Chapter 8 - Fault Tolerance

Introduction

- a major difference between distributed systems and single machine systems is that with the former, partial failure is possible, i.e., when one component in a distributed system fails
- such a failure may affect some components while others will continue to function properly
- an important goal of distributed systems design is to construct a system that can automatically recover from partial failure
- it should tolerate faults and continue to operate to some extent

Objectives of the Chapter

- we will discuss
  - fault tolerance, and making distributed systems fault tolerant
  - process resilience (techniques by which one or more processes can fail without seriously disturbing the rest of the system)
  - reliable multicasting to keep processes synchronized (by which message transmission to a collection of processes is guaranteed to succeed)
  - distributed commit protocols for ensuring atomicity in distributed systems
  - failure recovery by saving the state of a distributed system (when and how)

8.1 Introduction to Fault Tolerance

- Basic Concepts
  - fault tolerance is strongly related to dependable systems
  - dependability covers the following
    - availability
      - refers to the probability that the system is operating correctly at any given time; defined in terms of an instant in time
    - reliability
      - a property that a system can run continuously without failure; defined in terms of a time interval
    - safety
      - refers to the situation that even when a system temporarily fails to operate correctly, nothing catastrophic happens
    - maintainability
      - how easily a failed system can be repaired

- dependable systems are also required to provide a high degree of security
- a system is said to fail when it cannot meet its promises; for instance failing to provide its users one or more of the services it promises
- an error is a part of a system’s state the may lead to a failure; e.g., damaged packets in communication
- the cause of an error is called a fault
- building dependable systems closely relates to controlling faults
- a distinction is made between preventing, removing, and forecasting faults
- a fault tolerant system is a system that can provide its services even in the presence of faults

- faults are classified into three
  - transient
    - occurs once and then disappears; if the operation is repeated, the fault goes away; e.g., a bird flying through a beam of a microwave transmitter may cause some lost bits
  - intermittent
    - it occurs, then vanishes on its own accord, then reappears, ...; e.g., a loose connection; difficult to diagnose; take your sick child to the nearest clinic, but the child does not show any sickness by the time you reach there
  - permanent
    - one that continues to exist until the faulty component is repaired; e.g, disk head crash, software bug

- **Failure Types - 5 of them**
  - **Crash failure (also called fail-stop failure): a server halts, but was working correctly until it stopped; e.g., the OS halts; reboot the system**
  - **Omission failure: a server fails to respond to incoming requests**
    - **Receive omission: a server fails to receive incoming messages; e.g., may be no thread is listening**
    - **Send omission: a server fails to send messages**
  - **Timing failure: a server's response lies outside the specified time interval; e.g., may be it is too fast over flooding the receiver or too slow**
  - **Response failure: the server's response is incorrect**
    - **Value failure: the value of the response is wrong; e.g., a search engine returning wrong Web pages as a result of a search**

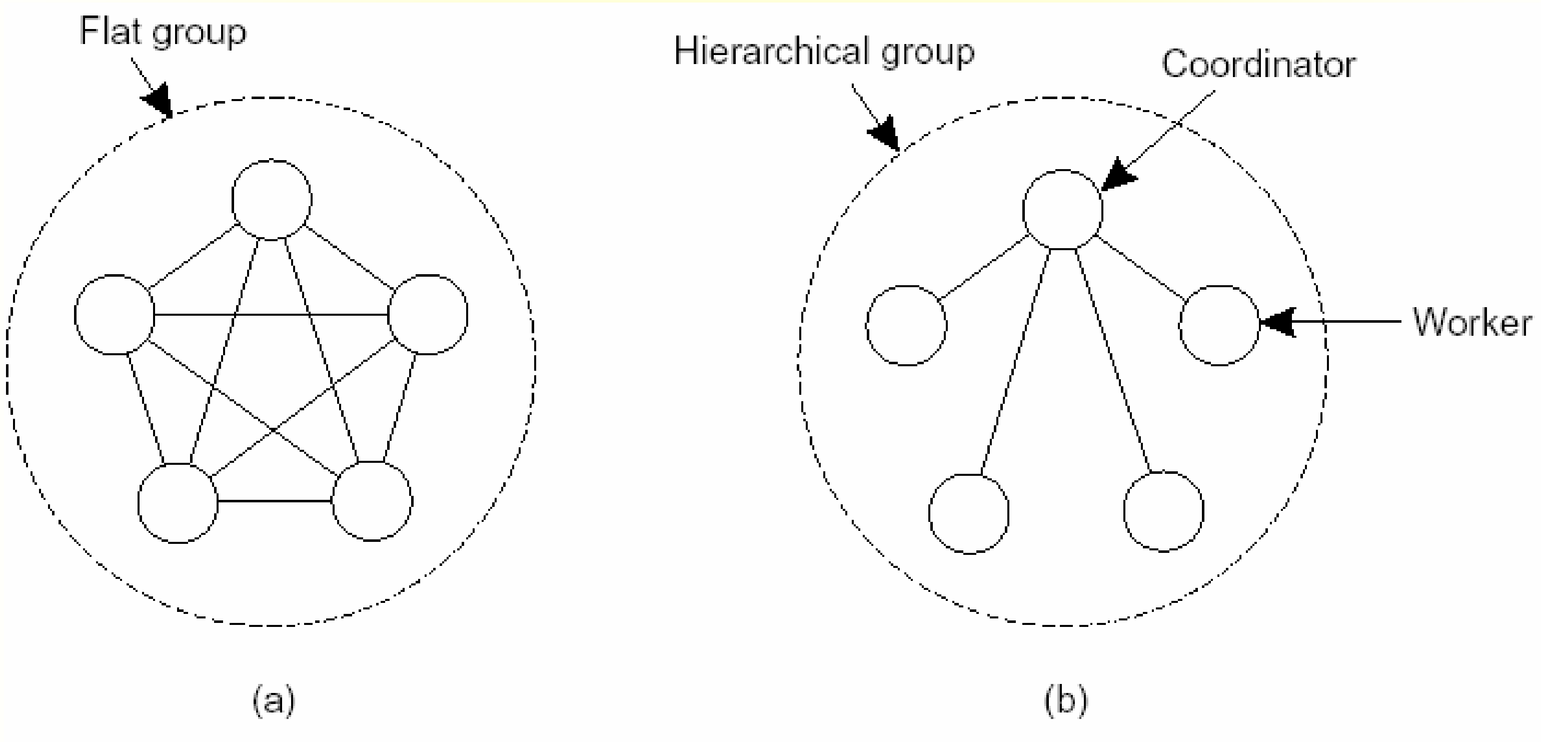
    - State transition failure: the server deviates from the correct flow of control; e.g., taking default actions when it fails to understand the request
  - Arbitrary failure (or Byzantine failure): a server may produce arbitrary responses at arbitrary times; the most serious
- Failure Masking by Redundancy
  - to be fault tolerant, the system tries to hide the occurrence of failures from other processes - masking
  - the key technique for masking faults is redundancy
  - three kinds are possible
    - information redundancy; add extra bits to allow recovery from garbled bits (error correction)
    - time redundancy: an action is performed more than once if needed; e.g., redo an aborted transaction; useful for transient and intermittent faults
    - physical redundancy: add (replicate) extra equipment (hardware) or processes (software)

8.2 Process Resilience

- how can fault tolerance be achieved in distributed systems
- one method is protection against process failures by replicating processes into groups
- we discuss
  - what are the general design issues of process groups
  - what actually is a fault tolerant group
  - how to reach agreement within a process group when one or more of its members cannot be trusted to give correct answers

- Design Issues
  - the key approach to tolerating a faulty process is to organize several identical processes into a group
  - all members of a group receive a message hoping that if one process fails, another one will take over
  - the purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction
  - process groups may be dynamic
    - new groups can be created and old groups can be destroyed
    - a process can join or leave a group
    - a process can be a member of several groups at the same time
  - hence group management and membership mechanisms are required

- **the internal structure of a group may be flat or hierarchical**
  - **flat: all processes are equal and decisions are made collectively**
  - **hierarchical: a coordinator and several workers; the coordinator decides which worker is best suited to carry a task**



(a) communication in a flat group

(b) communication in a simple hierarchical group

- the flat group
  - has no single point of failure
  - but decision making is more complicated (voting may be required for decision making which may create a delay and overhead)
- the hierarchical group has the opposite properties
- group membership may be handled
  - through a group server where all requests (joining, leaving, ...) are sent; it has a single point of failure
  - in a distributed way where membership is multicasted (if a reliable multicasting mechanism is available)
    - but what if a member crashes; other members have to find out this by noticing that it no more responds

- Failure Masking and Replication
  - how to replicate processes so that they can form groups and failures can be masked?
  - there are two ways for such replication (as discussed in the previous chapter)
    - primary-based replication
      - for fault tolerance, primary-backup protocol is used
      - organize processes hierarchically and let the primary (i.e., the coordinator) coordinate all write operations
      - if the primary crashes, the backups hold an election
    - replicated-write protocols
      - in the form of active replication or by means of quorum-based protocols
      - that means, processes are organized as flat groups

- another important issue is how much replication is needed
- for simplicity consider only replicated-write systems
- a system is said to be k fault tolerant if it can survive faults in k components and still meets its specifications
  - if the processes fail silently, then having k+1 replicas is enough; if k of them fail, the remaining one can function
  - if processes exhibit Byzantine failure, 2k+1 replicas are required; if the k processes generate the same reply (wrong), the k+1 will also produce the same answer (correct); but which of the two is correct is difficult to ascertain; unless we believe the majority

- **Agreement in Faulty Systems**
  - agreement is required in many cases; some examples are
    - electing a coordinator
    - deciding whether or not to commit a transaction
    - dividing tasks among workers
  - the objective of all agreement algorithms is to have all the non-faulty processes reach consensus on some issue in a finite number of steps
  - but, reaching an agreement in a faulty environment (if process and/or communication failure exists) is difficult; two cases

    1. processes are perfect but communication is unreliable
       - this is similar to the problem known as the two-army problem; the sender of a message does not know that its message has arrived
       - even if the receiver acknowledges, it does not know that its acknowledgement has been received
       - hence, it is even difficult to reach an agreement when there are only two perfect processes

