Distributed Systems
Delhi Technological University
Fault Tolerance

Divyashikha Sethia
divyashikha@dce.edu

Objectives

•General background on fault tolerance

• Process Resilience – incorporates one or more processes can fail without seriously disturbing the rest of the system.

• Reliable multicasting - – message transmission to a collection of processes is guaranteed to succeed and is used to keep processes synchronized

•Atomicity - is necessary to guarantee that every operation in a transaction is carried out or none of them are

Fault Tolerance Basic Concepts

- Fault tolerant strongly related to dependable systems
- Dependability implies:
  - Availability
  - Reliability
  - Safety
  - Maintainability

Dependability

Availability –

- system is ready to be used immediately.
- highly available system is one that will most likely be working at a given instant in time

Reliability -

- System can run continuously without failure.
- Reliability is in terms of time interval instead of instant in time as for availability
- highly-reliable system will continue to work without interruption during a relatively long period of time

Comparison:

- system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable
- system that never crashes but is shut down for two weeks every August has high reliability but only 96 percent availability.

Dependability

Safety:

- when a system temporarily fails to operate correctly, nothing catastrophic happens
- nuclear power plants if temporarily fail for only a very brief moment, the effects could be disastrous

Maintainability:

- how easy a failed system can be repaired
- highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically

Fault Tolerance Basic Concepts
Types of Faults

-**Transient**: occur once and then disappear. If the operation is repeated, the fault goes away.

- **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

- **Permanent**: continues to exist until the faulty component is replaced. Burnt-out chips, software bugs, and disk head crashes are examples

Fault Tolerance Basic Concepts
Failure Models

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

Basic Failure Models

•**crash failure:**
- server prematurely halts, but was working correctly until it stopped.
-operating system that comes to a grinding halt, and for which there is only one solution: reboot it.

• **omission failure**: server fails to respond to a request

•**receive omission failure**:
- Server never got request. connection between a client and a server has been correctly established, but that there was no thread listening to
- Incoming requests will not affect current state of server, since it is unaware of message sent.

•**send omission failure**:
- when server has done its work, but fails sending response due send buffer overflowing
- server must reissue request to client

- **Timing failures**: response lies outside a specified real-time interval.
  - providing data too soon may cause trouble for a recipient if there is not enough buffer space to hold all the incoming data
  - server responds too late, in which case a *performance* failure is said to occur

- **Response failure:** by which the server's response is simply incorrect.
  - value failure, a server simply provides the wrong reply to a request eg: search engine returns Web pages not related to any of search terms
  - state transition failure: server reacts unexpectedly to an incoming request.
  - Eg: if a server receives a message it cannot recognize, a state transition failure happens if no measures have been taken to handle such messages

- **Arbitrary failures**: a server is producing output it should never have produced, but which cannot be detected as being incorrect
- eg: Worse yet a faulty server may even be maliciously working together with other servers to produce intentionally wrong answers.

Failure Masking by Redundancy

Hide failure occurrences from other processes:

i) **Information redundancy:** extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line

ii) **Time redundancy:** an action is performed, and then if need be, it is performed again.
- transaction aborts, it can be redone with no harm.
- are helpful when the faults are transient or intermittent.

iii) **Physical redundancy**: extra equipment or processes are added to make it possible for system tolerate the loss or malfunctioning of some components.
- software based: extra processes can be added to the system so that if a small number of them crash, the system can still function correctly

Process Resilience

Protection against process failures - achieved by replicating processes into groups.

- How to design fault-tolerant groups?
- How to reach an agreement within a group when some members cannot be trusted to give correct answers?

Process Resilience..

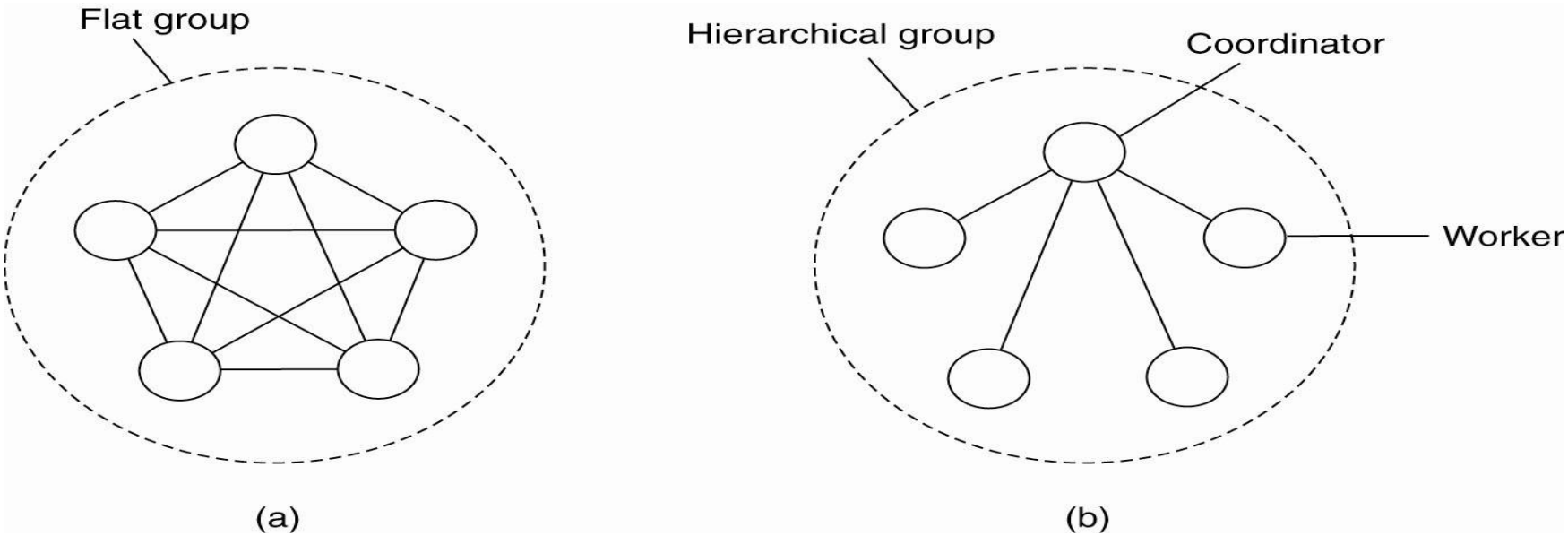
-key approach to tolerating a faulty process is to organize several identical processes into a group.

