Unit-8
Fault Tolerance

- A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure.
- A partial failure may happen when one component in a distributed system fails.
- This failure may affect the proper operation of other components, while at the same time leaving yet other components totally unaffected.
- In contrast, a failure in non-distributed systems is often total in the sense that it affects all components, and may easily bring down the entire system.

- An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance.
- In particular, whenever a failure occurs, the distributed system should continue to operate in an acceptable way while repairs are being made, that is, it should tolerate faults and continue to operate to some extent even in their presence.

- **Fault tolerance** has been subject to much research in computer science.
- In this section, we start with presenting the basic concepts related to processing failures, followed by a discussion of failure models.
- The key technique for handling failures is redundancy, which is also discussed.

To understand the role of fault tolerance in distributed systems we first need to take a closer look at what it actually means for a distributed system to tolerate faults.
Being fault tolerant is strongly related to what are called dependable systems.
Dependability is a term that covers a number of useful requirements for distributed systems including the following (Kopetz and Verissimo, 1993):

1. *Availability*
2. *Reliability*
3. *Safety*
4. *Maintainability*

- **Availability** is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users. In other words, a highly available system is one that will most likely be working at a given instant in time.
- **Reliability** refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly-reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability.
- If a system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable. Similarly, a system that never crashes but is shut down for two weeks every August has high reliability but only 96 percent availability. The two are not the same.

- **Safety** refers to the situation that when a system temporarily fails to operate correctly, nothing catastrophic happens. For example, many process control systems, such as those used for controlling nuclear power plants or sending people into space, are required to provide a high degree of safety. If such control systems temporarily fail for only a very brief moment, the effects could be disastrous. Many examples from the past (and probably many more yet to come) show how hard it is to build safe systems.
- Finally, **maintainability** refers to how easy a failed system can be repaired. A - highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically..

- **A system is said to fail when it cannot meet its promises**.
- In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be (completely) provided.
- An error is a part of a system's state that may lead to a failure.
- For example, when transmitting packets across a network, it is to be expected that some packets have been damaged when they arrive at the receiver.
- Damaged in this context means that the receiver may incorrectly sense a bit value (e.g., reading a 1 instead of a 0), or may even be unable to detect that something has arrived.
- The cause of an **error is called a fault**.
- Clearly, finding out what caused an error is important. For example, a wrong or bad transmission medium may easily cause packets to be damaged.
- In this case, it is relatively easy to remove the fault.
- However, transmission errors may also be caused by bad weather conditions such as in wireless networks. Changing the weather to reduce or prevent errors is a bit trickier.

- Faults are generally classified **as transient, intermittent, or permanent.**
- **Transient faults** occur once and then disappear. If the operation is repeated, the fault goes away. A bird flying through the beam of a microwave transmitter may cause lost bits on some network (not to mention a roasted bird). If the transmission times out and is retried, it will probably work the second time.
- **An intermittent fault** occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. Typically, when the fault doctor shows up, the system works fine.
- A **permanent fault** is one that continues to exist until the faulty component is replaced. Burnt-out chips, software bugs, and disk head crashes are examples of permanent faults.

8.2 PROCESS RESILIENCE

- Now that the basic issues of fault tolerance have been discussed, let us concentrate on how fault tolerance can actually be achieved in distributed systems.
- The first topic we discuss is protection against process failures, which is achieved by replicating processes into groups.
- In the following pages, we consider the general design issues of process groups, and discuss what a fault-tolerant group actually is.
- Also, we look at how to reach agreement within a process group when one or more of its members cannot be trusted to give correct answers.

- **The key approach to tolerating a faulty process** is to organize several identical processes into a group. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it.
- In this way, if one process in a group fails, hopefully some other process can take over for it (Guerraoui and Schiper, 1997).
- Process groups may be dynamic. New groups can be created and old groups can be destroyed.
- A process can join a group or leave one during system operation.
- A process can be a member of several groups at the same time.
- Consequently, mechanisms are needed for managing groups and group membership.

- Groups are roughly analogous to social organizations.
- Alice might be a member of a book club, a tennis club, and an environmental organization.
- On a particular day, she might receive mailings (messages) announcing a new birthday cake cookbook from the book club, the annual Mother's Day tennis tournament from the tennis club, and the start of a campaign to save the Southern groundhog from the environmental organization.
- At any moment, she is free to leave any or all of these groups, and possibly join other groups.
- The purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction.
- Thus a process can send a message to a group of servers without having to know who they are or how many there are or where they are, which may change from one call to the next.

Flat Groups versus Hierarchical Groups

- An important distinction between different groups has to do with their internal structure.
- In some groups, all the processes are equal. No one is boss and all decisions are made collectively.
- In other groups, some kind of hierarchy exists.
- For example, one process is the coordinator and all the others are workers.
- In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there.
- More complex hierarchies are also possible, of course.
- These communication patterns are illustrated in Fig. 8-3.