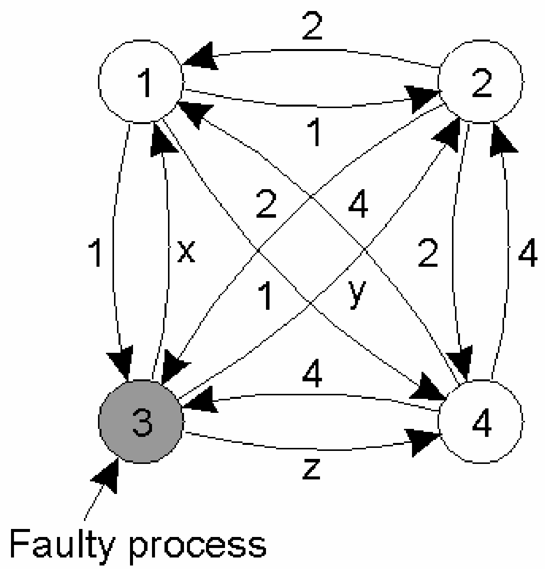
2. communication is perfect but processes are not

- this is also similar to the problem known as the Byzantine agreement (generals) problem, where out of n generals k are traitors (they send wrong responses); can the loyal generals reach an agreement?
- a recursive algorithm by Lamport
  - assume that each general is assumed to know how many troops s/he has
  - the goal is to exchange troop strengths (by telephone, hence reliable communication)
  - at the end of the algorithm, each general has a vector of length n corresponding to all the armies
  - if general i is loyal, then element i is his/her troop strength; otherwise, it is undefined

- assume that
  - messages are unicast while preserving ordering
  - communication delay is bounded
  - there are N processes where each process i will provide a value v_i to the others
  - there are at most k faulty processes
- the algorithm works in four steps (converting generals to processes, if possible ☺)

1. each nonfaulty process i sends v_i to every other process; faulty processes may send different values to different processes since we are using unicasting
2. the results are collected in the form of vectors
3. every process passes its vector to every other process
4. each process examines the ith element; if any value has a majority, that value is put into the result vector; else is marked UNKNOWN

- **e.g.1, for N = 4, 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)

the Byzantine agreement problem for 3 nonfaulty and 1 faulty processes

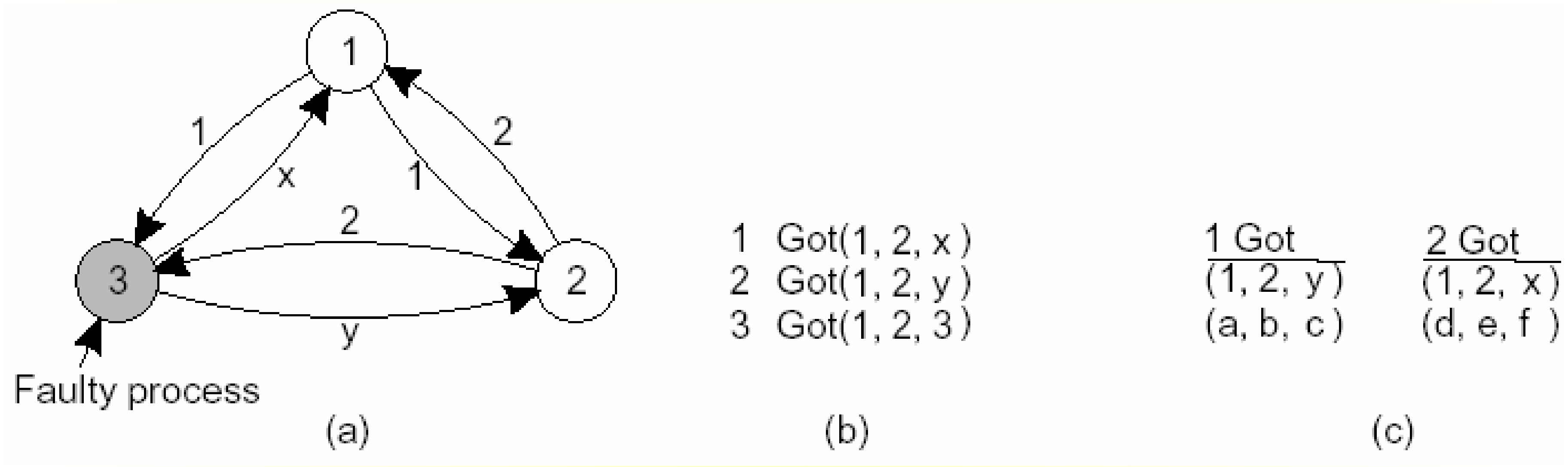
a) each process sends their value to the others

b) the vectors that each process assembles based on (a)

c) the vectors that each process receives in step 3

- **processes 1, 2, and 4 all come to an agreement on (1, 2, UNKNOWN, 4); consensus is reached on the value for the nonfaulty processes only, i.e., the system works**

- **e.g.2, the algorithm fails for N = 3, k = 1**



- **processes 1 and 2 could not reach an agreement; no majority for all elements**
- **it has been proved that in a system with k faulty processes, agreement can be reached only if 2k+1 correctly functioning processes are present, for a total of 3k+1; i.e., only if more than 2/3rds of the processes are working properly**

- Failure Detection
  - to mask failures, we need to detect them
  - for a group of processes, nonfaulty members should be able to decide who is still a member, and who is not, i.e., detecting the failure of a member
  - two possibilities
    1. pinging; processes send “are you alive?” messages to each other; set a timeout mechanism; but
       - if the network is unreliable, timeout does not necessarily mean failure (there could be false positive)
    2. passively wait until messages come in from different processes; only if there is enough communication between processes
  - another approach is gossiping where each node regularly announces to its neighbors that it is still up and running

- **network failures must also be distinguished from node failures**
    - **during timeout, send a request to other neighbors to see if they can reach the failing node**

8.3 Reliable Client-Server Communication

- **fault tolerance in distributed systems concentrates on faulty processes**
- **but communication failures also have to be considered**
- **a communication channel may exhibit failures in the form of**
    - **crash**
    - **omission**
    - **timing**
    - **arbitrary (duplicate messages as a result of buffering at nodes and the sender retransmitting)**

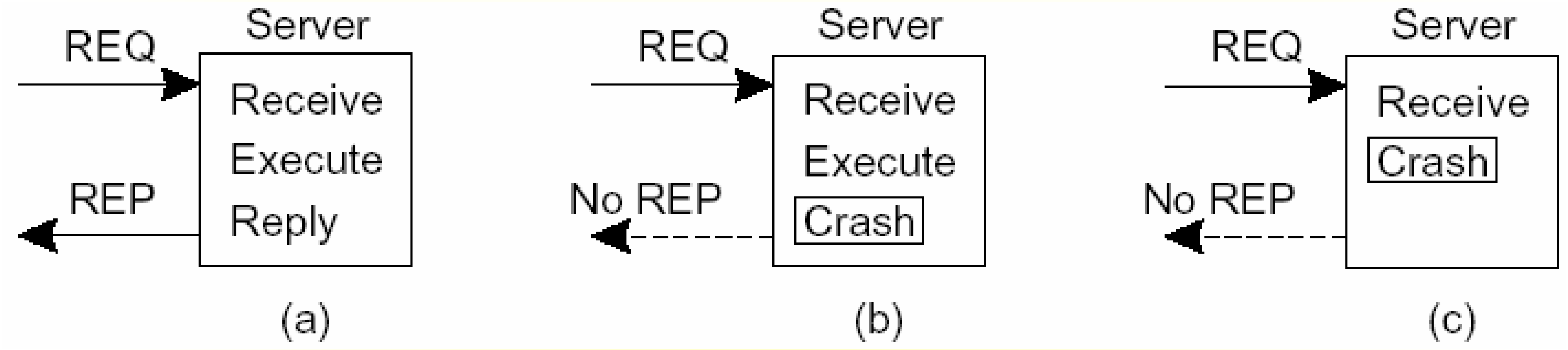
- Point-to-Point Communication
  - reliable transport protocols such as TCP can be used that mask most communication failures such as omissions (lost messages) using acknowledgements and retransmissions
  - however, crash failures are often not masked; e.g., a broken TCP connection; in which case the system may probably setup a new connection automatically

- RPC Semantics in the Presence of Failures
  - the goal of RPC is to hide communication by making remote procedure calls look like local ones; may be difficult to mask remote calls when there are failures
  - five different classes of failures can occur in RPC systems, each requiring a different solution
    - the client is unable to locate the sever
    - the client’s request to the server is lost
    - the server receives the request and crashes
    - the reply message from the server to the client is lost
    - the client crashes after sending a request

1. the client is unable to locate the sever
   - may be the client is unable to locate a suitable server or the server is down
   - or an interface at the server has been changed making the client’s stub obsolete (it was not used for a long time) and the binder failing to match it with the server
   - one solution: let the client raise an exception and have exception handlers; inserted by the programmer
   - examples are exceptions in Java or signal handlers in C
   - but
     - not every language may provide exceptions or signals
     - the major objective of having transparency is violated
2. the client’s request to the server is lost
   - let the OS or the client stub start a timer when sending a request; if the timer expires before a reply or acknowledgement comes back, the message is retransmitted

3. the server receives the request and crashes

- the failure can happen before or after execution



a server in client-server communication; (a) normal case, (b) crash after execution, (c) crash before execution

- in (b), let the system report failure back to the client (raise an exception)
- in (c) retransmit the request
- but the client's OS doesn't know which is which; only a timer has expired

- three schools of thought on the possible solutions
  - the client tries the operation again after the server has restarted (or rebooted) or rebinds to a new server, called at least once semantics; keep on trying until a reply is received; it guarantees that the RPC has been carried out at least one time, but possibly more
  - the client gives up after the first attempt and reports error, called at most once semantics; it guarantees that the RPC has been carried out at most one time, but possibly none at all
  - no guaranty at all (0-N times)
- none of the above is the right solution; what is required is exactly once semantics; but difficult to achieve

- as an example, assume that
  - a client would like to print some text
  - the client receives an acknowledgement of receipt
- the server can send a completion message and print or can print and then send a completion message (2 possibilities)
- assume the server crashes and recovers, announces this; the client doesn’t know if its request will be carried out
- what should the client do?; four possibilities
  - never reissue a request (the text may not be printed)
  - reissue a request (the text may be printed twice)
  - assuming the server crashed before the request was delivered, it can decide to reissue a request only if it has not received an acknowledgement
  - reissue a request only if it has received an acknowledgement
- hence, with 2 strategies for the server and 4 for the client, there are a total of 8 possibilities, none of which is satisfactory