-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

- 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

Flat Groups versus Hierarchical Groups



(a) Communication in a flat group.

- all processes are equal and decisions are made collectively

(b) Communication in a simple hierarchical group.

- One process is the coordinator and all others are workers.
- Work request generated is sent to the coordinator.
- Coordinator decides which worker is best suited and forwards it request

Flat Groups versus Hierarchical Groups

	Flat Group	Hierarchical
Advantage	Symmetrical and has no single point of failure	Fast decision making
Disadvantage	Decision making is more complicated	Loss of coordinator brings entire group to a grinding halt

Group Membership

•Method is needed for creating and deleting groups and for allowing processes to join and leave groups

•Possibility:
i) Group server to which all these requests can be sent:
- maintain a complete data base of all the groups and their membership
- Advantage: Easy implementation
- Disadvantage: centralized techniques: a single point of failure

•ii) Manage group membership in a distributed way
- (reliable) multicasting is available, an outsider can send a message to all group members announcing its wish to join the group.
-to leave a group, a member just sends a goodbye message to everyone.
-- member crashes – other members must detect it and remove it from the group

Group communication

- Processes part of the group must receive all communication meant for the group when it is part of it

- Only leaving group process must not be part of the communication

Failure Masking and Replication

- Process groups are part of the solution for building fault-tolerant systems
- Group of identical processes allows to mask one or more faulty processes in the group.
- - can replicate processes and organize them into a group to replace a single (vulnerable) process with a (fault tolerant) group.
- Two ways to approach such replication:

i) ***Primary-backup protocol***. A primary coordinates all write operations. If it fails, then the others hold an election to replace the primary

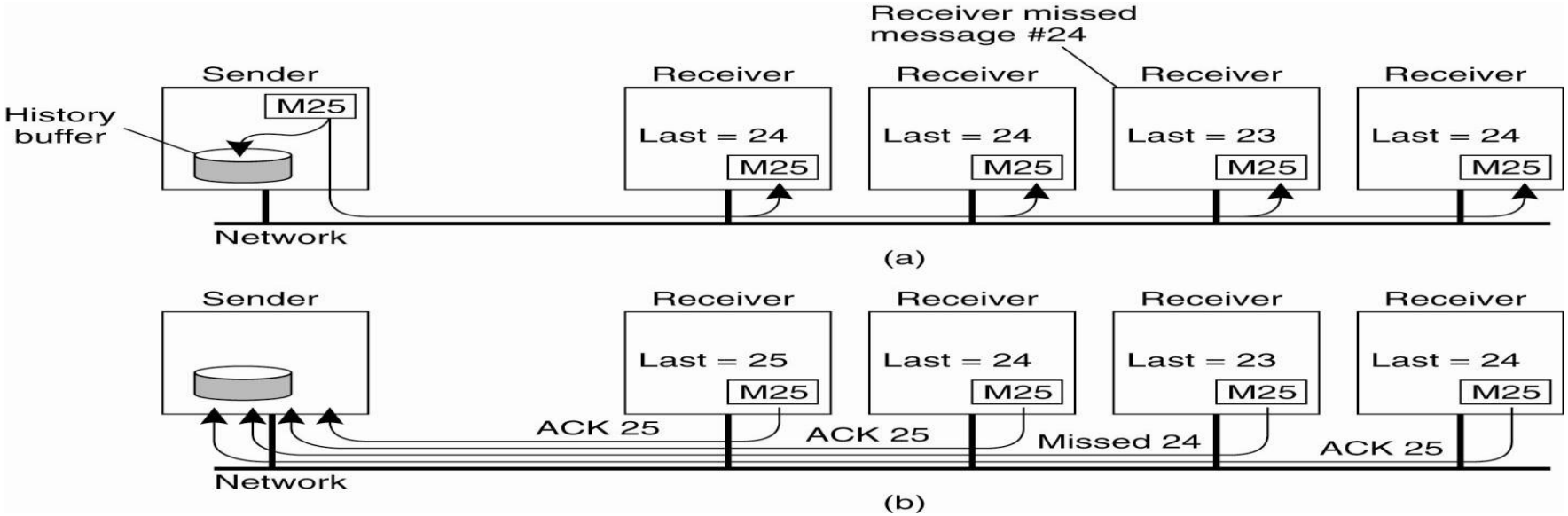
ii) ***Replicated-write protocols***. Active replication as well as quorum based protocols. Corresponds to organizing identical processes as a flat group. Have no single point of failure, at the cost of distributed coordination

A system is said to be ***k fault tolerant*** if it can survive faults in k components and still meet its specifications.

RELIABLE GROUP COMMUNICATION

- Reliable multicast services guarantee messages are delivered to all members in process group

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.