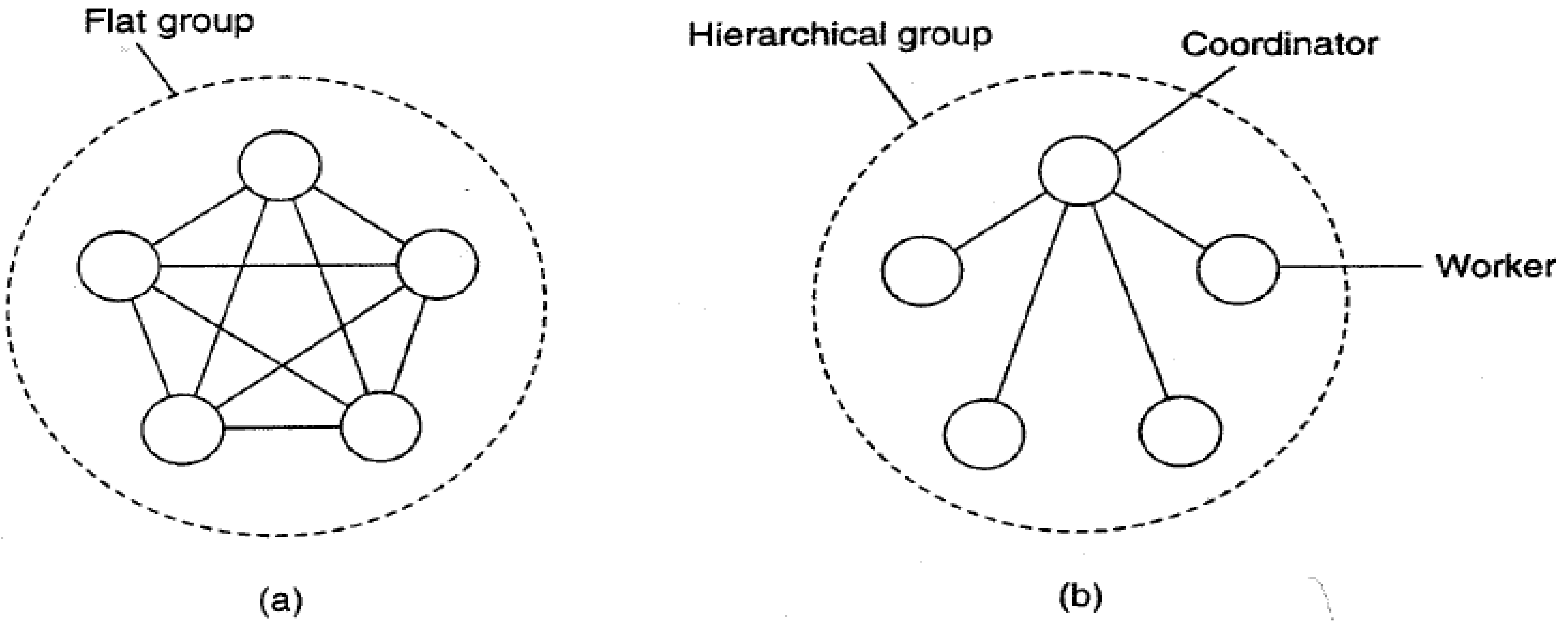

Figure 8-3. (a) Communication in a flat group. (b) Communication in a simple hierarchical group.

- Each of these organizations has its own advantages and disadvantages.
- The flat group is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue.
- A disadvantage is that decision making is more complicated.
- For example, to decide anything, a vote often has to be taken, incurring some delay and overhead.
- The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else.

8.3 RELIABLE CLIENT-SERVER COMMUNICATION

- In many cases, fault tolerance in distributed systems concentrates on faulty processes.
- However, we also need to consider communication failures. Most of the failure models discussed previously apply equally well to communication channels.
- In particular, a communication channel may exhibit crash, omission, timing, and arbitrary failures.
- In practice, when building reliable communication channels, the focus is on masking crash and omission failures.
- Arbitrary failures may occur in the form of duplicate messages, resulting from the fact that in a computer network messages may be buffered for a relatively long time, and are reinjected into the network after the original sender has already issued a retransmission

8.3.1 **Point-to-Point Communication**

- In many distributed systems, reliable point-to-point communication is established by making use of a reliable transport protocol, such as TCP.
- TCP masks omission failures, which occur in the form of lost messages, by using acknowledgments and retransmissions. Such failures are completely hidden from a TCP client.
- However, crash failures of connections are not masked. A crash failure may occur when (for whatever reason) a TCP connection is abruptly broken so that no more messages can be transmitted through the channel.
- In most cases, the client is informed that the channel has crashed by raising an exception. The only way to mask such failures is to let the distributed system attempt to automatically set up a new connection, by simply resending a connection request. The underlying assumptioriis that the other side is still, or again, responsive to such requests.

8.3.2 **RPC Semantics in the Presence of Failures**

- Let us now take a closer look at client-server communication when using high-level communication facilities such as Remote Procedure Calls (RPCs).
- The goal of RPC is to hide communication by making remote procedure calls look just like local ones.
- With a few exceptions, so far we have come fairly close. Indeed, as long as both client and server are functioning perfectly, RPC does its job well.
- The problem comes about when errors occur. It is then that the differences between local and remote calls are not always easy to mask.

To structure our discussion, let us distinguish between five different classes of failures that can occur in RPC systems, as follows:

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

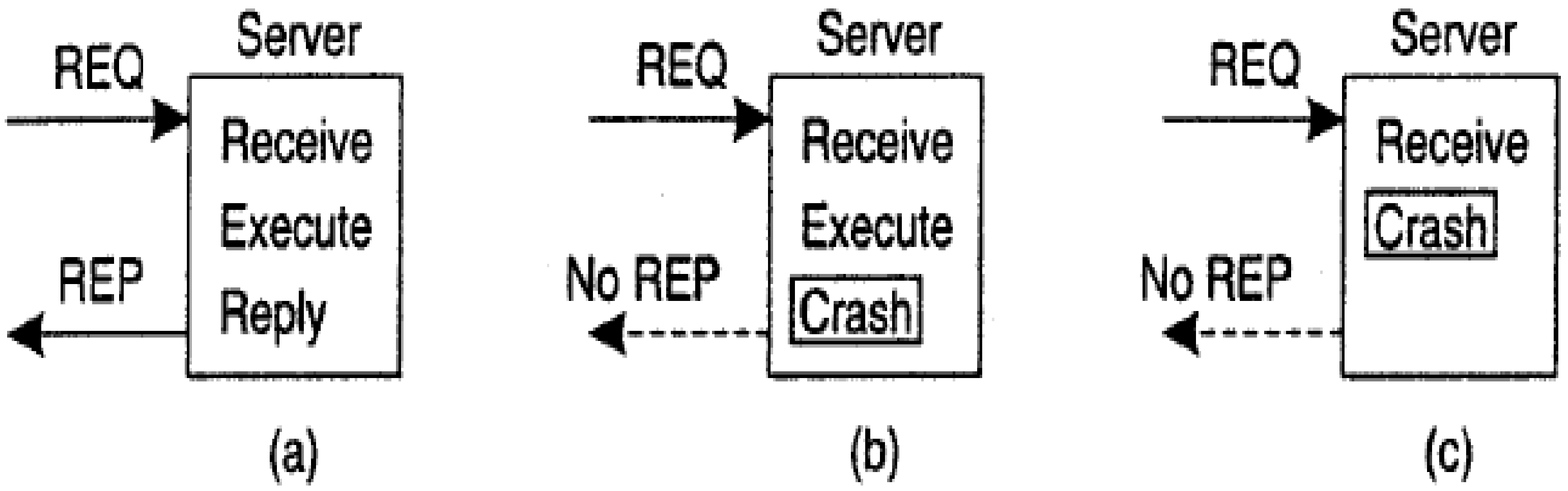

Figure 8-7. A server in client-server communication. (a) The normal case. (b) Crash after execution. (c) Crash before execution.

8.4 RELIABLE GROUP COMMUNICATION

- Considering how important process resilience by replication is, it is not surprising that reliable multicast services are important as well.
- Such services guarantee that messages are delivered to all members in a process group.
- Unfortunately, reliable multicasting turns out to be surprisingly tricky. In this section, we take a closer look at the issues involved in reliably delivering messages to a process group.

8.4.1 **Basic Reliable-Multicasting Schemes**

- Although most transport layers offer reliable point-to-point channels, they rarely offer reliable communication to a collection of processes.
- The best they can offer is to let each process set up a point-to-point connection to each other process it wants to communicate with.
- Obviously, such an organization is not very efficient as it may waste network bandwidth. Nevertheless, if the number of processes is small, achieving reliability through multiple reliable point-to-point channels is a simple and often straightforward solution.
- To go beyond this simple case, we need to define precisely what reliable multicasting is.
- Intuitively, it means that a message that is sent to a process group should be delivered to each member of that group.
- However, what happens if during communication a process joins the group? Should that process also receive the message? Likewise, we should also determine what happens if a (sending) process crashes during communication.

- To cover such situations, a distinction should be made between reliable communication in the presence of faulty processes, and reliable communication when processes are assumed to operate correctly.
- In the first case, multicasting is considered to be reliable when it can be guaranteed that all non-faulty group members receive the message.
- The tricky part is that agreement should be reached on what the group actually looks like before a message can be delivered, in addition to various ordering constraints. We return to these matters when we discuss atomic multicasts below.
- The situation becomes simpler if we assume agreement exists on who is a member of the group and who is not.
- In particular, 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.