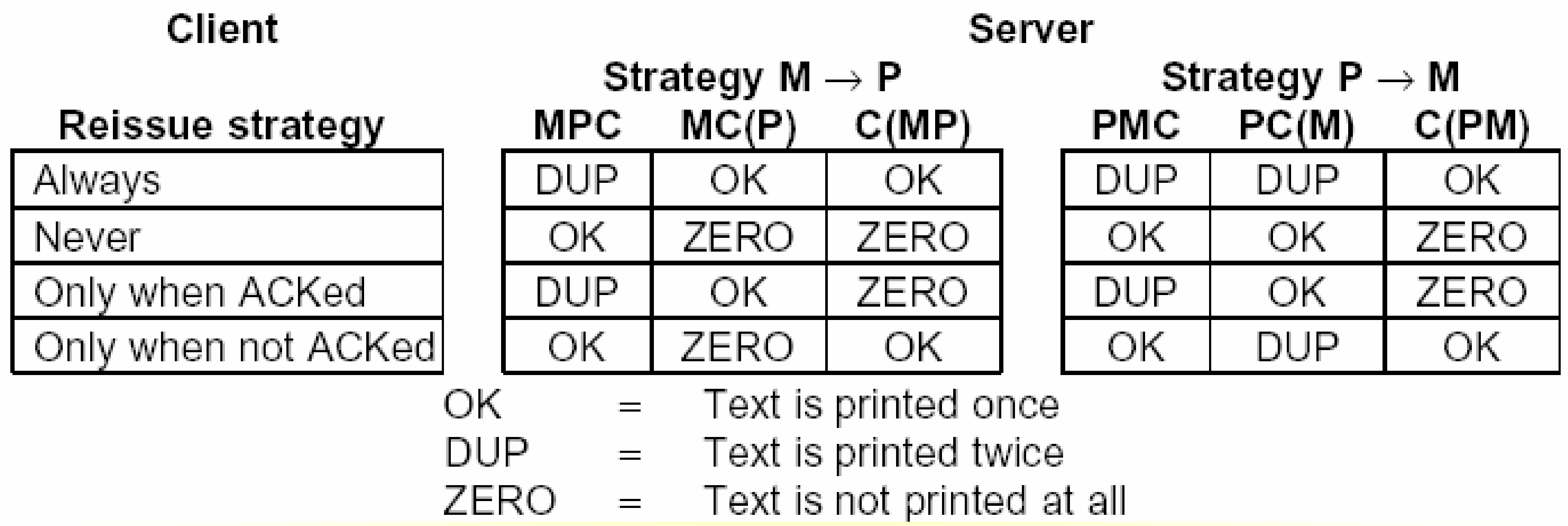
- three possible events at the server: send the completion message (M), print the text (P), crash (C)
- six different orderings of these events (the parentheses indicate an event that can no longer happen)

a. if the server's strategy is M → P

  - M →P→C - a crash occurs after sending the completion message and printing the text
  - M→C(→P)
  - C(→M→P)

b. if the server's strategy is P → M

  - P→M→C
  - P→C(→M)
  - C(→P→M)



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

different combinations of client and server strategies in the presence of server crashes

- **none of all the combinations works correctly under all possible event sequences; the client can never know whether the server crashes before or after printing**

4. the reply message from the server to the client is lost

- the client sets a timer and resends
- there is a risk that the request is sent more than once; may be the server was slow
- however, if the operation is idempotent (an operation such as requesting a read that can be repeated without any harm), it does not create any problem; but what if the request is to transfer money from one account to another?
- let the client assign a sequence number to each request so that the server knows it is a duplicated one; but needs the server to maintain information on each client and for how long should it maintain it
- in addition, we can have a bit in the message header that is used to distinguish initial requests from retransmissions so as to take more care on retransmissions

5. the client crashes after sending a request

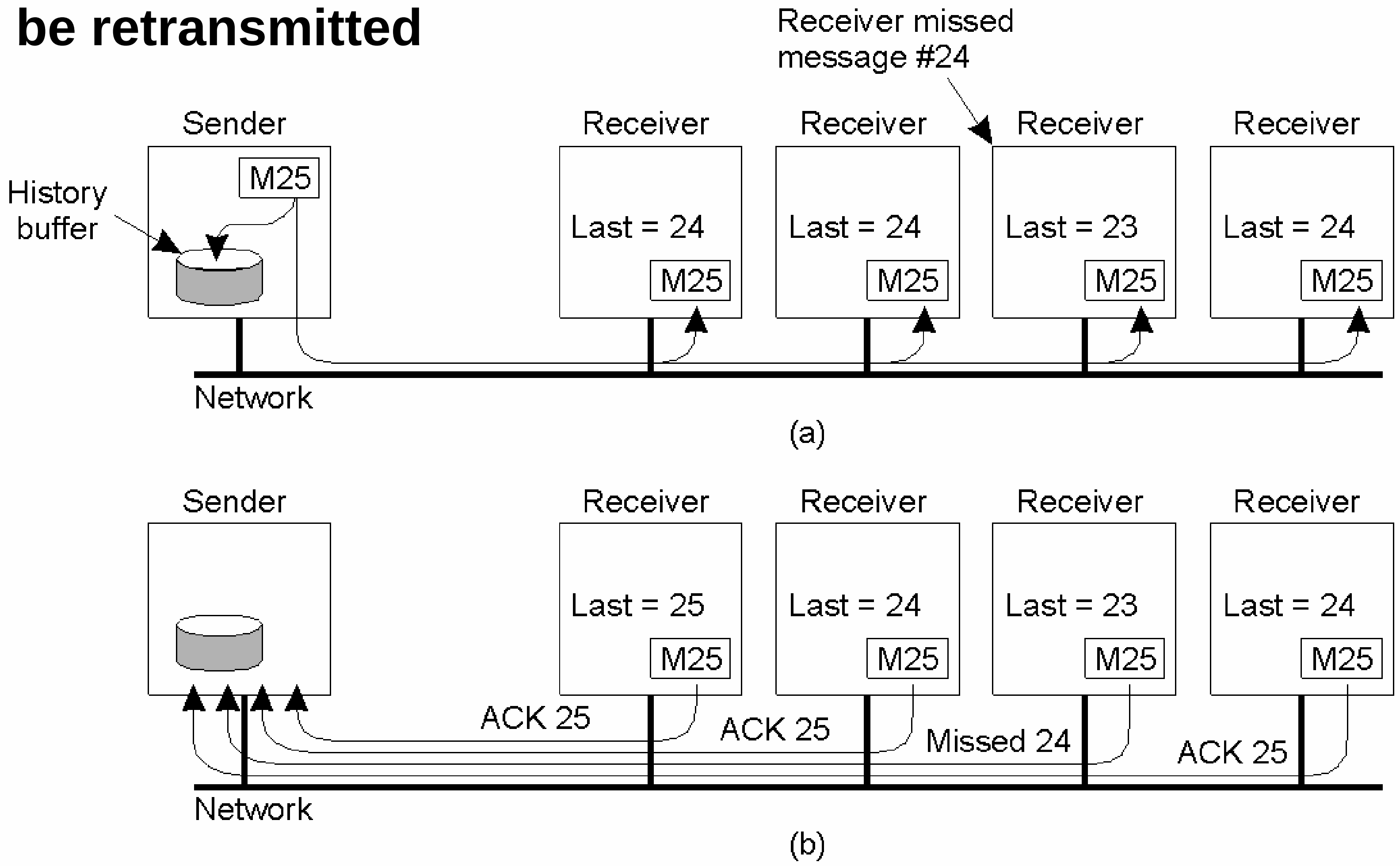
- in this case a computation may be done without a parent waiting for the result; such unwanted computations are called orphans
- orphans
  - waste CPU cycles
  - can lock valuable resources such as files
  - can cause confusion if the client reboots and does the RPC again and the reply from the orphan comes back immediately afterward
- proposed solutions
  - extermination: let the client stub maintain a log on disk before sending an RPC message; after a reboot, inspect the log and kill the orphan; some problems
    - the expense of writing a log for every RPC
    - it may not work if the orphans themselves do RPC, creating grandorphans that are difficult to locate

- **reincarnation: divide time into sequentially numbered epochs; when a client reboots, it broadcasts a message to all machines declaring the start of a new epoch; when such a broadcast comes in, all remote computations on behalf of that client are killed**
- **gentle reincarnation: when an epoch broadcast comes in, each machine checks to see if it has remote computations, and if so, tries to locate their owner; a computation is killed only if the owner cannot be found**
- **expiration: each RPC is given a standard amount of time, T, to do the job; if it cannot finish, it must explicitly ask for another quantum; if after a crash the client waits a time T before rebooting, all orphans are killed; the problem is how to choose a reasonable value of T, since RPCs have differing needs**
- **none of the solutions is satisfactory; still locked files or database records will remain locked if an orphan is killed, etc.**

8.4 Reliable Group Communication

- how to reliably deliver messages to a process group (multicasting)
- Basic Reliable-Multicasting Schemes
    - reliable multicasting means a message sent to a process group should be delivered to each member of that group
    - transport protocols do not offer reliable communication to a collection of processes
    - problems:
        - what happens if a process joins a group during communication? should it receive the message?
        - what happens if a (sending) process crashes during communication?
        - what if there are faulty processes?

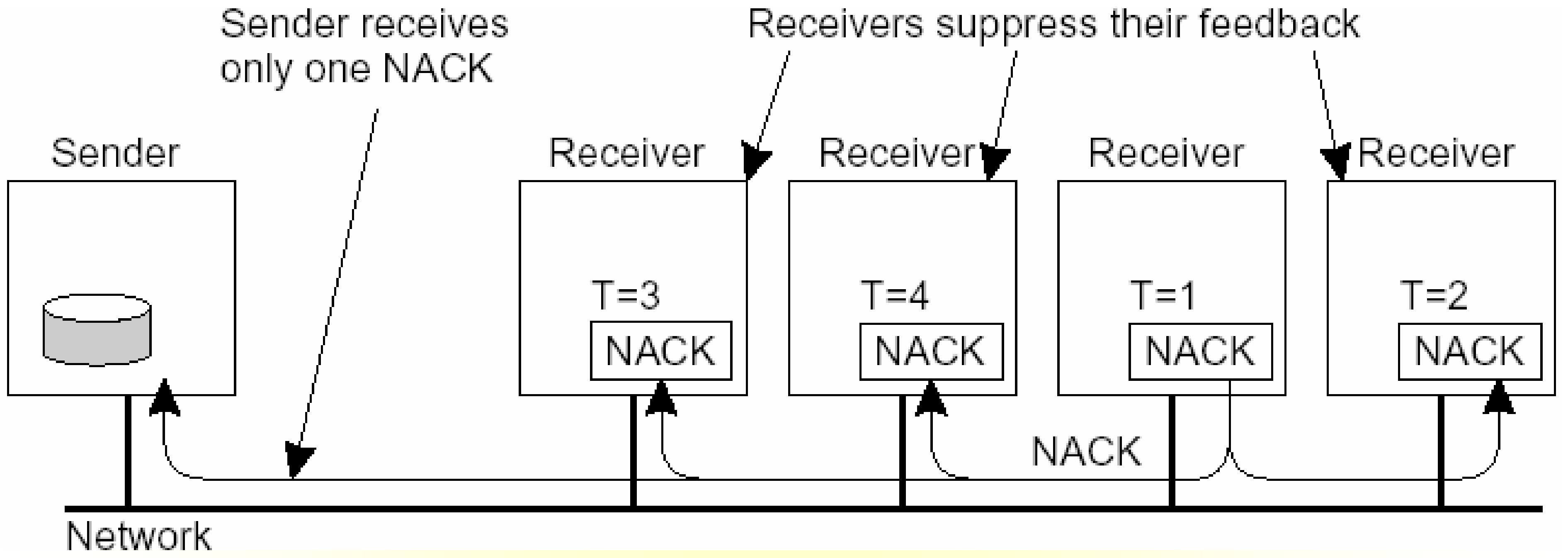
- **a weaker solution assuming that all receivers are known (and they are limited) and that none will fail is for the sending process to assign a sequence number to each message and to buffer all messages so that lost ones can be retransmitted**



a simple solution to reliable multicasting when all receivers are known and are assumed not to fail; (a) message transmission, (b) reporting feedback