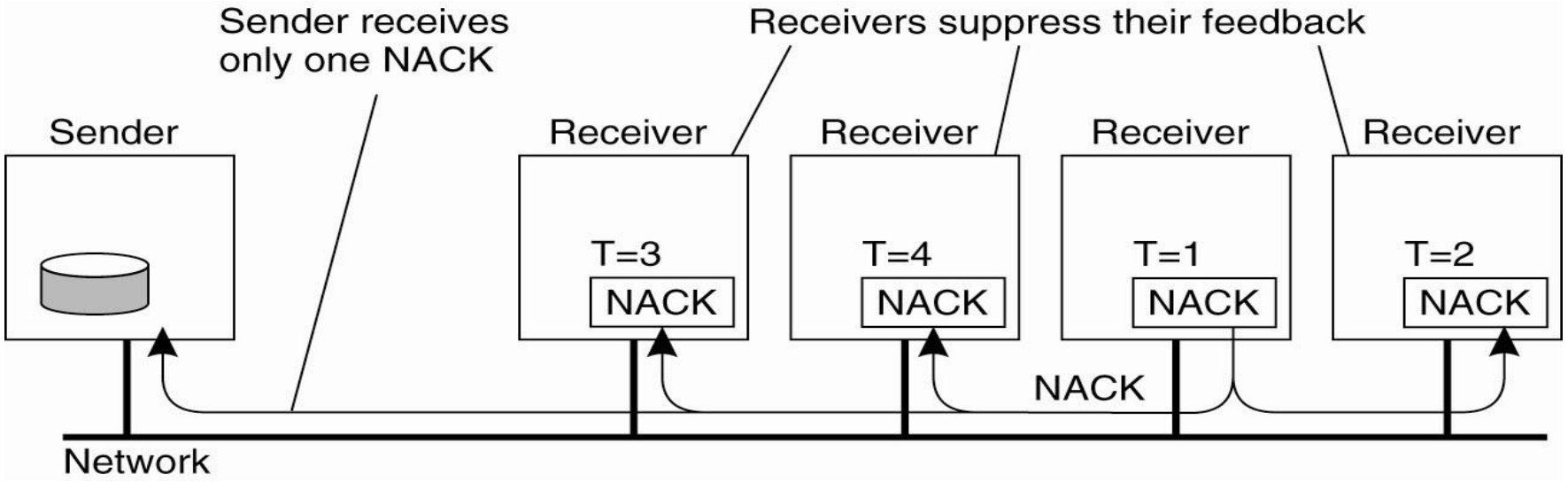
Basic Reliable-Multicasting Schemes Scalability

- Sender cannot accommodate large numbers of receivers, due to the large numbers of ACKs it'll receive

- Solution: use only NACK but may not guarantee that all have received

- Problem: sender may be forced to keep messages in buffer "forever" to resend for a NACK.

Nonhierarchical Feedback Control



- **Feedback suppression**: means to reduce feedback messages returned to sender.

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

Nonhierarchical Feedback Control

Scalable Reliable Multicasting (SRM) protocol:
-Receivers never ACK for success; they only send NACKs.

- Receiver multicasts NACK feedback to whole group to suppress more NACKS

- Receivers use a random delay instead of immediately sending NACKs. similar to binary exponential backoff algorithm receiver that generates smallest random number waits shortest amount of time and sends a NACK back, which alerts all other receivers that would send a NACK so that they suppress their NACKs.

In this approach, all receivers are treated equally and have equal opportunity to send feedback. For that reason, it's a non-hierarchical approach.

Non-hierarchical Feedback Control

- **Advantages and Disadvantages to This Approach:**

-It scales reasonably well.

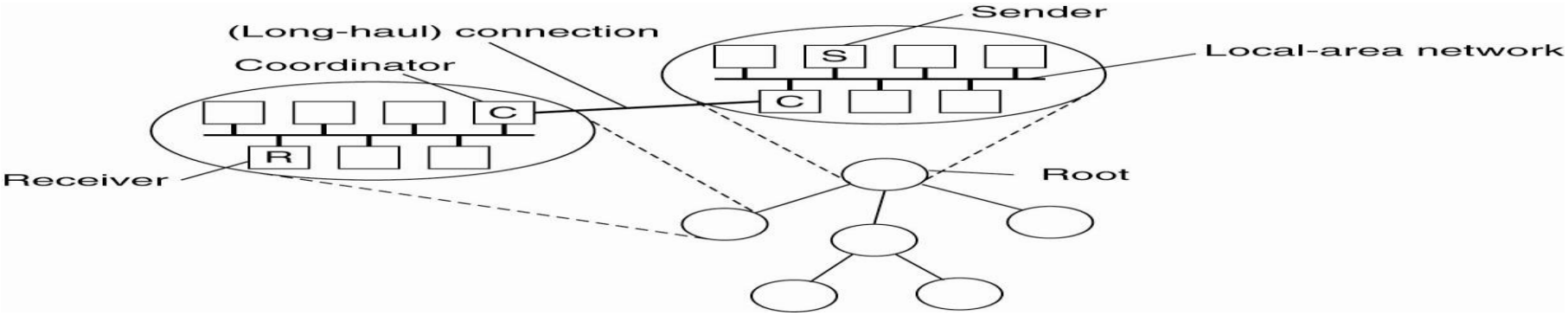
-Timing is tricky: if the processes aren't synchronized fairly well, then it's likely that multiple processes will still send feedback simultaneously or quite close.

-Even if they **are** well-synchronized, the delay in large-scale WANs can cause "simultaneous" feedback to occur (random timeouts differ in small milli seconds)

-

- Processes are forced to process messages they don't care (much) about.

Hierarchical Feedback Control



The essence of hierarchical reliable multicasting.

Each local coordinator forwards the message to its children and later handles retransmission requests.

- 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.

Hierarchical Feedback Control

- Main problem: how to build a hierarchical tree.
- Coordinator may implement any reliable scheme suitable for small groups.
- Coordinator must handle retransmission requests for receivers in its subgroups, so it has to have its own history buffer

Atomic multicast

Reliable multicasting in the presence of process failures.

