Fault Tolerance

Dealing successfully with *partial failure* within a Distributed System.

Key technique: ***Redundancy***.

Basic Concepts

- *Fault Tolerance* is closely related to the notion of "Dependability". In Distributed Systems, this is characterized under a number of headings:

- *Availability* – the system is ready to be used immediately.
- *Reliability* – the system can run continuously without failure.
- *Safety* – if a system fails, nothing catastrophic will happen.
- *Maintainability* – when a system fails, it can be repaired easily and quickly (and, sometimes, without its users noticing the failure).

But, What Is "Failure"?

- Definition:

- A system is said to "fail" when it *cannot meet* its promises.

- A failure is brought about by the *existence* of "errors" in the system.

- The *cause* of an error is called a "fault".

Types of Fault

- There are three main types of 'fault':
- *Transient Fault* – appears once, then disappears.
- *Intermittent Fault* – occurs, vanishes, reappears; but: follows no real pattern (worst kind).
- *Permanent Fault* – once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally.

Classification of Failure Models

- Different types of failures, with brief descriptions.

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

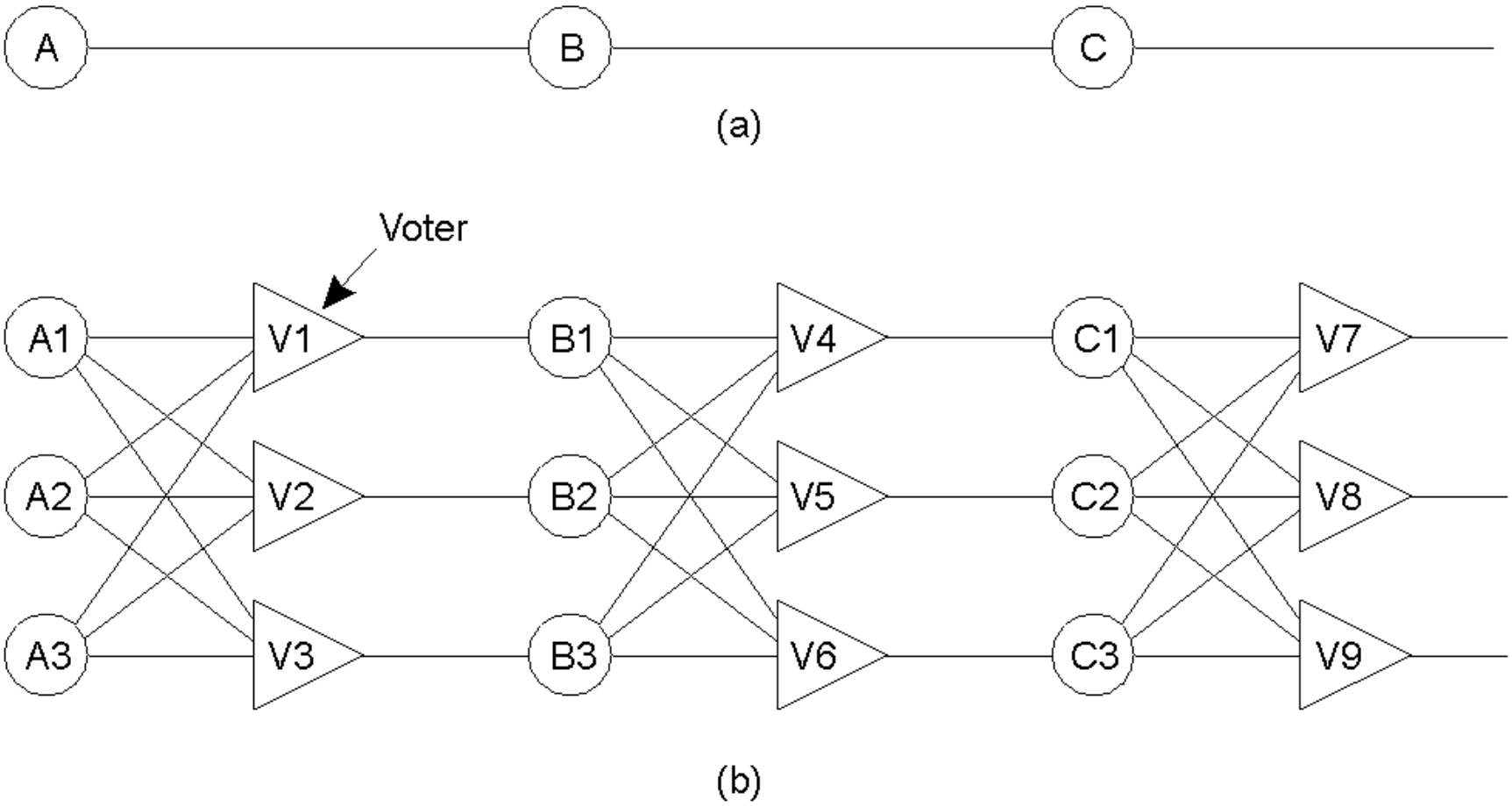
Failure Masking by Redundancy

- **Strategy**: hide the occurrence of failure from other processes using *redundancy*. Three main types:

- *Information Redundancy* – add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).
- *Time Redundancy* – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).
- *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

Failure Masking by Redundancy

- Triple modular redundancy. (Physical Redundancy)