- Scalability in Reliable Multicasting
  - a reliable multicast scheme does not support large number of receivers; there will be feedback implosion
  - two approaches for solving the problem

1. Nonhierarchical Feedback Control
   - reduce the number of feedback messages returned to the sender using feedback suppression
   - the Scalable Reliable Multicasting (SRM) Protocol uses this principle
   - only negative acks are returned as feedback and they are multicasted to all so that others that missed the message will refrain from sending a negative ack
   - a receiver R that did not receive a message M schedules a feedback message with some random delay T; if another message for retransmitting M reaches R, R suppresses its transmission; else it transmits
   - the sender will receive only a single negative ack and retransmits M to all

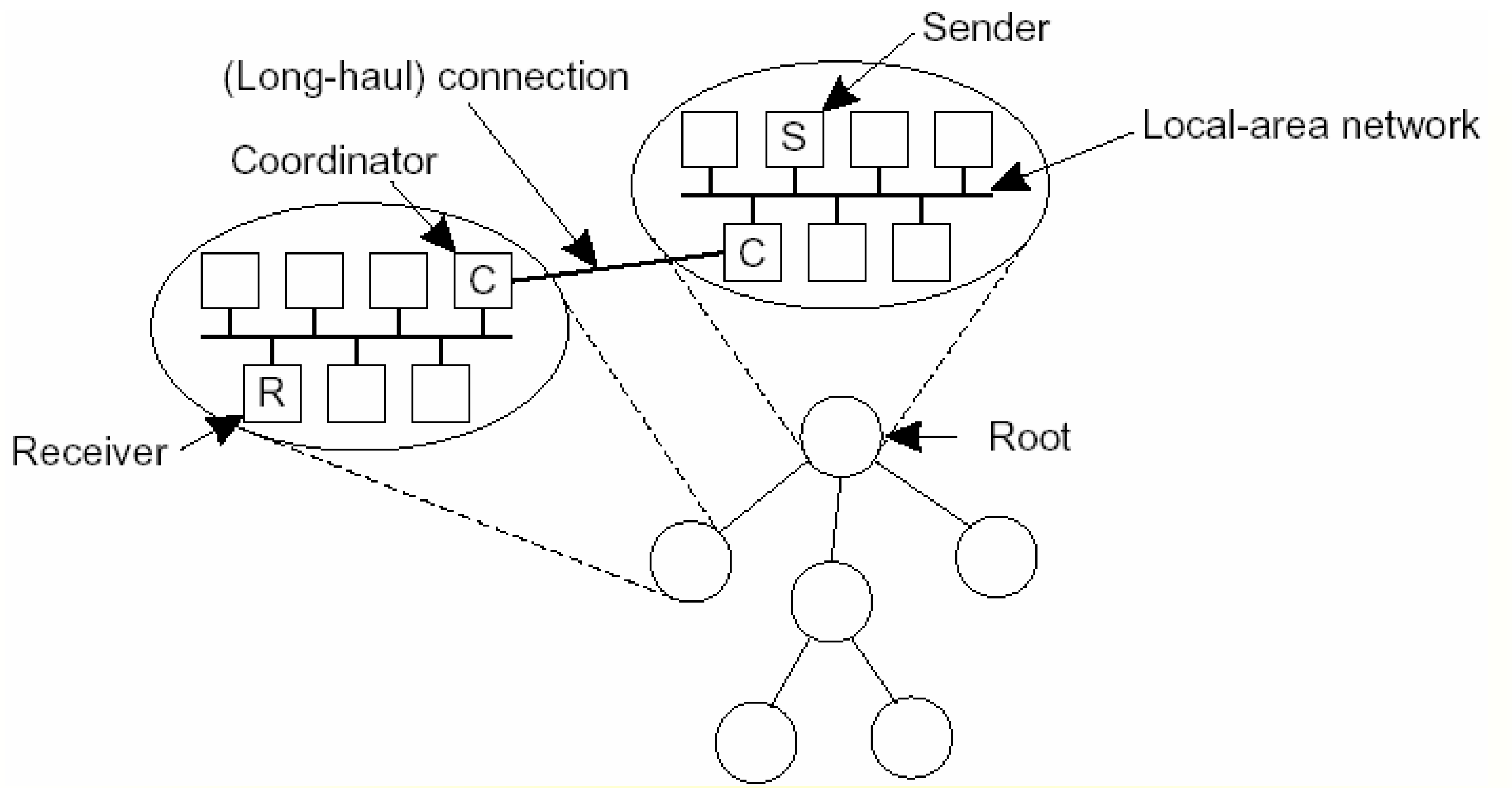


several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others

- **feedback suppression scales well**
- **disadvantages**
  - **ensuring that only one feedback will reach the sender is difficult; there is a possibility of many receivers sending their feedback at the same time**
  - **receivers that successfully receive a message waste their time processing negative acks and the duplicated message**

2. Hierarchical Feedback Control

- hierarchical solutions are better to achieve scalability for very large groups of receivers
- the group of receivers is partitioned into a number of subgroups, subsequently organized into a tree; the subgroup containing the sender (assume a single sender) forms the root of the tree
- within each subgroup, use any reliable multicasting scheme that works for small groups
- each subgroup appoints a local coordinator, responsible for handling retransmission requests in its subgroup; hence it has its history buffer
- if the coordinator has missed a message, it asks the coordinator of the parent subgroup to retransmit it
- if a coordinator has received acks for a message from all members of its subgroup as well as from its children, it can remove the message from its history buffer



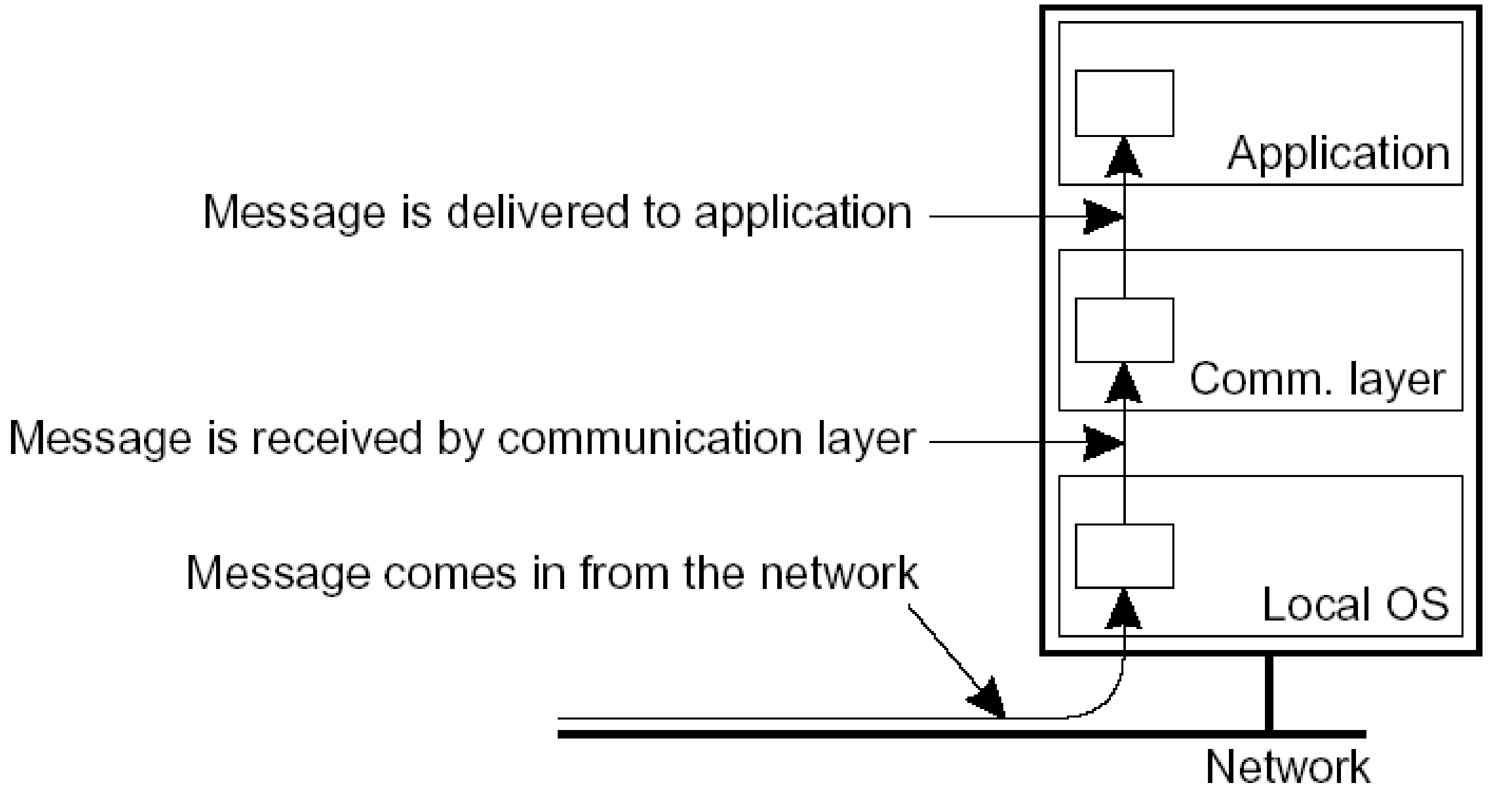
the essence of hierarchical reliable multicasting; each local coordinator forwards the message to its children and later handles retransmission requests