- message is delivered to either all processes or to none
- all messages are delivered in same order to all processes

Atomic multicast…

- Replicated database constructed as an application on top of a distributed system. The distributed system offers reliable multicasting facilities

- The replicated database is therefore constructed as a group of processes, one process for each replica.

- Update operations are always multicast to all replicas and subsequently performed locally

- Series of updates is to be performed and a replica crashes, update is lost of the replica but performed on other replicas

- When brought up replica needs to be kept upto date

Atomic multicast…

•Atomic multicast requires:

- update performed if remaining replicas agree that crashed replica no longer belongs to group

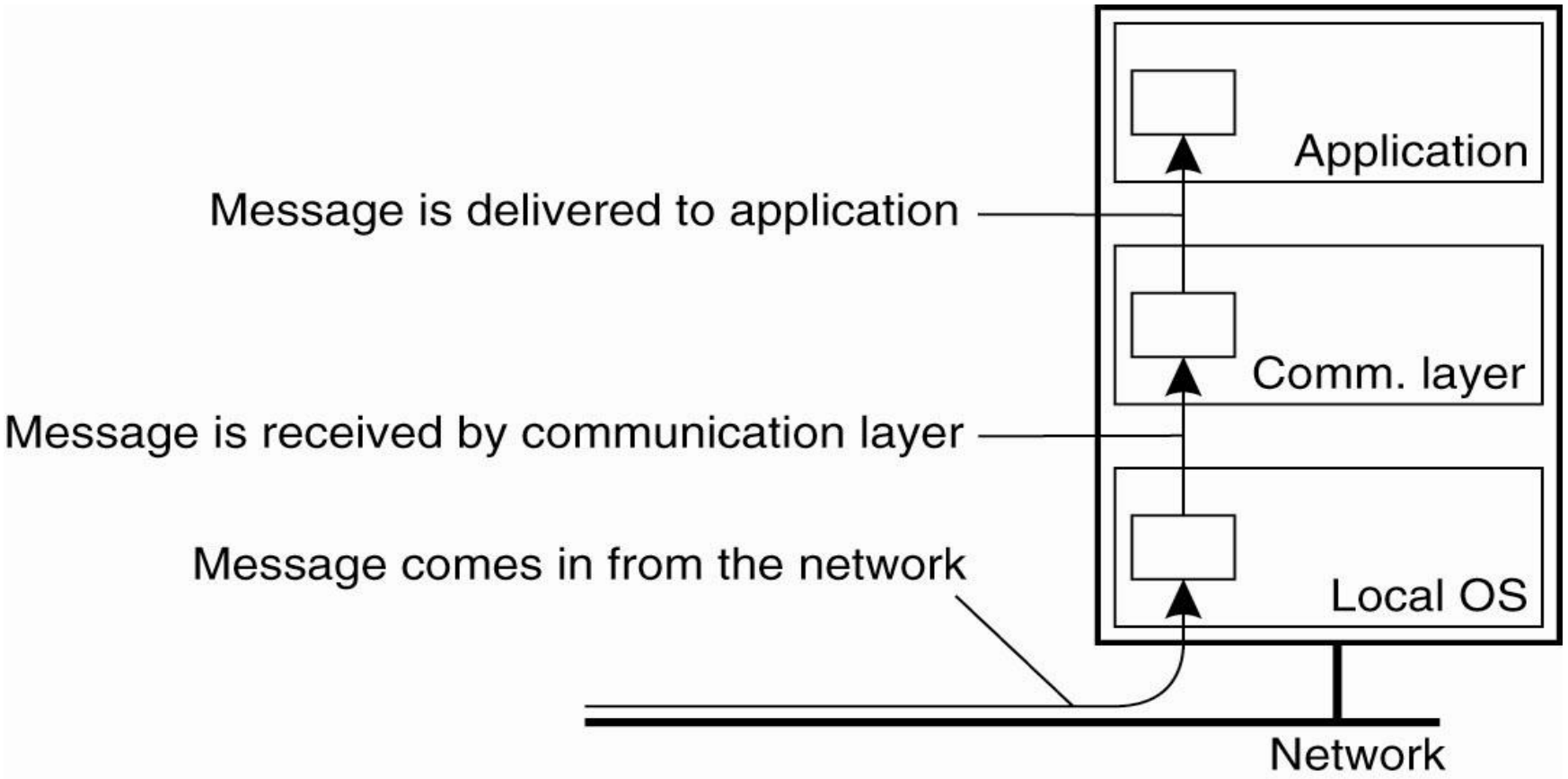
- crashed replica on recovery, forced to join group once more. update operations forwarded until after re-registration.

- atomic multicasting ensures non faulty processes maintain consistent view of database, and forces reconciliation when replica recovers and rejoins the group.

- distributed system consists of a communication layer within which messages are sent and received

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

Virtual Synchrony



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

Virtual Synchrony
Receiving vs Delivery

- Receiving message means that operating system communication layer has received it.

- Delivery means that destination process has received it.

Group View

•Multicast message *m* is uniquely associated with list of processes to which it should be delivered.

•**Group view** means the list of processes currently in the group at the time of a message being sent.

•A **view change** is act of process entering or leaving group (thus, changing the group's size and changing the sender's group view).

•Couple types of messages can be in transit simultaneously:
- *m* (sender's multicast message) & *vc* (view change message).