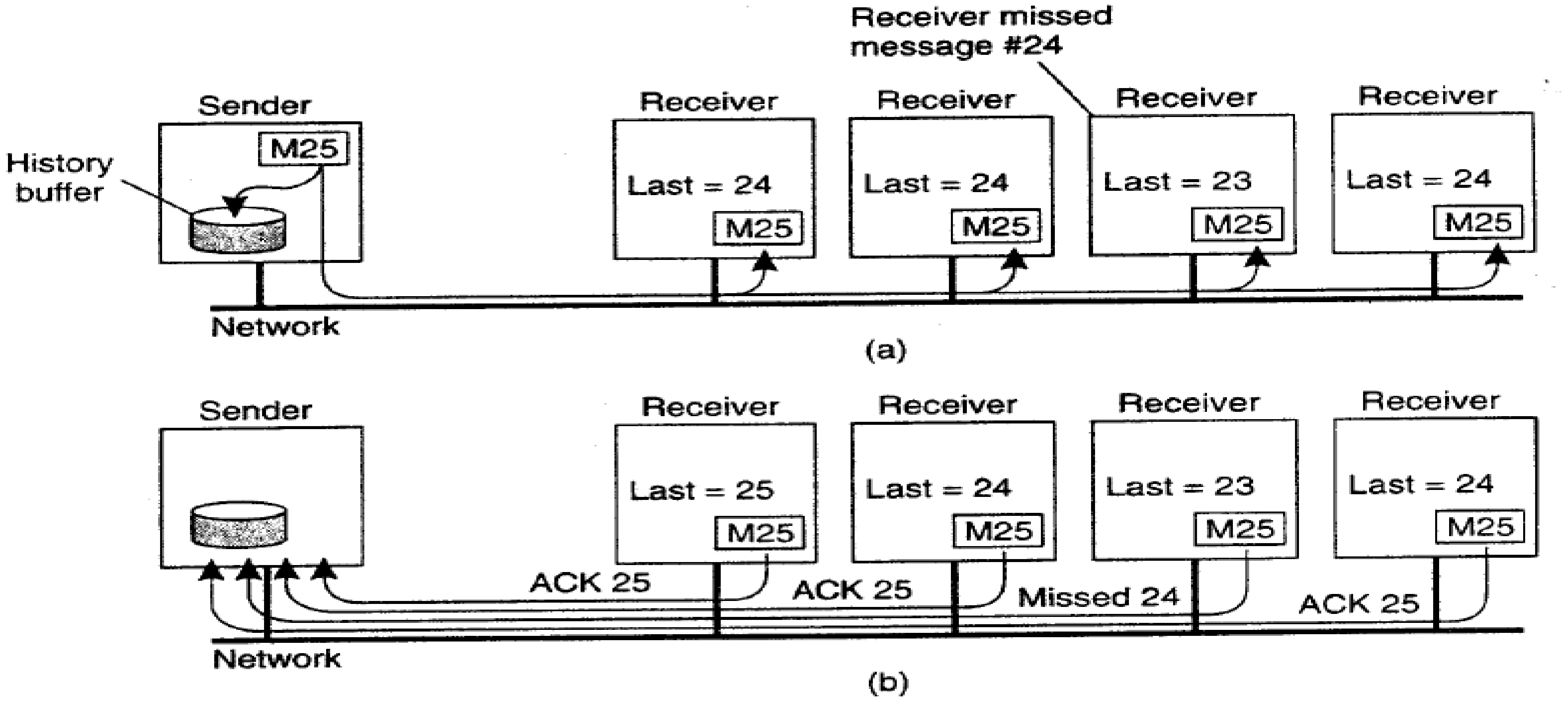

Figure 8-9. A simple solution to reliable multicasting when all receivers are known and are assumed not to fail. (a) Message transmission. (b) Reporting feedback..

Assume that the underlying communication system offers only unreliable multicasting, meaning that a multicast message may be lost part way and delivered to some, but not all, of the intended receivers.

- A simple solution is shown in Fig. 8-9. The sending process assigns a sequence number to each message it multicasts.
- We assume that messages are received in the order they are sent. In this way, it is easy for a receiver to detect it is missing a message.
- Each multicast message is stored locally in a history buffer at the sender. Assuming the receivers are known to the sender, the sender simply keeps the message in its history buffer until each receiver has returned an acknowledgment.
- If a receiver detects it is missing a message, it may return a negative acknowledgment, requesting the sender for a retransmission.
- Alternatively, the sender may automatically retransmit the message when it has not received all acknowledgments within a certain time.

8.4.2 Scalability in Reliable Multicasting

- The main problem with the reliable multicast scheme just described is that it cannot support large numbers of receivers.
- If there are *N* receivers, the sender must be prepared to accept at least *N* acknowledgments. With many receivers, the sender may be swamped with such feedback messages, which is also referred to as a feedback implosion.
- In addition, we may also need to take into account that the receivers are spread across a wide-area network.
- One solution to this problem is not to have receivers acknowledge the receipt of a message.
- Instead, a receiver returns a feedback message only to inform the sender it is missing a message.
- Returning only such negative acknowledgments can be shown to generally scale better but no hard guarantees can be given that feedback implosions will never happen.

- Another problem with returning only negative acknowledgments is that the sender will, in theory, be forced to keep a message in its history buffer forever.
- Because the sender can never know if a message has been correctly delivered to all receivers, it should always be prepared for a receiver requesting the retransmission of an old message.
- In practice, the sender will remove a message from its history buffer after some time has elapsed to prevent the buffer from overflowing.
- However, removing a message is done at the risk of a request for a retransmission not being honored.

8.5 DISTRIBUTED COMMIT

- The atomic multicasting problem discussed in the previous section is an example of a more general problem, known as **distributed commit.**
- The distributed commit problem involves having an operation being performed by each member of a process group, or none at all.
- In the case of reliable multicasting, the operation is the delivery of a message.
- With distributed transactions, the operation may be the commit of a transaction at a single site that takes part in the transaction.

- Distributed commit is often established by means of a coordinator. In a simple scheme, this coordinator tells all other processes that are also involved, called participants, whether or not to (locally) perform the operation in question.
- This scheme is referred to as a **one-phase commit protocol.** It has the obvious drawback that if one of the participants cannot actually perform the operation, there is no way to tell the coordinator.
- For example, in the case of distributed transactions, a local commit may not be possible because this would violate concurrency control constraints.
- In practice, more sophisticated schemes are needed, the most common one being the two-phase commit protocol, which is discussed in detail below.
- The main drawback of this protocol is that it cannot efficiently handle the failure of the coordinator. To that end, a three-phase protocol has been developed, which we also discuss.

8.5.1 Two-Phase Commit

- The original **two-phase commit protocol** (2PC) is due to Gray (1978) Without loss of generality, consider a distributed transaction involving the participation of a number of processes each running on a different machine.
- Assuming that no failures occur, the protocol consists of the following two phases, each consisting of two steps [see also Bernstein et al. (1987)]:

1. The coordinator sends a *VOTE-.REQUEST* message to all participants.
2. When a participant receives a *VOTE-.REQUEST* message, it returns either a *VOTE_COMMIT* message to the coordinator telling the coordinator that it is prepared to locally commit its part of the transaction, or otherwise a *VOTE-ABORT* message.