- **dynamically constructing the tree is a problem**

➲ **building reliable multicast schemes that can scale to a large number of receivers spread in a WAN is a difficult problem; more research is required**

- Atomic Multicast
  - how to achieve reliable multicasting in the presence of process failures
  - for example, in a replicated database, how to handle update operations when a replica crashes during update operations
  - the atomic multicast problem: to guarantee that a message is delivered to either all processes or none at all and that messages are delivered in the same order to all processes

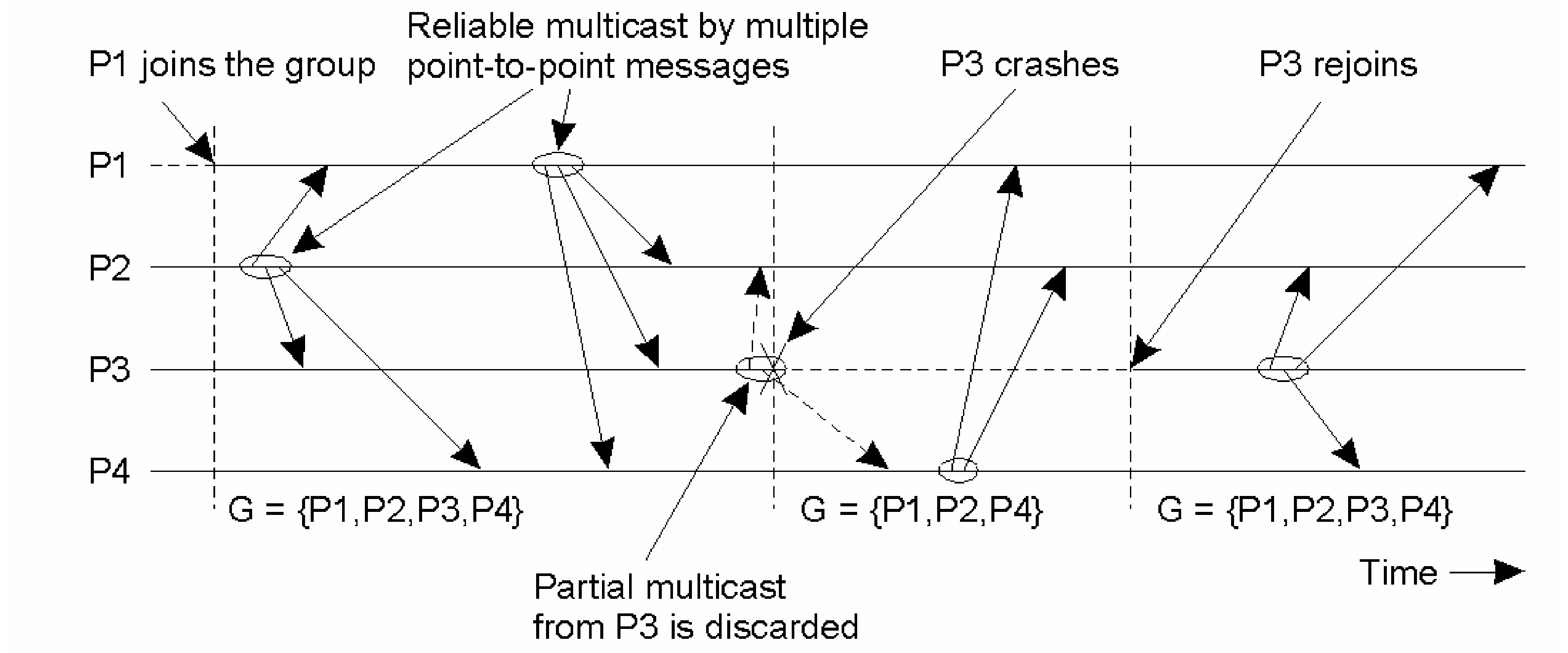
- Virtual Synchrony
  - **consider the following model in which the distributed system consists of a communication layer for sending and receiving messages; a received message is locally buffered in this layer until it is delivered to the application**


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

the logical organization of a distributed system to distinguish between message receipt and message delivery

- the whole idea of atomic multicasting is that a multicast message m is uniquely associated with a list of processes to which it should be delivered; this delivery list corresponds to a group view G, the view on the set of processes contained in the group, which the sender had at the time message m was multicast
- a view change takes place by multicasting a message vc announcing the joining or leaving of a process, while the multicasting takes place
- then there are two messages in transit: m and vc; we need a guarantee that m is either delivered to all processes in G before each of them receives vc, or m is not delivered at all
- a reliable multicast guarantees that a message multicast to group G is delivered to each non-faulty process in G; if the sender 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

- **e.g., 4 processes; P3 crashes after successfully multicasting a message to P2 and P4 but not to P1; virtual synchrony guarantees that the message is not delivered at all; later P3 can join after its state has been brought up to date**



- **hence, all multicasts take place between view changes; a view change acts as a barrier across which no multicast can pass; similar to synchronization variables used in distributed data stores (Chapter 7)**

- Message Ordering
  - there are four different orderings

1. Unordered multicasts
   - a reliable, unordered multicast is a virtual synchronous multicast in which no guarantees are given concerning the order in which received messages (by the communication layer) are delivered by different processes

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

2. FIFO-ordered multicasts

- **for reliable FIFO-ordered multicasts, the communication layer is forced to deliver incoming messages from the same process in the same order as they have been sent**
- **in the following example, m1 must always be delivered before m2; same to m3 and m4; but no constraint regarding messages originating from different processes**

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

3. Causally-ordered multicasts

- reliable causally-ordered multicast delivers messages so that potential causality between different messages is preserved (even if they originate from different processes)

4. Totally-Ordered Multicasts

- besides the above three, a further constraint can be used; regardless of whether message delivery is unordered, FIFO ordered, or causally ordered, when messages are delivered, they are delivered in the same order to all group members
- i.e., totally-ordered multicast is used in combination with one of the three
- in the previous example of FIFO-ordered multicast, P2 and P3 must deliver messages m1, m2, m3, and m4 in the same order (still respecting FIFO ordering)
- virtually synchronous reliable multicasting offering totally-ordered delivery of messages is called atomic multicasting

- **hence, there are a total of six forms of reliable multicasting (the three message orderings combined with atomicity)**

Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable unordered 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**

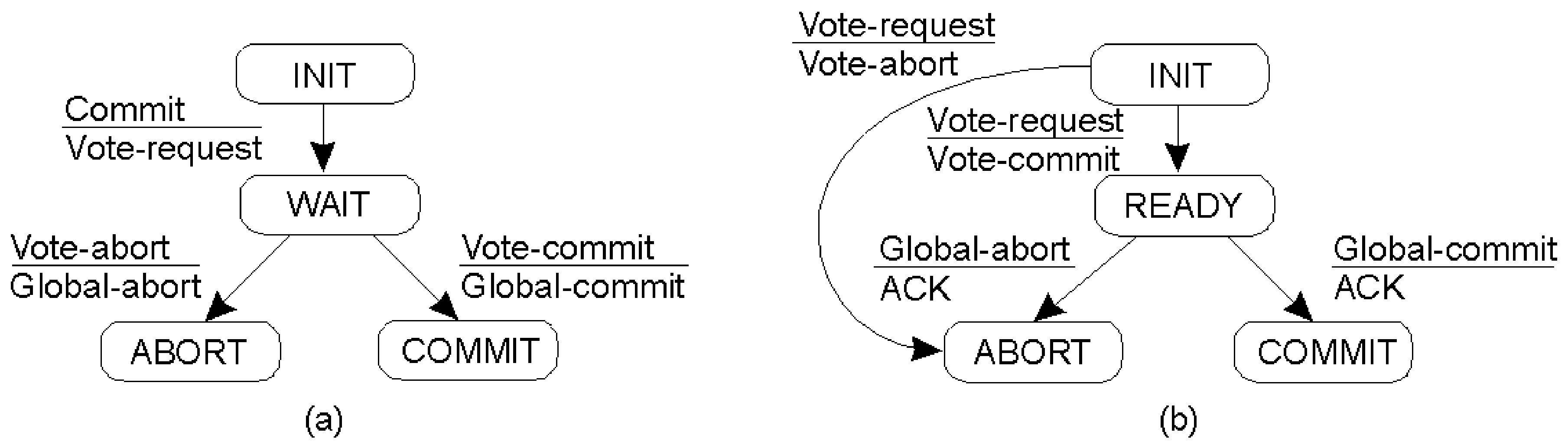
six different versions of virtually synchronous reliable multicasting

- **read about Isis, a fault-tolerant distributed system, as an example of implementing virtual synchrony - pages 352 - 354**

8.5 Distributed Commit

- atomic multicasting is an example of the more generalized problem known as distributed commit
- in atomic multicasting, the operation is delivery of a message
- but the distributed commit problem involves having an(y) operation being performed by each member of a process group, or none at all
- there are three protocols: one-phase commit, two-phase commit, and three-phase commit
- One-Phase Commit Protocol
    - a coordinator tells all other processes, called participants, whether or not to (locally) perform an operation
    - drawback: if one of the participants cannot perform the operation, there is no way to tell the coordinator; for example due to violation of concurrency control constraints in distributed transactions

- Two-Phase Commit Protocol (2PC)
  - it has two phases: voting phase and decision phase, each involving two steps
  - voting phase
    - the coordinator sends a VOTE_REQUEST message to all participants
    - each participant then sends a VOTE_COMMIT or VOTE_ABORT message depending on its local situation
  - decision phase
    - the coordinator collects all votes; if all vote to commit the transaction, it sends a GLOBAL_COMMIT message; if at least one participant sends VOTE_ABORT, it sends a GLOBAL_ABORT message
    - each participant that voted for a commit waits for the final reaction of the coordinator and commits or aborts


INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Global-commit
ABORT
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Global-commit / ACK
ABORT
COMMIT
(b)

a) *the finite state machine for the coordinator in 2PC*
b) *the finite state machine for a participant*