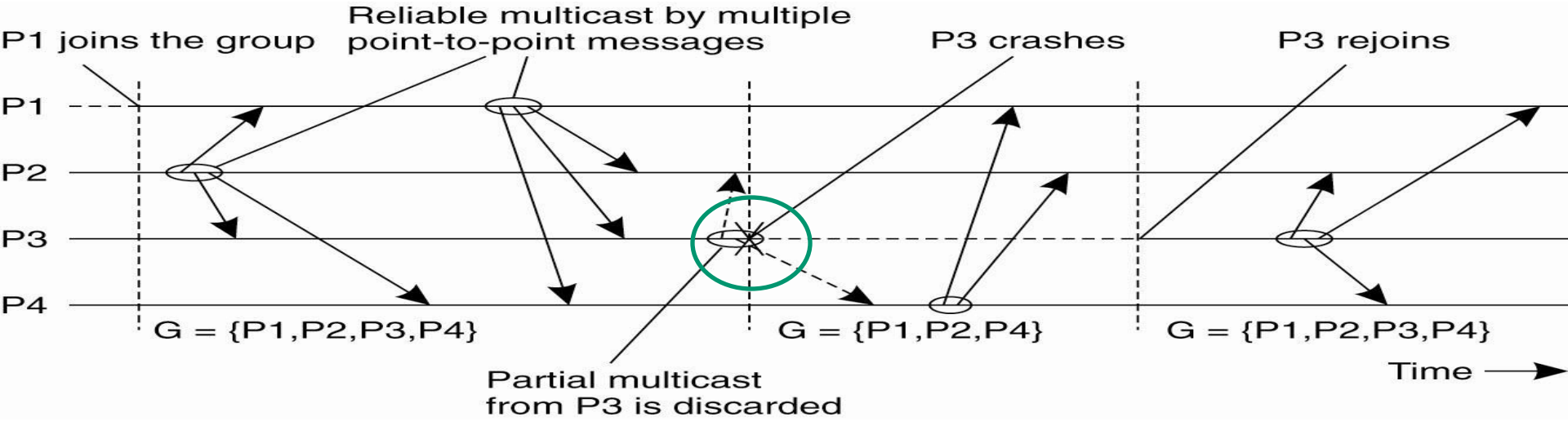
•Proper implementation: *m* must be delivered to all group members before they receive *vc*, or it is not delivered at all.

Group View..

- There's only one situation in which delivery of a multicast message is allowed to fail: situation in which group membership change occurs because the sender of the multicast crashed.

- In **virtually synchronous** multicast, if sender of message crashes during multicast, then message is ignored by all members of multicast group.

Virtual Synchrony



- All multicasts take place between view changes.
- View change acts as a barrier across which no multicast can pass and is comparable to the use of a synchronization variable in distributed data stores

Consider the four processes:

-process PI joins the group, which then consists of P1 P2, P3, and P4

-After some messages have been multicast, P3 crashes.

-Before crashing. it succeeded in multicasting a message to process P2 and P4, but not to PI.

Virtual synchrony guarantees that the message is not delivered at all, effectively establishing the situation that the message was never sent before P3 crashed.

Message Ordering

Four different orderings are:

- Unordered multicasts
- FIFO-ordered multicasts
- Causally-ordered multicasts
- Totally-ordered multicasts

Reliable, unordered multicast: Message Ordering…

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.
Ordering of events per process is shown along vertical axis.

FIFO-ordered multicasting Message Ordering…

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

-Communication layer is forced to deliver incoming messages from the same process in the same order as they have been sent

Message Ordering…

- **Reliable causally-ordered multicast**
  - Delivers messages so that causality between different messages is preserved.

  - If message m1 causally precedes another message m2, then communication layer at each receiver will always deliver m2 after it has received and delivered m1.

- **Total-ordered delivery:**
  - Regardless of whether message delivery is unordered, FIFO ordered/causally ordered, it is required that when messages are delivered, they are delivered in same order to all group members.

- FIFO ordering should also be respected

Message Ordering…

Virtually synchronous reliable multicasting offering totally-ordered delivery of messages is called **atomic multicasting**

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.

Distributed Commit

- **Distributed** commit - operation being performed by each member of a process group, or none at all.

Example: Atomic multicasting

Distributed Commit
One-Phase Commit

The coordinator tells all processes whether or not to (locally) perform the operation in question.

Disadvantage: If one of the participants cannot perform the operation, then there is no way to tell the coordinator.

Two Phase Commit

Assuming that no failures occur, the protocol consists of the following two phases, each consisting of two steps

Phase 1: Voting Phase

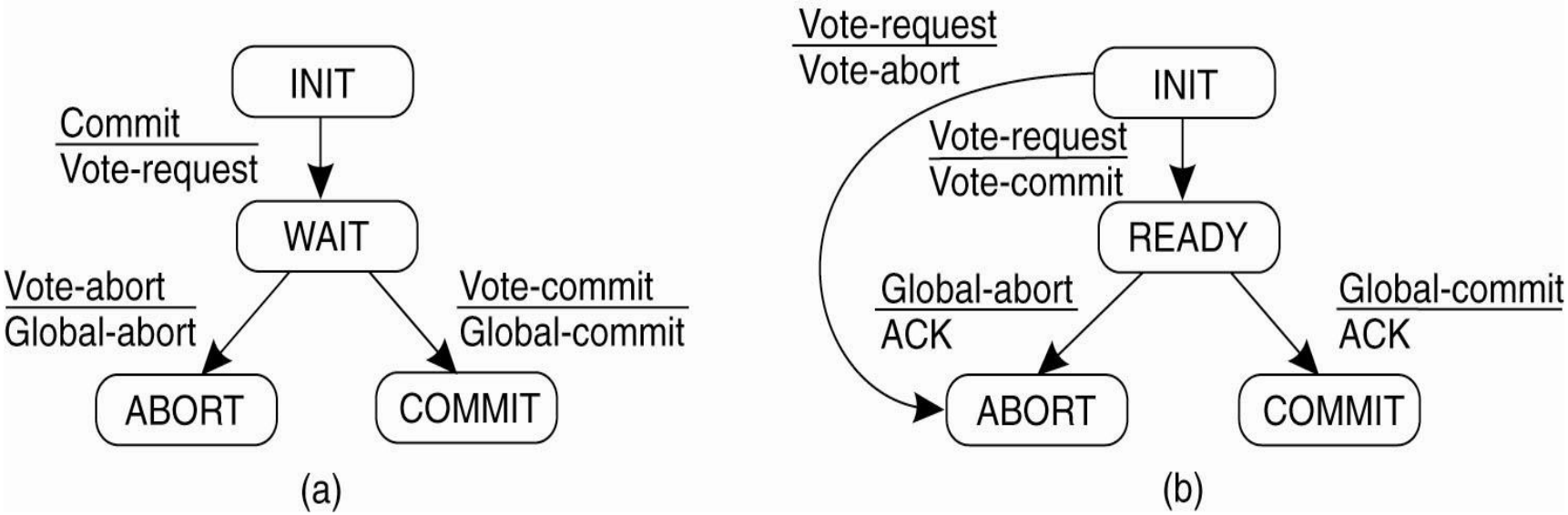
1. Coordinator sends a VOTE-.REQUEST message to all participants.
2. Participant receives a VOTE-.REQUEST message, and returns VOTE_COMMIT for local commit / VOTE-ABORT