3. The coordinator collects all votes from the participants. If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a *GLOBAL_COMMIT* message to all participants. However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a *GLOBAL..ABORT* message.
4. Each participant that voted for a commit waits for the final reaction by the coordinator. If a participant receives a *GLOBAL_COMMIT* message, it locally commits the transaction. Otherwise, when receiving a *GLOBAL..ABORT* message, the transaction is locally aborted as well.

The first phase is the voting phase, and consists of steps 1 and 2. The second phase is the decision phase, and consists of steps 3 and 4. These four steps are shown as finite state diagrams in Fig. 8-18.

The first phase is the voting phase, and consists of steps 1 and 2. The second phase is the decision phase, and consists of steps 3 and 4. These four steps are shown as finite state diagrams in Fig. 8-18.

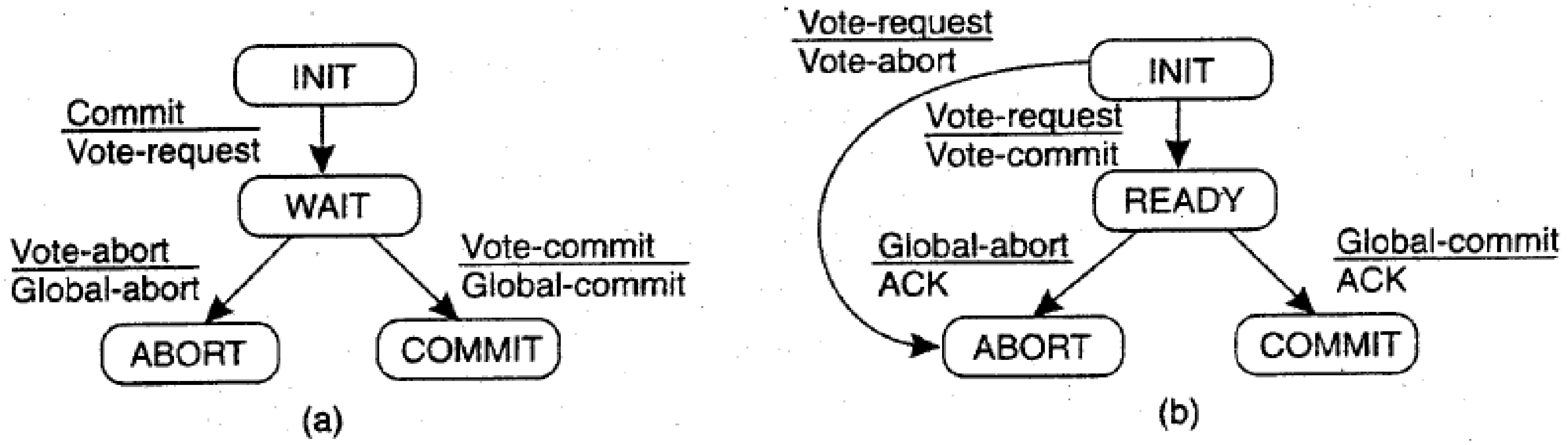
INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Global-commit
ABORT
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Global-commit / ACK
ABORT
COMMIT
(b)

Figure 8~18. (a) The finite state machine for the coordinator in ;2PC. (b) The finite state machine for a participant..

- Several problems arise when this basic 2PC protocol is used in a system where failures occur.
- First, note that the coordinator as well as the participants have states in which they block waiting for incoming messages.
- Consequently, the protocol can easily fail when a process crashes for other processes may be indefinitely waiting for a message from that process. For this reason, timeout mechanism are used.

8.5.2 **Three-Phase Commit**

- A problem with the two-phase commit protocol is that when the coordinator has crashed, participants may not be able to reach a final decision.
- Consequently, participants may need to remain blocked until the coordinator recovers. Skeen (1981) developed a variant of 2PC, called the three-phase commit protocol (3PC), that avoids blocking processes in the presence of fail-stop crashes.
- Although 3PC is widely referred to in the literature, it is not applied often in practice as the conditions under which 2PC blocks rarely occur.
- We discuss the protocol, as it provides further insight into solving fault-tolerance problems in distributed systems.

- Like 2PC, 3PC is also formulated in terms of a coordinator and a number of participants. Their respective finite state machines are shown in Fig. 8-22.
- The essence of the protocol is that the states of the coordinator and each participant satisfy the following two conditions:

1. There is no single state from which it is possible to make a transition directly to either a *COMMIT* or an *ABORT* state.
2. 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.

It can be shown that these two conditions are necessary and sufficient for a commit protocol to be nonblocking (Skeen and Stonebraker, 1983).