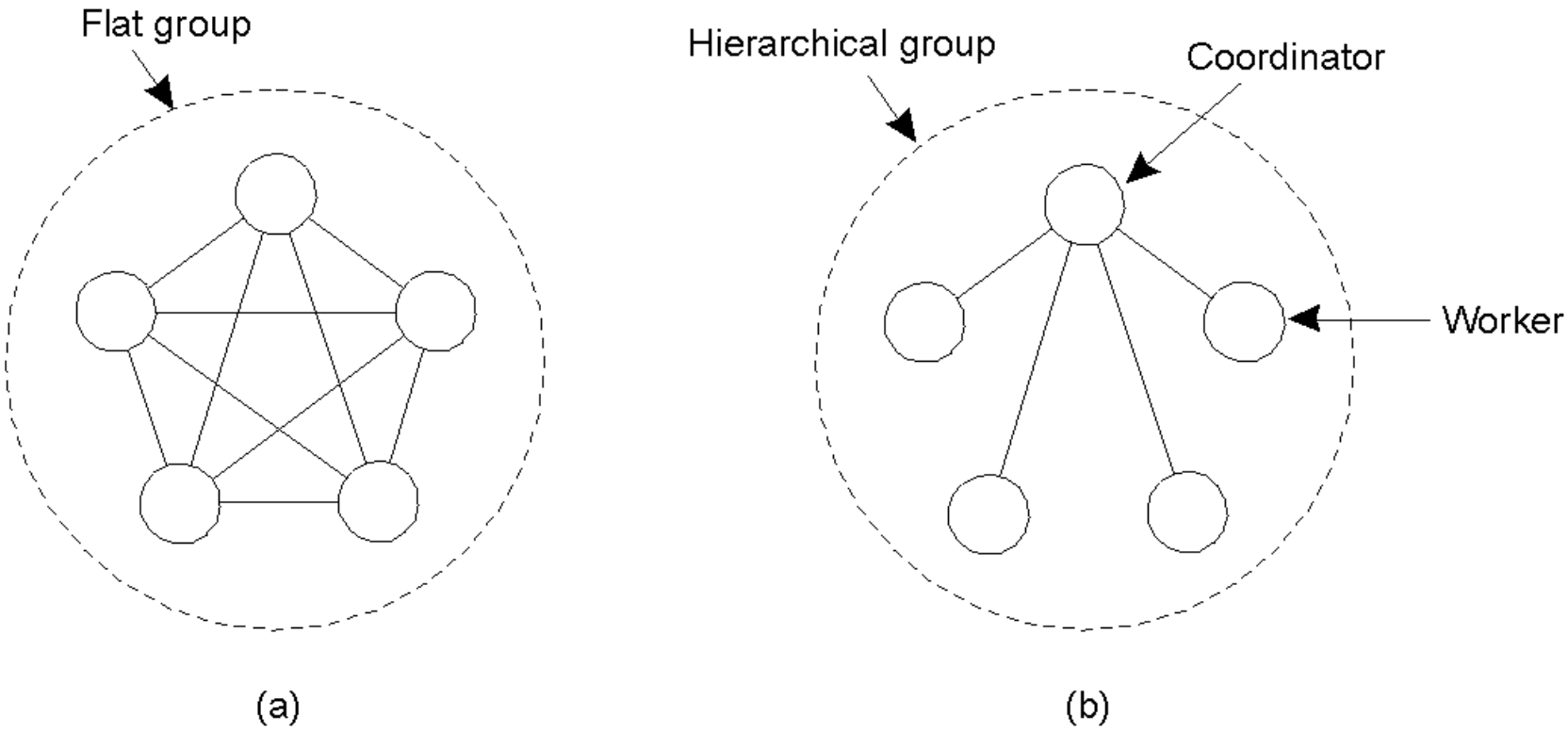
DS Fault Tolerance Topics

1. Process Resilience
2. Reliable Client/Server Communications
3. Reliable Group Communication
4. Distributed COMMIT
5. Recovery Strategies

1. Process Resilience

- Processes can be made fault tolerant by arranging to have a group of processes, with each member of the group being *identical*.
- A message sent to the group is delivered to all of the "copies" of the process (the group members), and then *only one* of them performs the required service.
- If one of the processes fail, it is assumed that one of the others will still be able to function (and service any pending request or operation).

Flat vs. Hierarchical Groups



a) **Communication in a flat group** – all the processes are equal, decisions are made collectively. **Note**: no single point-of-failure, however: decision making is complicated as consensus is required.

b) **Communication in a simple hierarchical group** – one of the processes is elected to be the coordinator, which selects another process (a worker) to perform the operation. **Note**: single point-of-failure, however: decisions are easily and quickly made by the coordinator without first having to get consensus.

Failure Masking and Replication

- By organizing a *fault tolerant group of processes*, we can protect a single vulnerable process.

- There are two approaches to arranging the replication of the group:

1. Primary (backup) Protocols.
2. Replicated-Write Protocols.