Phase 2: Decision Phase

1. Coordinator collects all votes from participants. If all have voted to commit then it sends GLOBAL_COMMIT to all participants else GLOBAL..ABORT

2. 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.

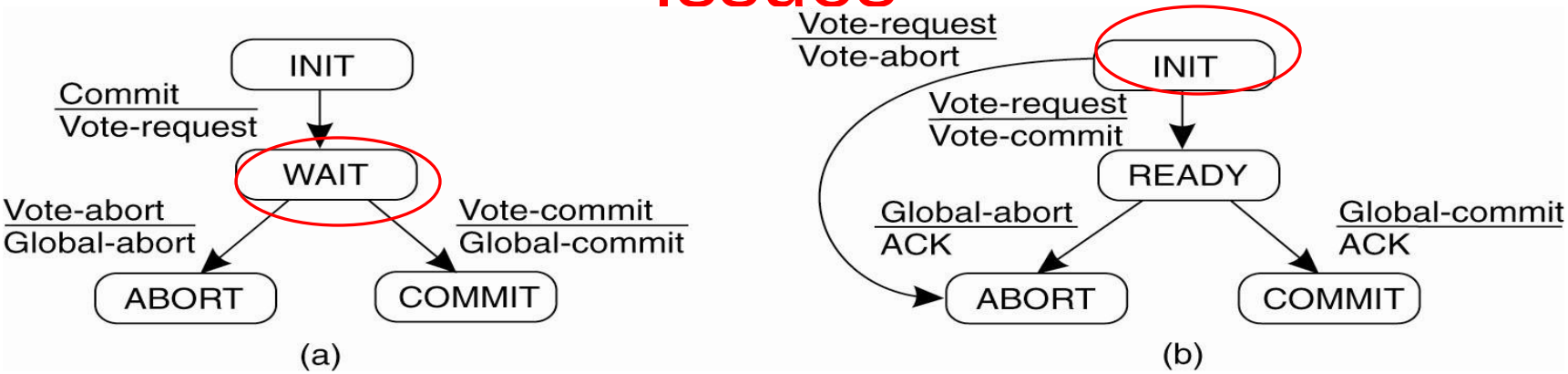
Two Phase Commit



finite state machine for the coordinator

finite state machine for a participant

Two Phase Commit… Issues

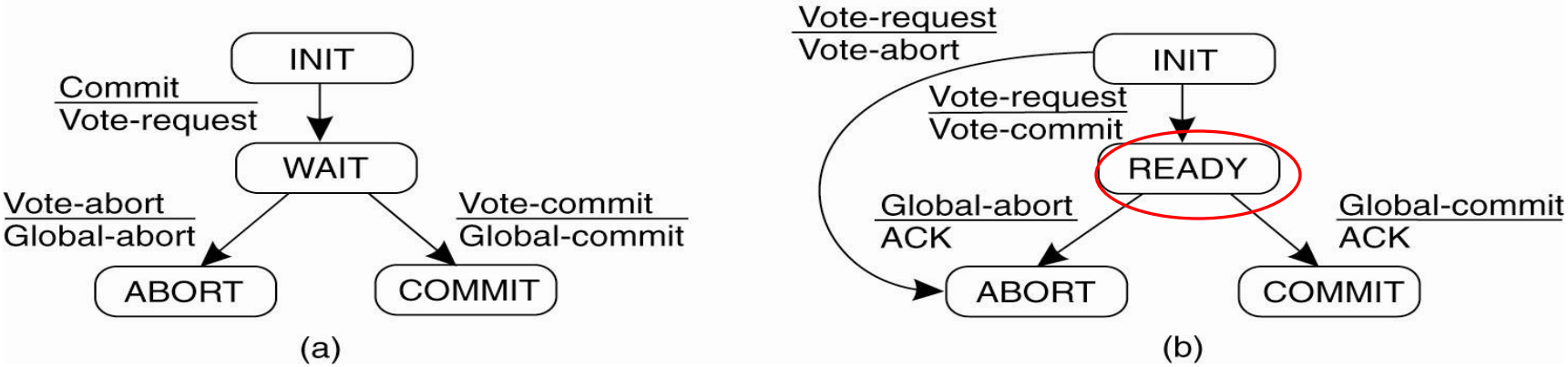


- Processes (coordinator/participants) have states in which they block waiting for incoming messages. If the other process crashes there may be a failure – hence require timers.