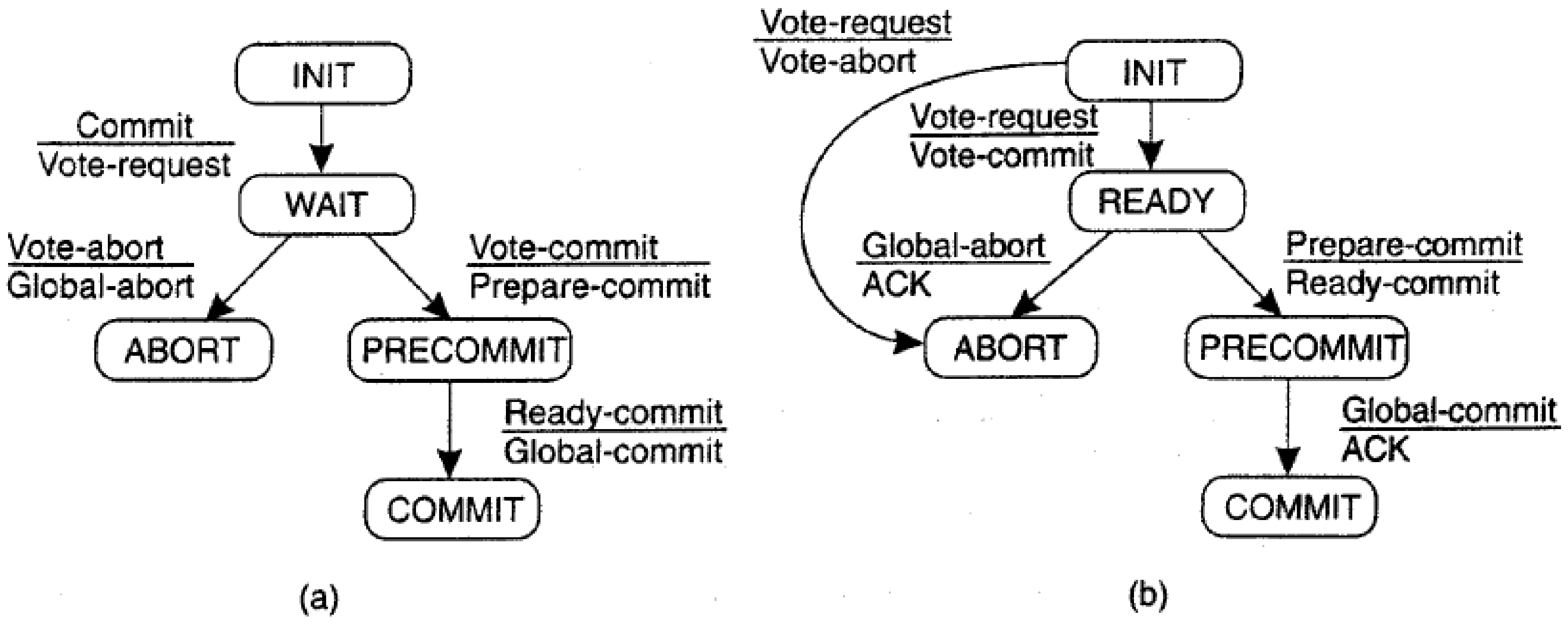

Figure 8-22. (a) The finite state machine for the coordinator in 3PC. (b) The finite state machine for a participant.

8.6 RECOVERY

- So far, we have mainly concentrated on algorithms that allow us to tolerate faults.
- However, once a failure has occurred, it is essential that the process where the failure happened can recover to a correct state.
- In what follows, we first concentrate on what it actually means to recover to a correct state, and subsequently when and how the state of a distributed system can be recorded and recovered to, means of check pointing and message logging.

8.6.1 Introduction

- Fundamental to fault tolerance is the recovery from an error. Recall that an error is that part of a system that may lead to a failure.
- The whole idea of error recovery is to replace an erroneous state with an error-free state. There are essentially two forms of error recovery.
- **In backward recovery**, the main issue is to bring the system from its present erroneous state back into a previously correct state.
- To do so, it will be necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong.
- Each time (part of) the system's present state is recorded, a checkpoint is said to be made.

- Another form of error recovery is **forward recovery.** In this case, when the system has entered an erroneous state, instead of moving back to a previous, check pointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute.
- The main problem with forward error recovery mechanisms is that it has to be known in advance which errors may occur.
- Only in that case is it possible to correct those errors and move to a new state.
- **The distinction between backward and forward error recovery** is easily explained when considering the implementation of reliable communication.
- The common approach to recover from a lost packet is to let the sender retransmit that packet.
- In effect, packet retransmission establishes that we attempt to go back to a previous, correct state, namely the one in which the packet that was lost is being sent.
- Reliable communication through packet retransmission is therefore an example of applying backward error recovery techniques.

- An alternative approach is to use a method known as **erasure correction**.
- In this approach. a missing packet is constructed from other, successfully delivered packets.
- For example, in an *(n,k)* block erasure code, a set of *k source packets* is encoded into a set of *n encoded packets,* such that *any* set of *k* encoded packets is enough to reconstruct the original *k* source packets.
- Typical values are *k* =16' or *k=32,* and *k<11~2k* [see, for example, Rizzo (1997)].
- If not enough packets have yet been delivered, the sender will have to continue transmitting packets until a previously lost packet can be constructed.
- Erasure correction is a typical example of a forward error recovery approach

- By a large, backward error recovery techniques are widely applied as a general mechanism for recovering from failures in distributed systems.
- The major benefit of backward error recovery is that it is a generally applicable method independent of any specific system or process.
- In other words, it can be integrated into (the middleware layer) of a distributed system as a general-purpose service.

- Another important distinction between **check pointing and schemes** that additionally use logs follows.
- In a system where only check pointing is used, processes will be restored to a check pointed state.
- From there on, their behavior may be different than it was before the failure occurred.
- For example, because communication times are not deterministic, messages may now be delivered in a different order, in tum leading to different reactions by the receivers.
- However, if message logging takes place, an actual replay of the events that happened since the last checkpoint takes place.
- Such a replay makes it easier to interact with the outside world,

Stable Storage

- To be able to recover to a previous state, it is necessary that information needed to enable recovery is safely stored.
- Safely in this context means that recovery information survives process crashes and site failures, but possibly also various storage media failures.
- Stable storage plays an important role when it comes to recovery in distributed systems. We discuss it briefly here.
- **Storage comes in three categories.**
- First there is ordinary RAM memory, which is wiped out when the power fails or a machine crashes.
- Next there is disk storage, which survives CPU failures but which can be lost in disk head crashes.

