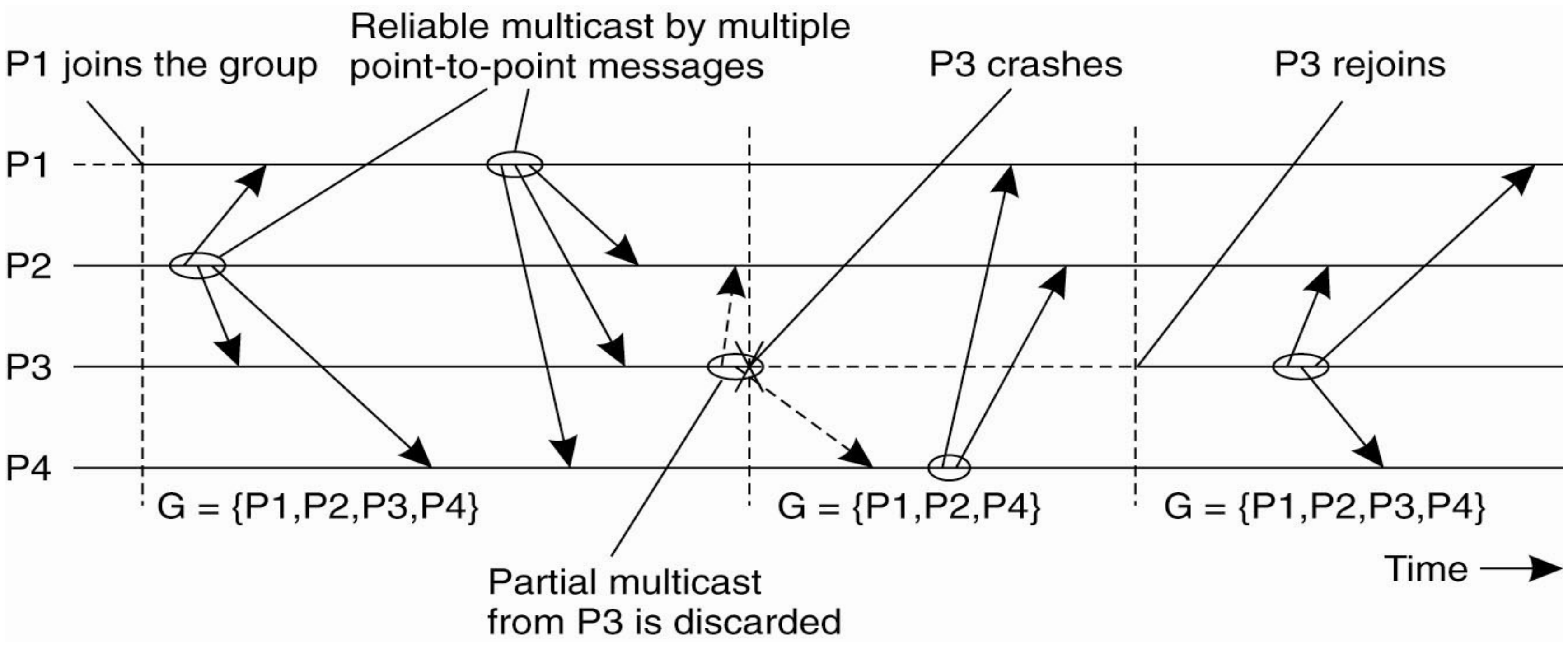


Fig: The principle of virtual synchronous multicast

- After some messages have been multicast, P3 crashes. However, before crashing. it succeeded in multicasting a message to process P2 and P4, but not to P1.
- However, virtual synchrony guarantees that the message is not delivered at all, effectively establishing the situation that the message was never sent before P3 crashed.

Reliability of Group Communication?