1. Participant may be waiting in its INIT state for a VOTE-REQUEST message from the coordinator. If that message is not received after some time, the participant will simply decide to locally abort the transaction, and thus send a VOTE..ABORT message to the coordinator (Fig (b))

2. Coordinator is waiting for participant votes & process takes too long, then coordinator votes abort & sends global abort message. (Fig (a))

Two Phase Commit… Issues



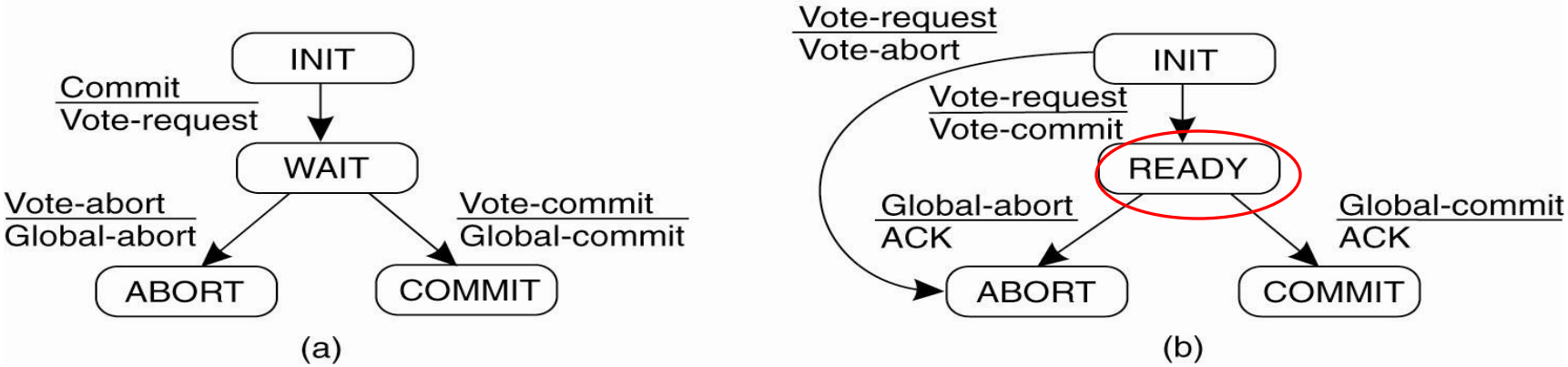
3. Participant has entered READY state and waits for coordinator to send global vote. On timeout cannot imply abort transaction. Need to find what message was sent by coordinator by contacting another participant Q.

Several options involving asking another participant/ neighbor Q:

-Q is in commit state, then P can commit. (coordinator started sending out global commit messages and crashed before P was notified)

-Q is in abort state, then P can abort.(coordinator crashed before notifying P of the abort.)

Two Phase Commit… Issues



-Q is still in init state=> P was given vote request message, then coordinator crashed before all other processes were given vote request messages. Thus, commit can be aborted.

- Q is also in ready state=> all processes must block until coordinator **recovers**

Two-Phase Commit (2)

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

Actions taken by a participant *P* when residing in state *READY* and having contacted another participant *Q*.

Two Phase Commit…

- Process recovery – it must save state in persistent storage

- eg process crashes in INIT stage - decide to locally abort the transaction when it recovers, and then inform the coordinator

-When it crashed it is in either COMMIT or ABORT state – it must recover to that state again and retransmit its decision to coordinator again

- if it crashed in READY state it must contact other participants

Coordinator needs to keep track of two critical states:

1. If it has initialized and entered WAIT state and crashes then it must record entering WAIT state so that VOTE REQUEST can be sent to all participants

2. When it crashes in second phase ABORT or COMIT state it must record it so that decision can be resent after recovery

Two-Phase Commit (3)

actions by coordinator:

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

Outline of the steps taken by the coordinator in a two phase commit protocol

Two-Phase Commit (4)

actions by participant:

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

Two-Phase Commit (5)

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

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

Steps taken for handling incoming decision requests.

if the participant times out while waiting for the coordinator's decision to come in, it executes a termination protocol by first multicasting a *DECISION-REQUEST message to the other processes, after which it subsequently* blocks while waiting for a response.

2PC is also referred to as a **blocking commit protocol.**

•all participants have received and processed the *VOTE-REQUEST from the coordinator*
•*in the* meantime, the coordinator crashed.
•Participants cannot cooperatively decide on the final action to take.

Solution:
1. Receiver immediately multicasts a received message to all other processes
-allows a participant to reach a final decision, even if the coordinator has not yet recovered
2. three-phase commit protocol

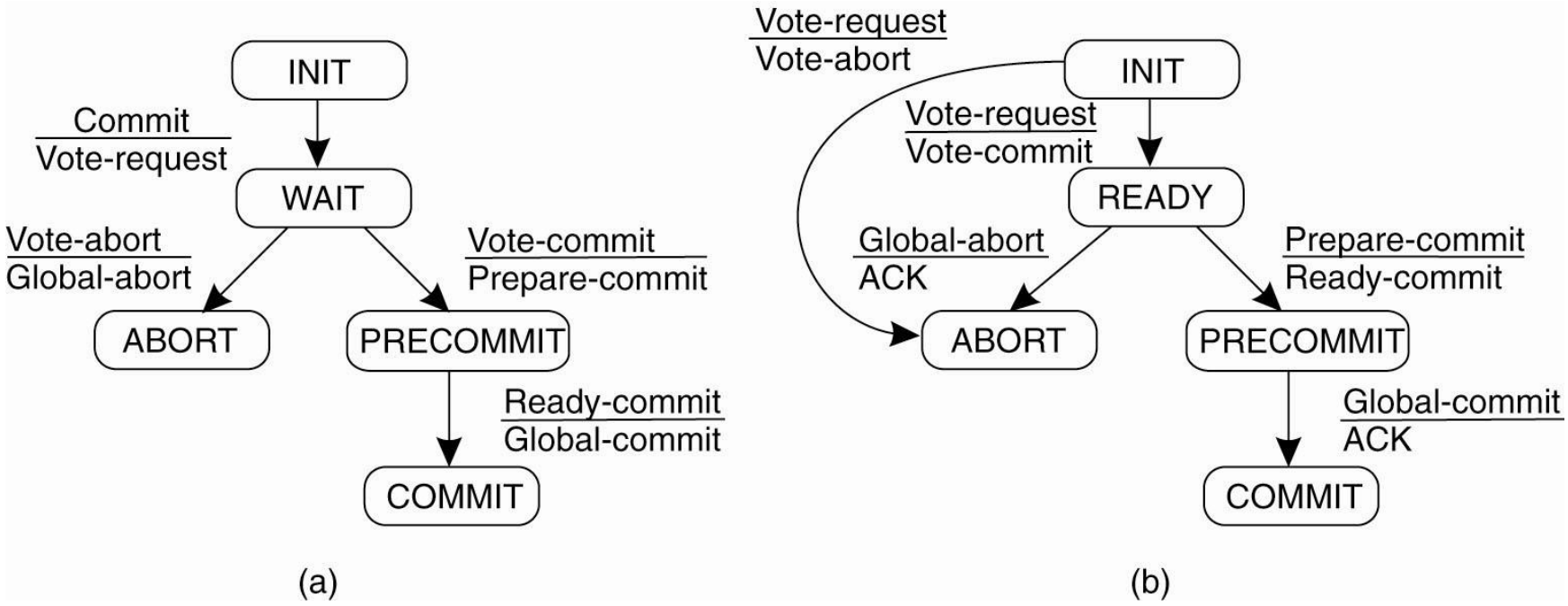
Three-Phase Commit (1)