- problems may occur in the event of failures
  - the coordinator and participants have states in which they block waiting for messages: INIT, READY, WAIT
  - when a process crashes, other processes may wait indefinitely
- hence, timeout mechanisms are required
- a participant waiting in its INIT state for VOTE_REQUEST from the coordinator aborts and sends VOTE_ABORT if it does not receive a vote request after some time
- the coordinator blocking in state WAIT aborts and sends GLOBAL_ABORT if all votes have not been collected on time
- a participant P waiting in its READY state waiting for the global vote cannot abort; instead it must find out which message the coordinator actually sent
  - by blocking until the coordinator recovers
  - or requesting another participant, say Q

State of Q	Action by P	Comments
COMMIT	Make transition to COMMIT	Coordinator sent GLOBAL_COMMIT before crashing, but P didn’t receive it
ABORT	Make transition to ABORT	Coordinator sent GLOBAL_ABORT before crashing, but P didn’t receive it
INIT	Make transition to ABORT	Coordinator sent VOTE_REQUEST before crashing, P received it but not Q
READY	Contact another participant	If all are in state READY, wait until the coordinator recovers

actions taken by a participant P when residing in state READY and having contacted another participant Q

- **a process (participant or coordinator) can recover from crash if its state has been saved to persistent storage**

- **actions by a participant after recovery**

State before Crash	**Action by the Process after Recovery**
INIT	Locally abort the transaction and inform the coordinator
COMMIT or ABORT	Retransmit its decision to the coordinator
READY	Can not decide on its own what it should do next; has to contact other participants

- **there are two critical states for the coordinator**

State before Crash	**Action by the Coordinator after Recovery**
WAIT	Retransmit the VOTE_REQUEST message
After the Decision in the 2nd phase	Retransmit the decision

actions by the coordinator:

```
write START_2PC to local log; /* records it is entering the WAIT state */
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;          voting phase
        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 {                                           decision phase
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

outline of the steps taken by the coordinator in a two-phase commit protocol actions by a participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator; //by a separate thread
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;
}
```

steps taken by a participant process in 2PC

actions for handling decision requests: /*executed by a separate thread */

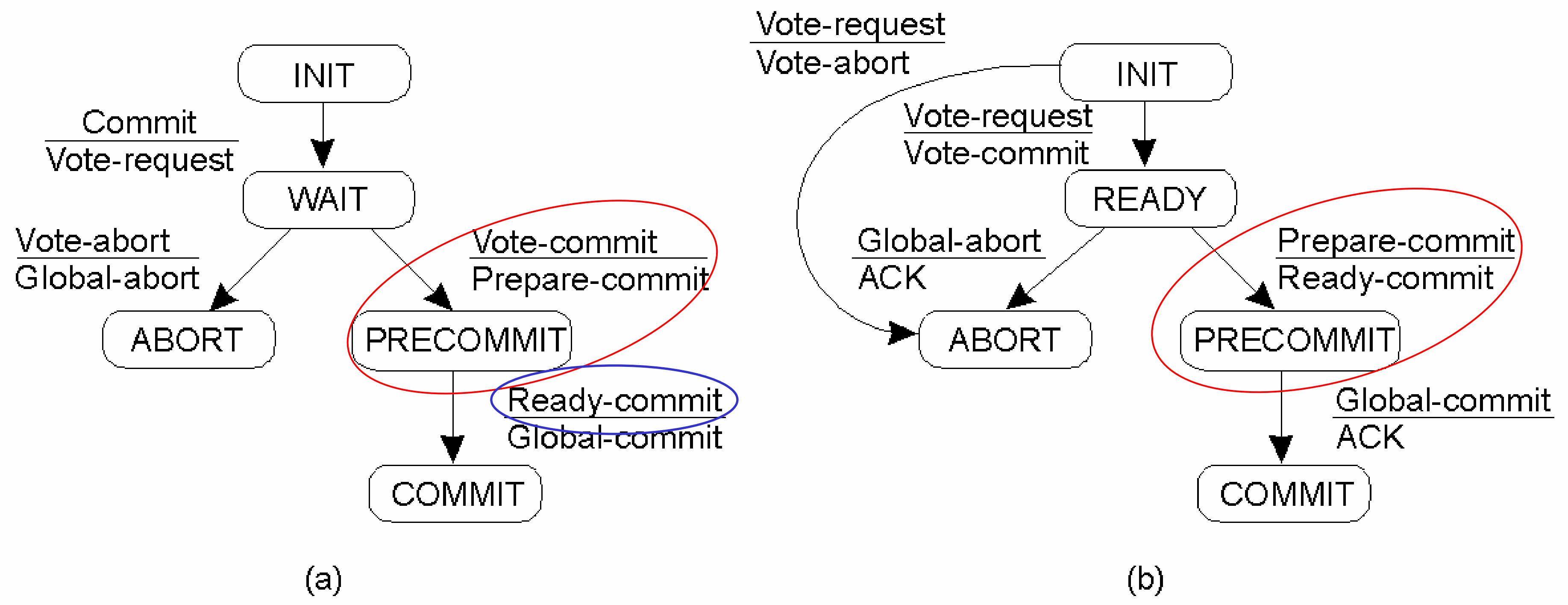
```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send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

}
```

steps taken for handling incoming decision requests (by a participant)

- **if the coordinator crashed after all participants have received a VOTE_REQUEST, they cannot cooperate to reach a decision and all must block; very bad**
- **hence 2PC is also referred to as blocking commit protocol**

- Three-Phase Commit Protocol (3PC)
  - the problem with 2PC is that, if the coordinator crashes, participants will need to block until the coordinator recovers (although such a probability is low and hence 2PC is widely used)
  - 3PC avoids blocking processes in the presence of crashes
  - the states of the coordinator and each participant satisfy the following two conditions (necessary and sufficient conditions for a commit protocol to be nonblocking)
    - there is no single state from which it is possible to make a transition directly to either 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


INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
ABORT
Vote-commit / Prepare-commit
PRECOMMIT
Ready-commit / Global-commit
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
ABORT
Prepare-commit / Ready-commit
PRECOMMIT
Global-commit / ACK
COMMIT
(b)

a) *finite state machine for the coordinator in 3PC*
b) *finite state machine for a participant*

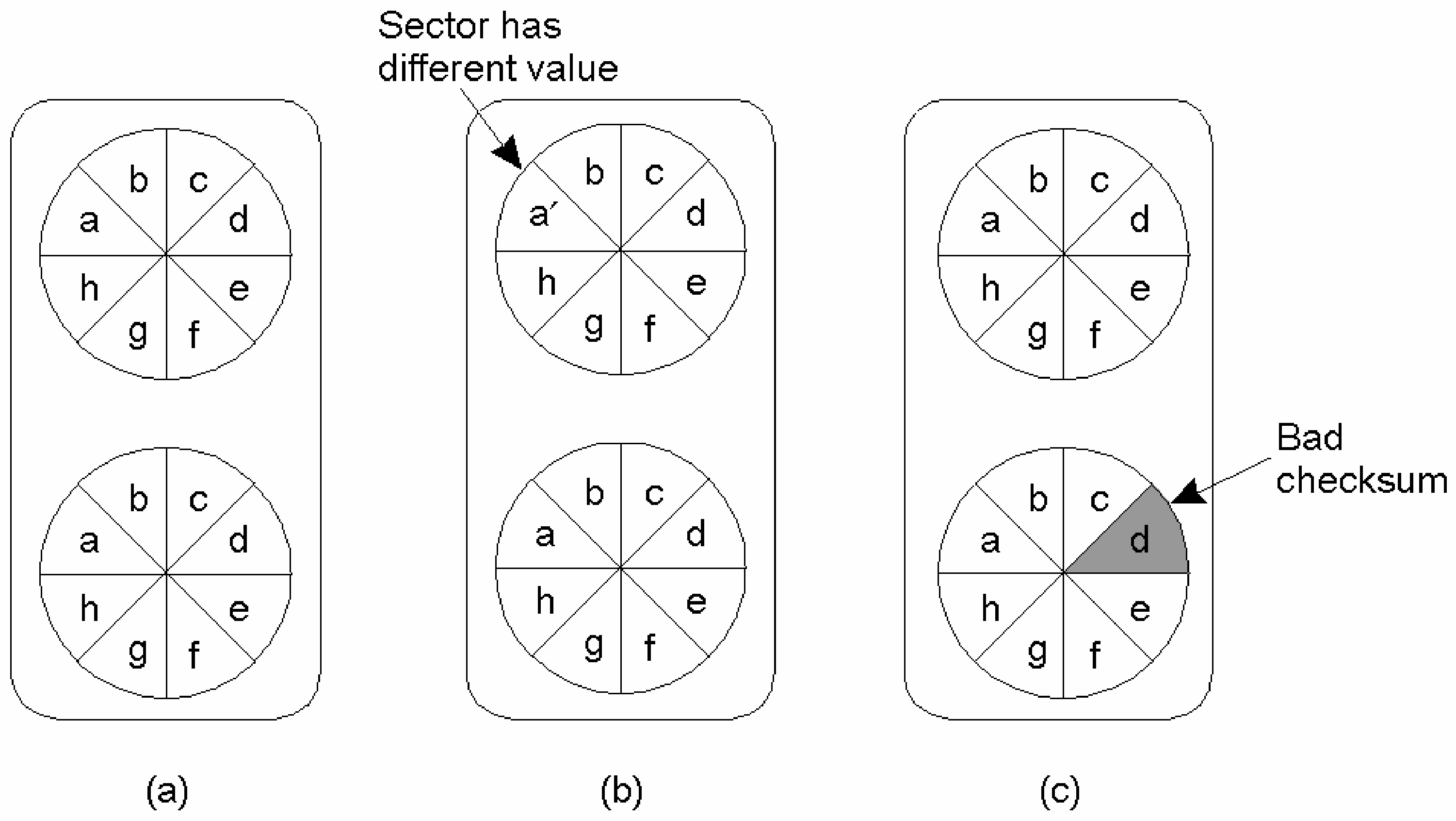
- **the coordinator sends a GLOBAL_COMMIT only after it has accepted an ack from all for its PREPARE_COMMIT message**
- **no process can be in state INIT while another one is in state PRECOMMIT**
- **if the majority are in state READY they can decide to abort**

8.6 Recovery

- fundamental to fault tolerance is recovery from an error
- recall: an error is that part of a system that may lead to a failure
- error recovery means to replace an erroneous state with an error-free state
- two forms of error recovery: backward recovery and forward recovery
- Backward Recovery
  - bring the system from its present erroneous state back into a previously correct state
  - for this, the system’s state must be recorded from time to time; each time a state is recorded, a checkpoint is said to be made
  - e.g., retransmitting lost or damaged packets in the implementation of reliable communication