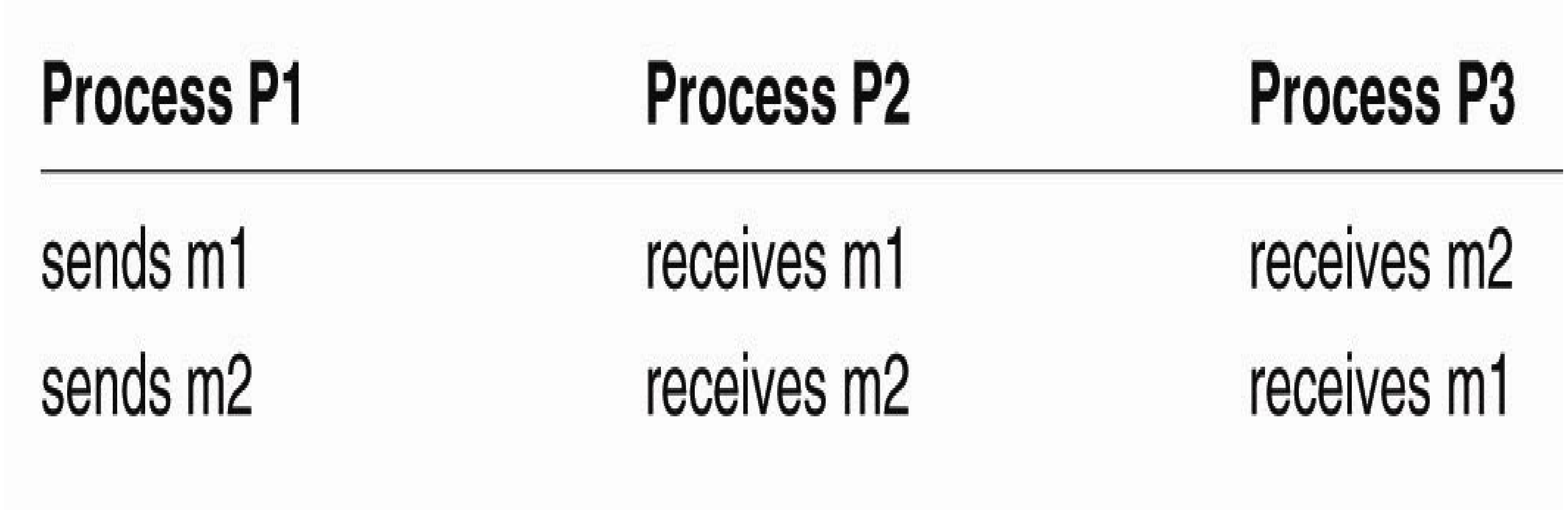
- A sent message is received by all members
  (acks from all => ok)
- Problem: during a multicast operation
  - an old member disappears from the group
  - a new member joins the group
- Solution
  - membership changes synchronize multicasting
    - ⇒ during a MC operation no membership changes
  - Virtual synchrony: “all” processes see message and membership change in the same order

Message Ordering (1)

- Four different orderings are distinguished:

1. Unordered multicasts
2. FIFO-ordered multicasts
3. Causally-ordered multicasts
4. Totally-ordered multicasts

Unordered

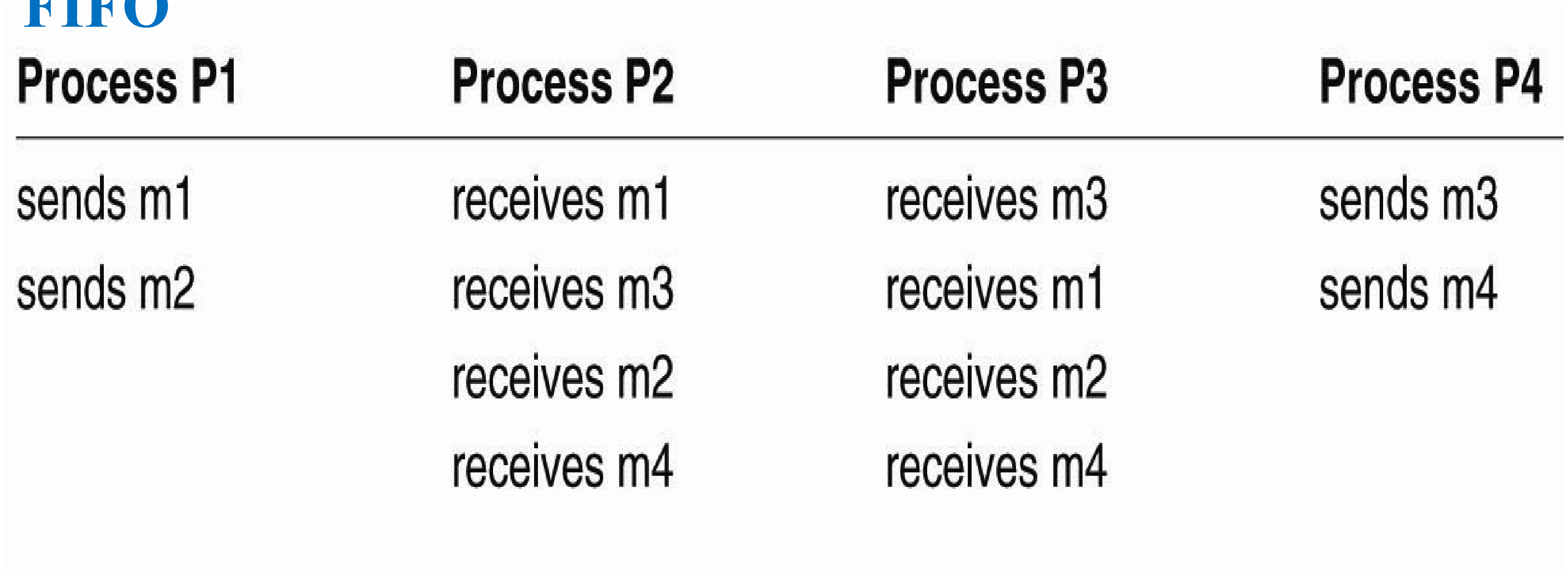


Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

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

Message Ordering (2)