Finally, there is also stable storage, which is designed to survive anything except major calamities such as floods and earthquakes.
Stable storage can be implemented with a pair of ordinary disks, as shown in Fig. 8-23(a).
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.

Suppose that the system crashes after drive 1 is updated but before the update on drive 2, as shown in Fig. 8-23(b).

Upon recovery, the disk can be compared block for block.

Whenever two corresponding blocks differ, it can be assumed that drive 1 is the correct one (because drive 1 is always updated before drive 2), so the new block is copied from drive 1 to drive 2. When the recovery process is complete, both drives will again be identical.

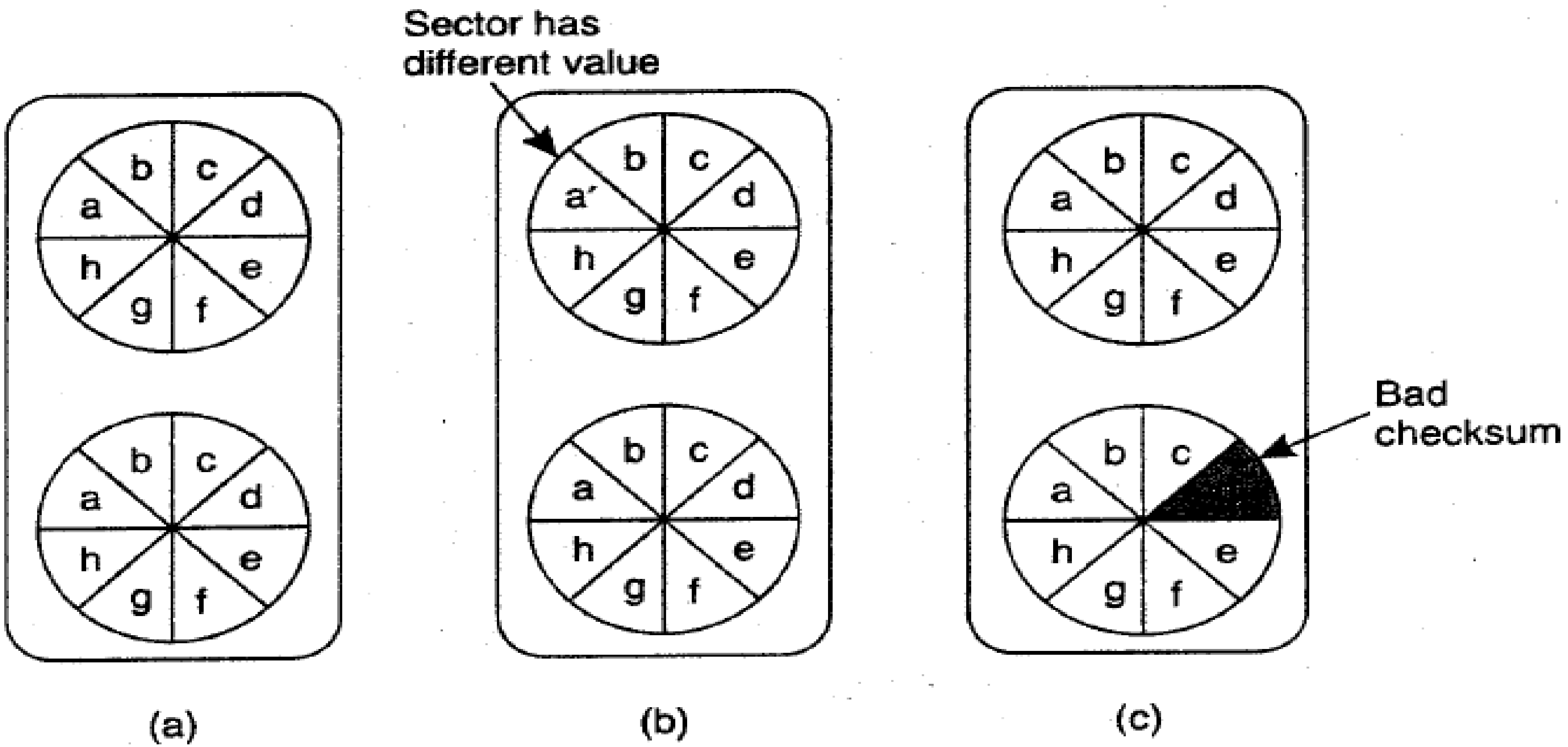

Figure 8-23. (a) Stable storage. (b) Crash after drive I is updated. (c) Bad spot.

8.6.2 Checkpointing

- In a fault-tolerant distributed system, backward error recovery requires that the system regularly saves its state onto stable storage.
- In particular, we need to record a consistent global state, also called a **distributed snapshot.**
- In a distributed snapshot, if a process *P* has recorded the receipt of a message, then there should also be a process *Q* that has recorded the sending of that message. After all, it must have come from somewhere.