The Goal of Agreement Algorithms

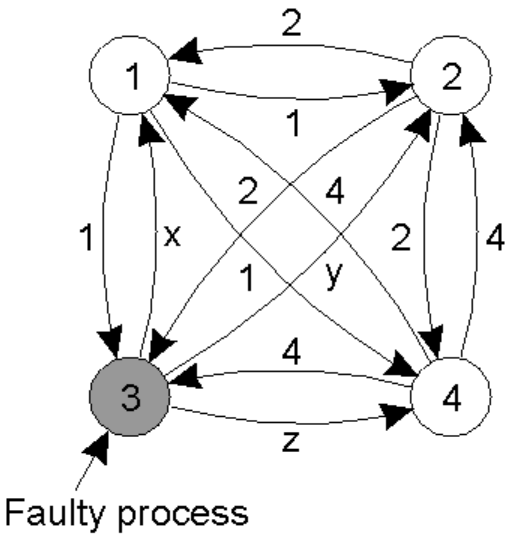
- "To have all *non-faulty* processed reach consensus on some issue (quickly)."
- The **two-army problem**. pp. 372
- Even with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.

History Lesson: The Byzantine Empire

- *Time*: 330-1453 AD.
- *Place*: Balkans and Modern Turkey.
- Endless conspiracies, intrigue, and untruthfullness were alleged to be common practice in the ruling circles of the day (*sounds strangely familiar …*).
- That is: it was typical for intentionally wrong and malicious activity to occur among the ruling group. A similar occurrence can surface in a DS, and is known as 'Byzantine failure'.
- *Question*: how do we deal with such malicious group members within a distributed system?

Agreement in Faulty Systems (1)

How does a process group deal with a faulty member?



(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 Generals Problem" for 3 loyal generals and 1 traitor.

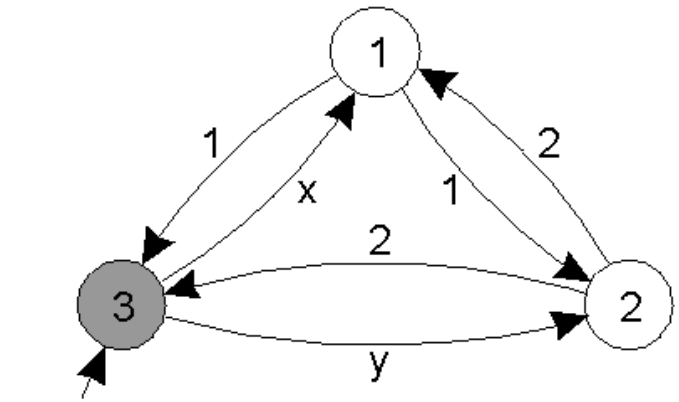
a) The generals announce their troop strengths (in units of 1 kilosoldiers) to the other members of the group by sending a message.

b) The vectors that each general assembles based on (a), each general knows their own strength. They then send their vectors to all the other generals.

c) The vectors that each general receives in step 3. It is clear to all that General 3 is the traitor. In each 'column', the majority value is assumed to be correct.