  - most widely used, since it is a generally applicable method and can be integrated into the middleware layer of a distributed system
  - disadvantages:
    - checkpointing and restoring a process to its previous state are costly and performance bottlenecks
    - no guarantee can be given that the error will not recur, which may take an application into a loop of recovery
    - some actions may be irreversible; e.g., deleting a file, handing over cash to a customer
- Forward Recovery
  - bring the system from its present erroneous state to a correct new state from which it can continue to execute
  - it has to be known in advance which errors may occur so as to correct those errors
  - e.g., erasure correction (or simply error correction) where a lost or damaged packet is constructed from other successfully delivered packets

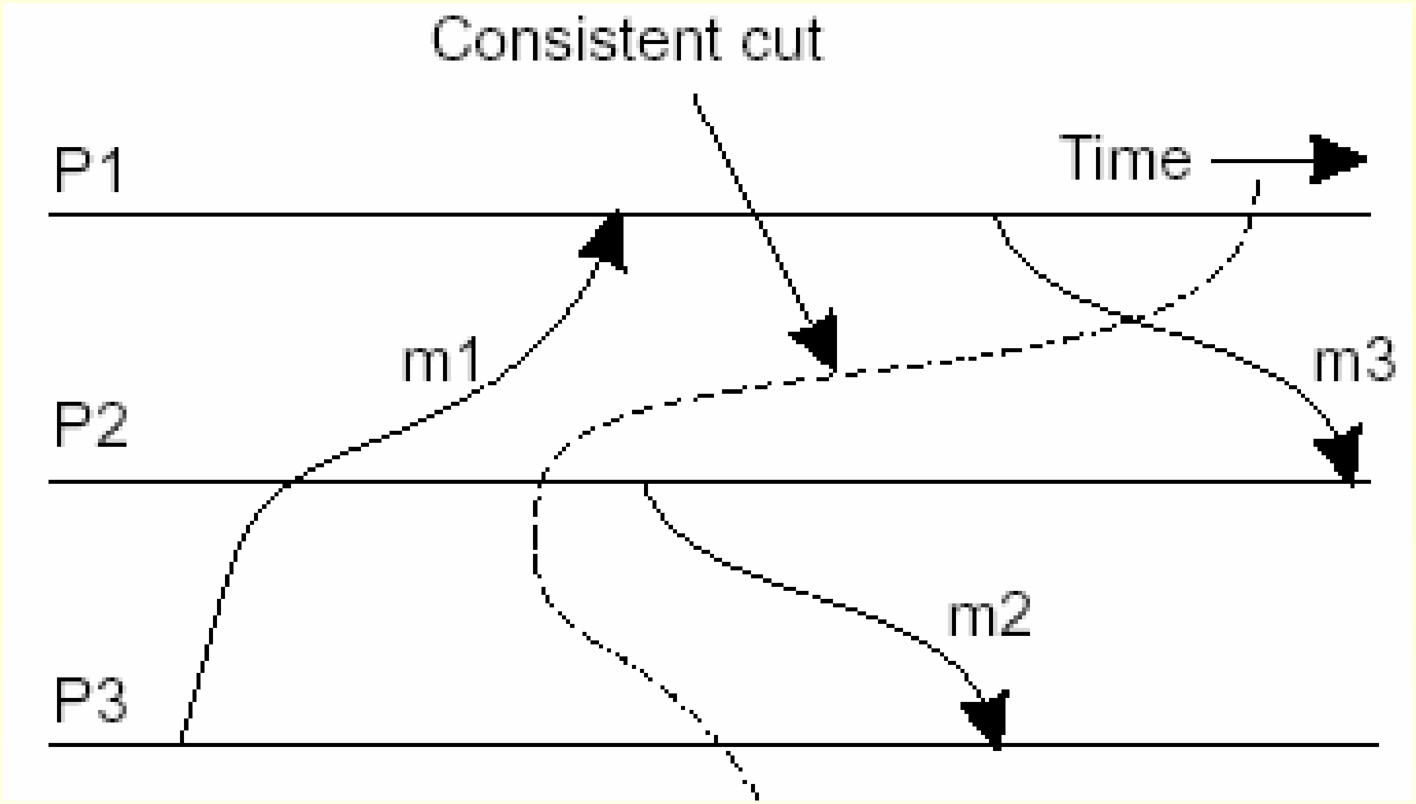
- Stable Storage
  - process states must be safely saved for recovery
  - how to design a storage system that can survive most problems
  - three possible storage mechanisms
    - RAM: power failure will wipe out the information
    - Disk: looses its contents if there is a head crash
    - Stable Storage
      - can survive most failures (other than those that are major calamities such as flood and earthquakes)
      - use a pair of ordinary disks
      - each block on drive 2 is an exact copy of the corresponding block on drive 1
      - when a block is updated, first the block on drive 1 is updated and verified, then the same block on drive 2 is done



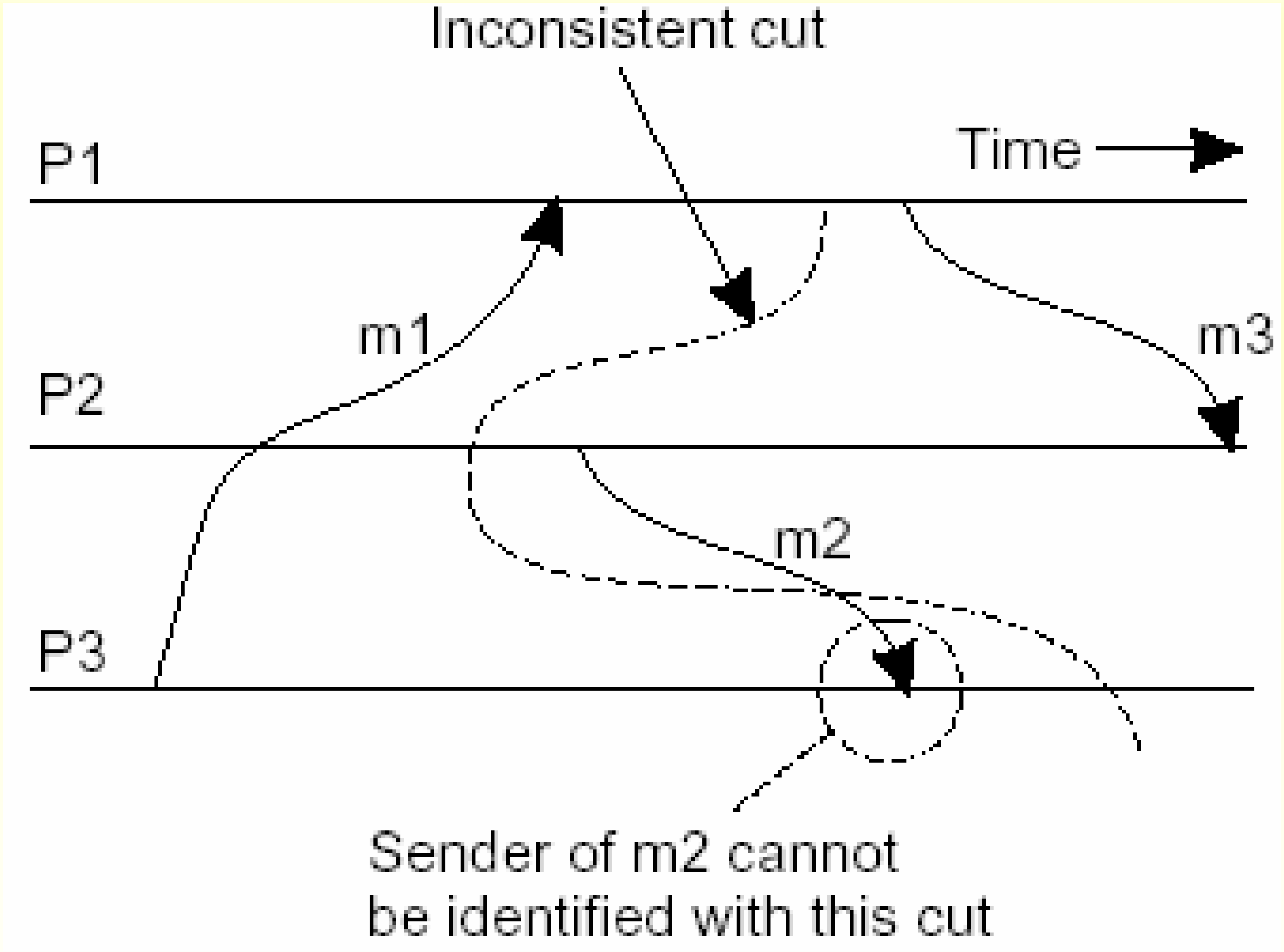
a) *Stable Storage*
b) *Crash after drive 1 is updated; upon recovery, the disk can be compared block for block; if corresponding blocks differ, the block on drive 1 is assumed to be correct (because drive 1 is always updated before drive 2)*
c) *Bad spot; copy the corresponding block from the other drive*

- Global State
  - the global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit (that have been sent but not delivered)
  - if local computations have stopped and there are no messages in transit, then there is no more progress
    - may be it is a deadlock, so something has to be done

      or
    - a distributed computation has correctly terminated
  - the global state can be recorded using a distributed snapshot
  - a distributed snapshot must reflect a consistent global state; if a message is recorded to be received, it has also to be recorded that it was sent by another process
  - a global state can be graphically represented by a cut