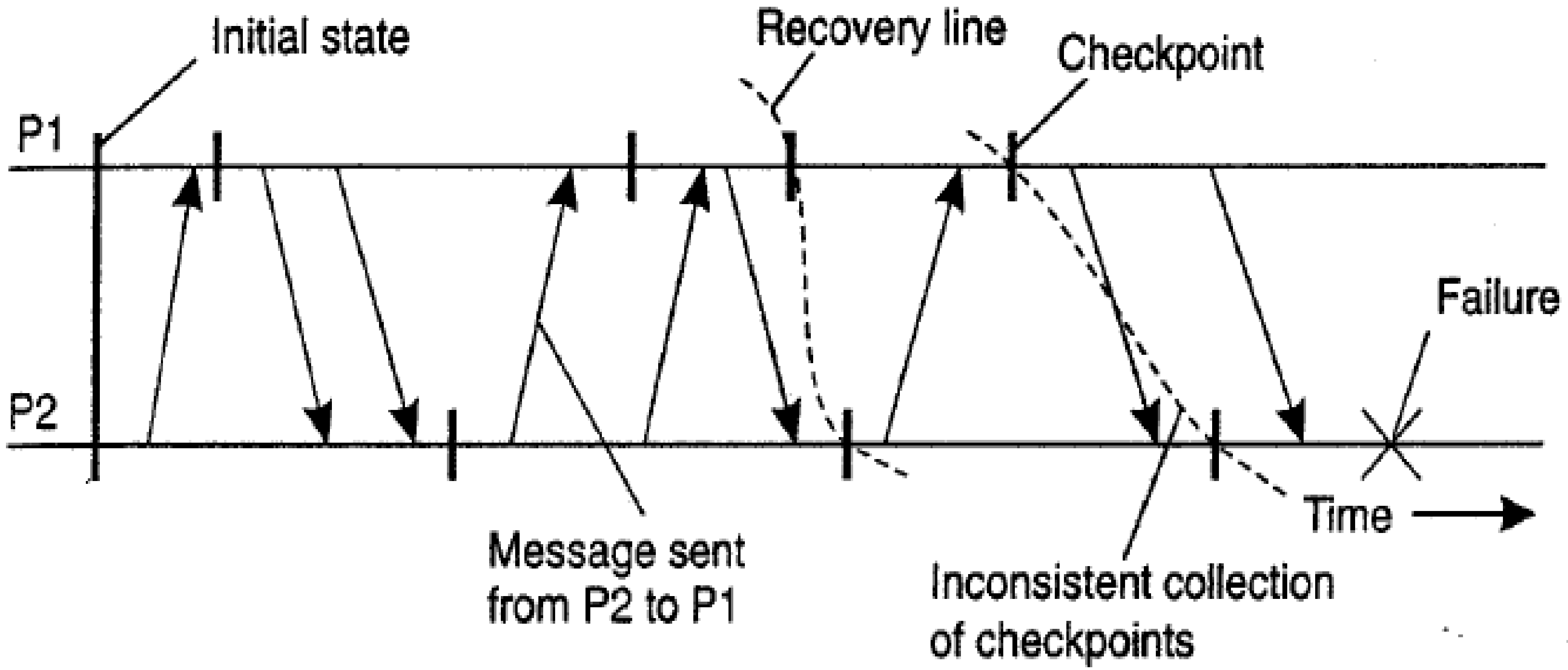
Initial state
Recovery line
Checkpoint
P1
P2
Failure
Message sent from P2 to P1
Time
Inconsistent collection of checkpoints

Figure 8-24. A recovery line.

- In backward error recovery schemes, each process saves its state from time to time to a locally-available stable storage.
- To recover after a process or system failure requires that we construct a consistent global state from these local states.
- In particular, it is best to recover to the *most recent* distributed snapshot, also referred to as a recovery line.
- In other words, a recovery line corresponds to the most recent consistent collection of checkpoints, as shown in Fig. 8-24.

8.6.3 Message Logging

- Considering that check pointing is an expensive operation, especially concerning the operations involved in writing state to stable storage, techniques have been sought to reduce the number of checkpoints, but still enable recovery.
- An important technique in distributed systems is logging messages.
- The basic idea underlying message logging is that 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.
- Instead, a checkpointed state is taken as a starting point, and all messages that have been sent since are simply retransmitted and handled accordingly.

- This approach works fine under the assumption of what is called a piecewise deterministic model. In such a model, the execution of each process is assumed to take place as a series of intervals in which events take place.
- These events are the same as those discussed in the context of Lamport's happened-before relationship in Chap. 6.
- For example, an event may be the execution of an instruction, the sending of a message, and so on.
- Each interval in the piecewise deterministic model is assumed to start with a nondeterministic event, such as the receipt of a message.
- However, from that moment on, the execution of the process is completely deterministic.
- An interval ends with the last event before a nondeterministic event occurs.

8.6.4 Recovery-Oriented Computing

- A related way of handling recovery is essentially to start over again.
- The underlying principle toward this way of masking failures is that it may be much cheaper to optimize for recovery, then it is aiming for systems that are free from failures for a long time.
- This approach is also referred to as recovery-oriented computing (Candea et aI., 2004a).
- There are different flavors of recovery-oriented computing. One flavor is to simply reboot (part of a system) and has been explored to restart Internet servers (Candea et al., 2004b, 2006).
- In order to be able reboot only a part of the system, it is crucial the fault is properly localized.
- At that point, rebooting simply means deleting all instances of the identified components. along with the threads operating on them, and (often) to just restart the associated requests.
- Note that fault localization itself may be a nontrivial exercise (Steinder and Sethi. 2004).

- Another flavor of recovery-oriented computing is to apply checkpointing and recovery techniques, but to continue execution in a changed environment.
- The basic idea here is that many failures can be simply avoided if programs are given some more buffer space, memory is zeroed before allocated, changing the ordering of message delivery (as long as this does not affect semantics), and so on (Qin et aI., 2005).
- The key idea is to tackle software failures (whereas many of the techniques discussed so far are aimed at, or are based on hardware failures).
- Because software execution is highly deterministic, changing an execution environment may save the day, but, of course, without repairing anything.