Overcomes main problem of two phase commit: coordinator has crashed, participants may not be able to reach a final decision

The states of the coordinator and each participant satisfy the following two conditions:

- There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
- There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.

Three-Phase Commit (2)



(a) The finite state machine for the coordinator in 3PC.

b) The finite state machine for a participant.

Three-Phase Commit...

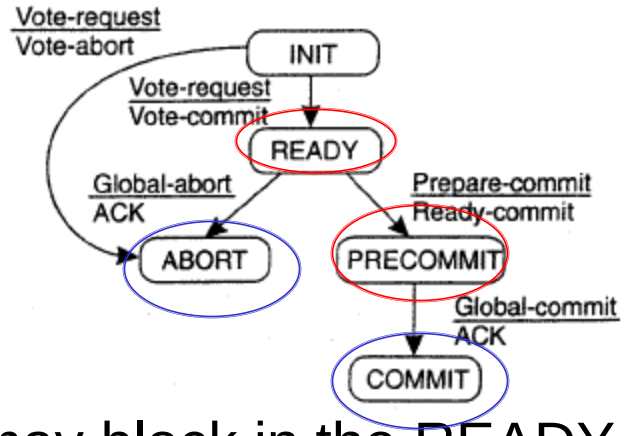
Conditions in which process is blocked while waiting for incoming messages:

1. Participant in INIT state waiting for vote request from coordinator on timeout:
   - will eventually make a transition to state ABORT, assuming that coordinator crashed

2. Coordinator in *WAIT state , waiting for votes from participants, On a timeout:*
- *coordinator will* conclude that a participant crashed, and will thus abort the transaction by multicasting a *GLOBAL-ABORT message*

*3. C*oordinator is blocked in *PRECOMMIT state- On a timeout,* will conclude:
- one of the participants had crashed, but that participant is known to have voted for committing the transaction.
- coordinator can safely instruct operational participants to commit by multicasting *GLOBAL_COMMIT message.*
- crashed participant will commit its part of the transaction when it comes up again.

Three-Phase Commit..



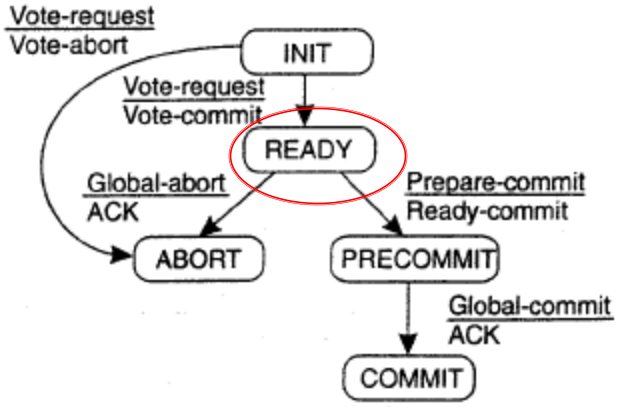
P may block in the READY state or in the PRECOMMIT state and on timeout, concludes coordinator has failed. It can as in 2 PC:

- contact any other participant that is in state COMMIT (or ABORT), and P should move to that state as well.

- if all participants are in state PRECOMMIT, transaction can be safely committed

- if another participant Q is still in the *INIT state, the* transaction can safely be aborted , No other process can be in PRECOMMIT state if Q is in INIT state since the coordinator first moves to PRECOMMIT state and then the other participants move to PRECOMMIT state.

Three-Phase Commit....



If each: of the participants that *P can contact is in state READ Y (and they* together form a majority), the transaction should be aborted

- Any crashed process when recovers can return back in only either INIT,ABORT or PRECOMMIT state and in that case it is OK to abort to go with the majority
- unlike in 2PC surviving processes can always come to final decision.

If each of the participant that P can contact is in PRECOMMIT state then it is safe to commit the transaction:

- All the other processes that crashes and then recovered will either be in READY state or will recover to READY, PRECOMMIT or COMMIT state.

Summary

- Fault Tolerance Basics
- Types of Faults
- Process redundancy for process resilience
- Formation of process groups - flat , hierarchical
- Reliable communication (Scalable Reliable multicasting SRM protocol)
- Atomic Multicast
- Virtual Synchrony
- Message Ordering
- Distributed commit protocols

THANKS