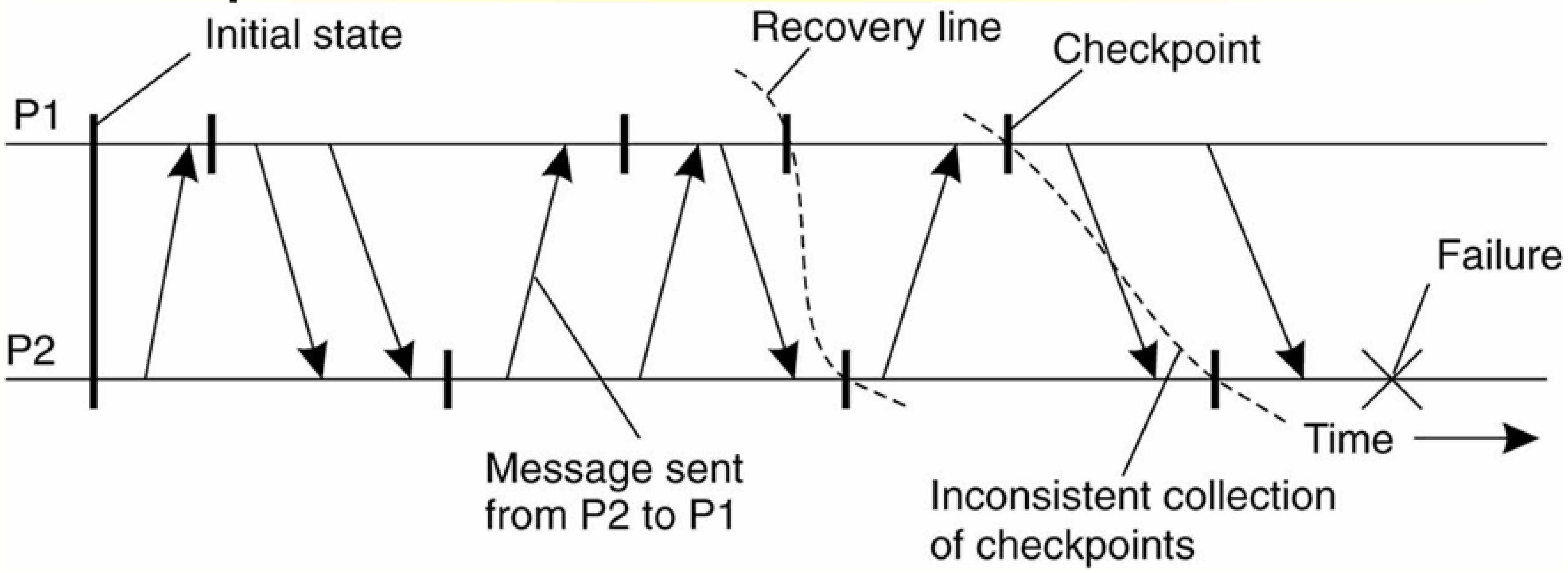


a consistent cut



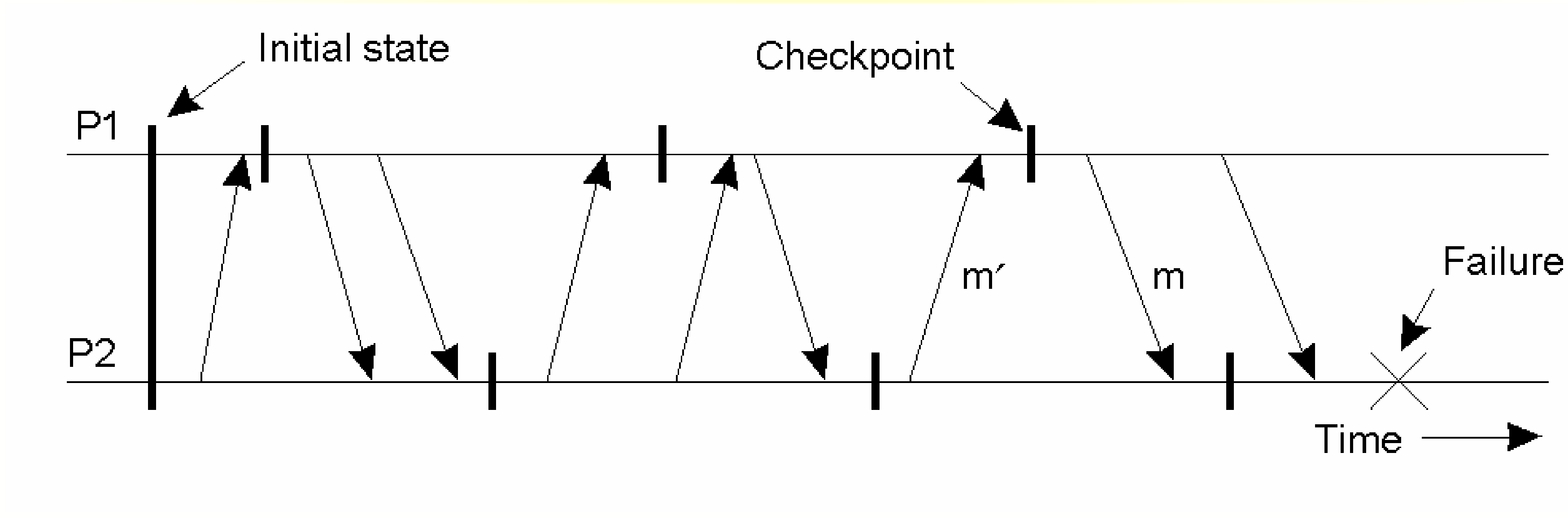
an inconsistent cut

- **Checkpointing**
  - **for backward recovery after a process or system failure requires that we construct a consistent global state (also called distributed snapshot) i.e., if a process has recorded the receipt of a message, then there should be another process that has recorded the sending of that message**
  - **recovery is done from the most recent distributed snapshot, also referred to as a recovery line or consistent cut; it corresponds to the most recent collection of checkpoints**


Initial state
Recovery line
Checkpoint
P1
P2
Failure
Time
Message sent from P2 to P1
Inconsistent collection of checkpoints

a recovery line

- **Independent Checkpointing**
  - **discovering a recovery line requires that each process should roll back to its most recently saved state; if the local states jointly do not form a distributed snapshot, further rolling back is necessary**
  - **this process of cascaded rollback may lead to what is called the domino effect, i.e., it may not be possible to find a recovery line except the initial states of processes**
  - **this happens if processes checkpoint independently**



the domino effect

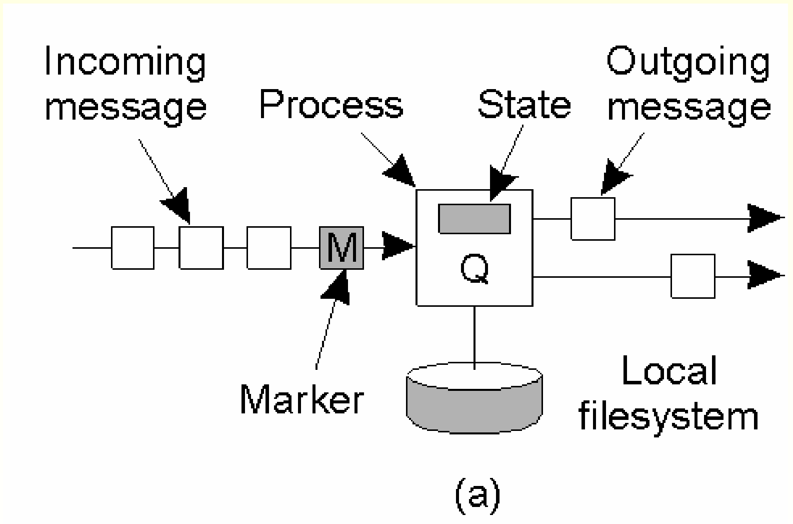
- **Coordinated Checkpointing**
  - **all processes synchronize to jointly write their state to local stable storage**
  - **the saved state is automatically globally consistent, so that cascaded rollback leading to the domino effect is avoided**
  - **but coordination requires global synchronization which may introduce performance problems**
  - **there are two ways to coordinate checkpointing**

1. **a two-phase blocking protocol can be used (centralised)**
   - **a coordinator multicasts a CHECKPOINT_REQUEST message to all processes**
   - **in receipt of such a message, each process takes a local snapshot, queues subsequent messages handed to it by the application, and acknowledges to the coordinator**
   - **after receiving acknowledgement from all processes, the coordinator then multicasts a CHECKPOINT_DONE message to allow (blocked) processes to continue**

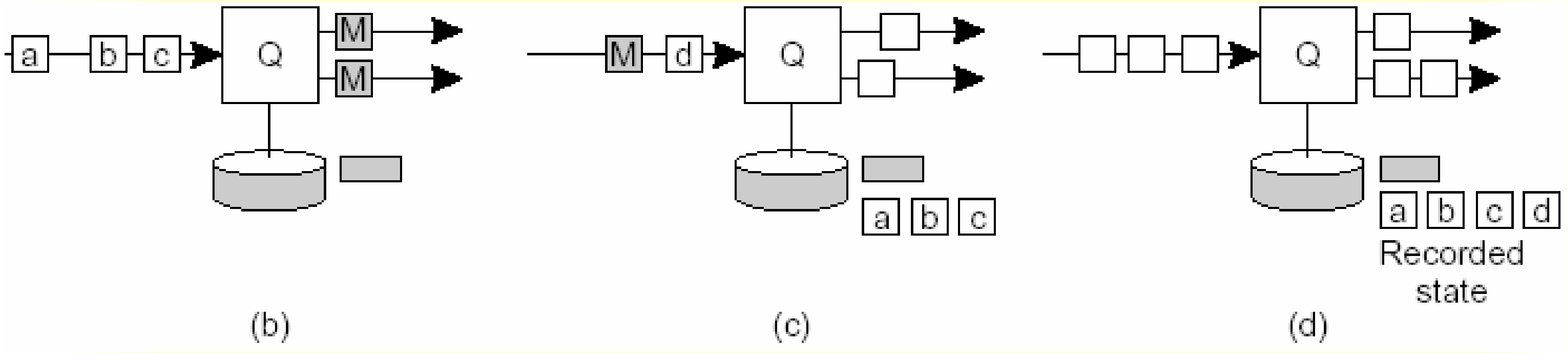
    - it leads to a globally consistent state, since no incoming messages will be recorded as part of the checkpoint

2. Distributed Snapshot
    - for simplicity, assume that the distributed system can be represented as a collection of processes connected to each other through unidirectional point-to-point communication channels; for example by first setting up TCP connections before any communication takes place
    - any process, say P, can initiate the algorithm
        - P starts by recording its own local state
        - then it sends a marker along each of its outgoing channels, indicating that the receiver should participate in recording the global state

- a process, say Q, receiving a marker through an incoming channel C
  - first records its local state (if it hasn’t done so) and sends a marker along each of its outgoing channels
  - if it has already recorded its state (and has passed the marker), the marker is an indicator that Q should record the state of the channel (the state is formed by the sequence of messages that have been received since the last time it recorded its own local state, and before it received the marker)



a) organization of a process and channels for a distributed snapshot



b) process Q receives a marker for the first time and records its local state

c) Q records all incoming message

d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel

- **a process Q is said to finish its part of the algorithm when it has received a marker along each of its incoming channels, and processed each one**
- **at that point, its recorded local state, as well as the state it recorded for each incoming channel, can be collected and sent, for example, to the process that initiated the snapshot**
- **the construction of several snapshots may be in progress at the same time; hence a marker is tagged with the identifier (possibly also a version number) of the initiating process**
- **a process P can finish the construction of the marker's snapshot, only after it has received that marker through each of its incoming channels**

- **Read about Message Logging and Recovery-Oriented Computing (Pages 369- 373)**