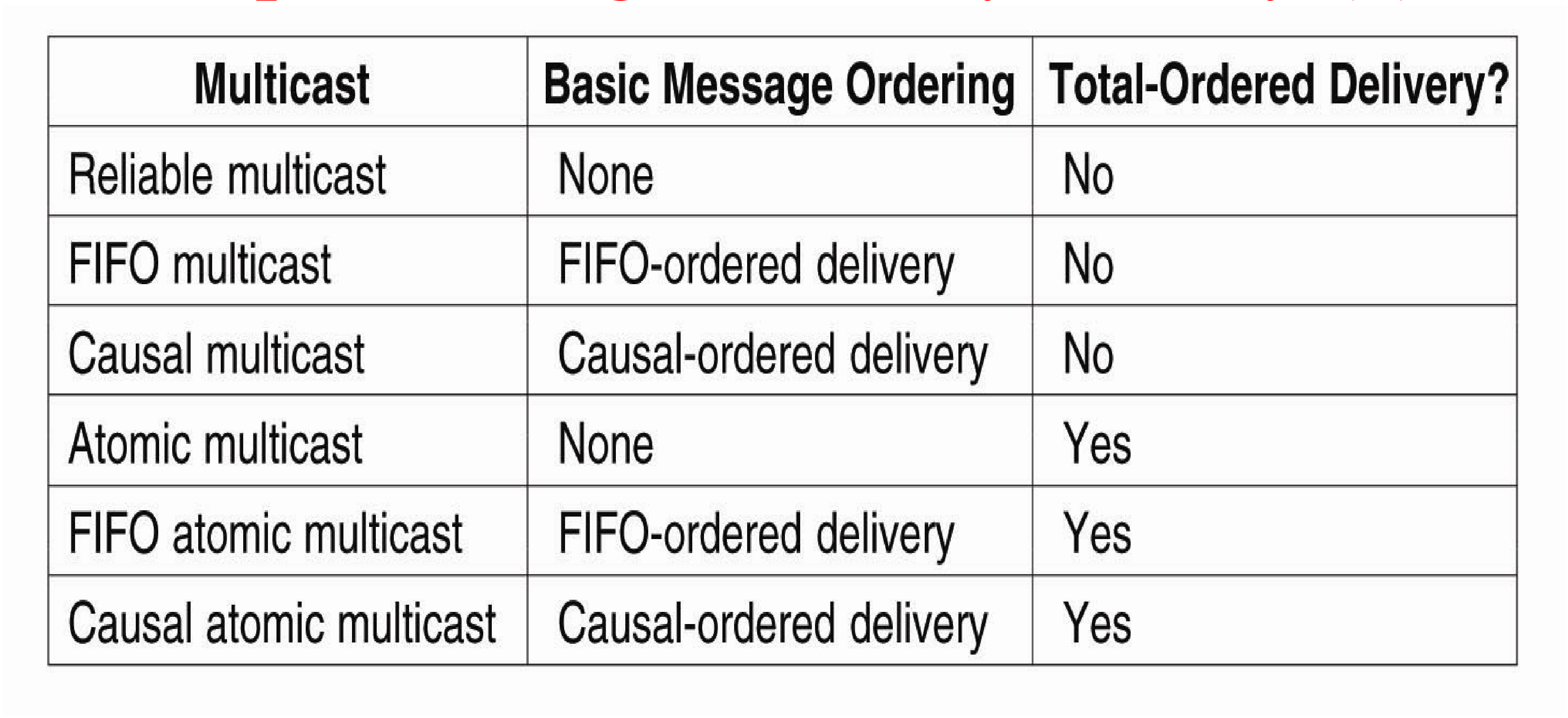
FIFO



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	

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

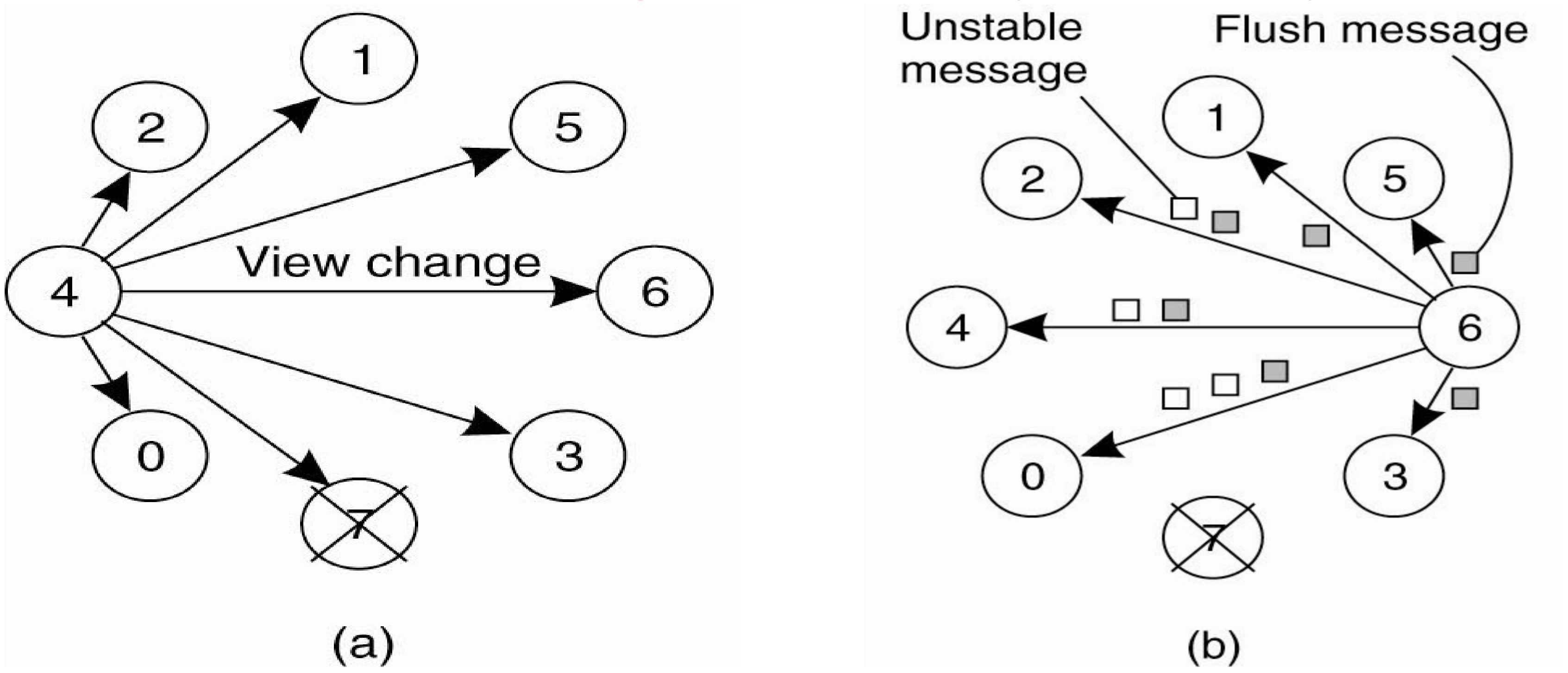
Implementing Virtual Synchrony (1)



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

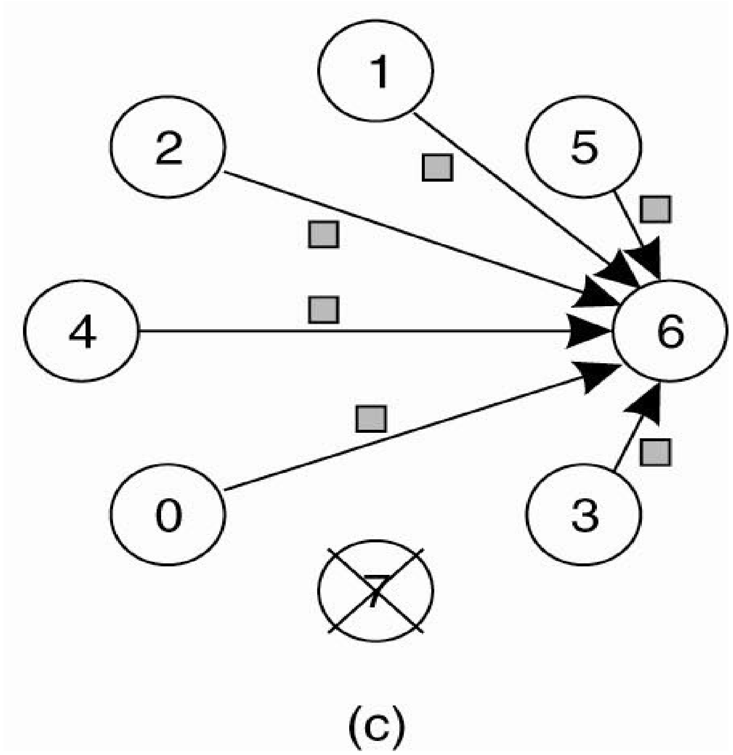
Implementing Virtual Synchrony (2)



(a)Process 4 notices that process 7 has crashed and sends a view change

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

Implementing Virtual Synchrony (3)



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

Distributed Commit

- **Goal**: Either **all** members of a group decide to perform an operation, or **none** of them perform the operation
- **Atomic transaction**: a transaction that happens completely or not at all
- Is required in distributed transactions

Assumptions

- Failures:
  - Crash failures that can be recovered
  - Communication failures detectable by timeouts

- Notes:
  - Commit requires a set of processes to agree…
  - …similar to the Byzantine generals problem…
  - … but the solution much simpler because stronger assumptions

Summary (1 of 2)

- Fault Tolerance:
  - *The characteristic by which a system can mask the occurrence and recovery from failures. A system is fault tolerant if it can continue to operate even in the presence of failures.*
- Types of failure:
  - *Crash* (system halts);
  - *Omission* (incoming request ignored);
  - *Timing* (responding too soon or too late);
  - *Response* (getting the order wrong);
  - *Arbitrary/Byzantine* (indeterminate, unpredictable).

Summary (2 of 2)

- Fault Tolerance is generally achieved through use of *redundancy* and *reliable multitasking protocols*.

- Processes, client/server and group communications can all be "enhanced" to tolerate faults in a distributed system. Commit protocols allow for fault tolerant multicasting (with *two-phase* the most popular type).

- *Recovery* from errors within a Distributed System tends to rely heavily on **Backward Recovery** techniques that employ some type of *checkpointing* or *logging* mechanism, although **Forward Recovery** is also possible.