Agreement in Faulty Systems (2)

Warning: the algorithm does not always work!



(a)

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

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

- The same algorithm as in previous slide, except now with 2 loyal generals and 1 traitor. Note: It is no longer possible to determine the majority value in each column, and the algorithm has failed to produce agreement.
- It has been shown that for the algorithm to work properly, *more* than two-thirds of the processes have to be working correctly. That is: if there are M faulty processes, we need 2M + 1 functioning processes to reach agreement.

2 .Reliable Client/Server Comms.

- In addition to process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures.

- In practice, the focus is on masking *crash* and *omission* failures.

- ***For example***: the point-to-point TCP masks omission failures by guarding against lost messages using ACKs and retransmissions. However, it performs poorly when a crash occurs (although a DS may try to mask a TCP crash by automatically re-establishing the lost connection).

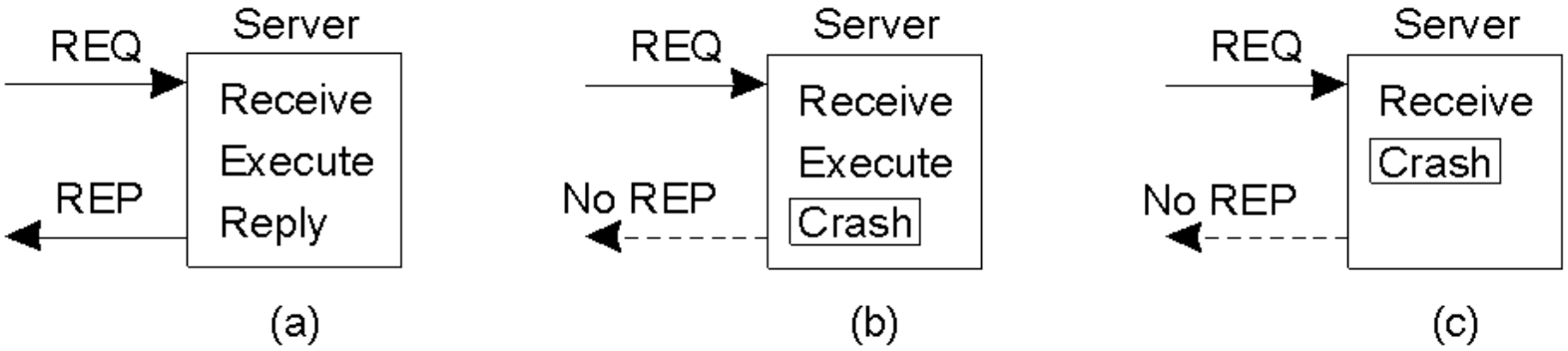
Example: RPC Semantics and Failures

- The RPC mechanism works well as long as both the client and server function perfectly. (the higher level)

- Five classes of RPC failure can be identified:

1. *The client cannot locate the server*, so no request can be sent.
2. *The client's request to the server is lost*, so no response is returned by the server to the waiting client.
3. *The server crashes after receiving the request*, and the service request is left acknowledged, but undone.
4. *The server's reply is lost on its way to the client*, the service has completed, but the results never arrive at the client
5. *The client crashes after sending its request*, and the server sends a reply to a newly-restarted client that may not be expecting it.

The Five Classes of Failure (1)



- A server in client-server communication.

a) The normal case.
b) Crash *after* service execution.
c) Crash *before* service execution.

The Five Classes of Failure (2)

- An appropriate exception handling mechanism can deal with a missing server. However, such technologies tend to be very language-specific, and they also tend to be non-transparent.

- Dealing with lost request messages can be dealt with easily using timeouts. If no ACK arrives in time, the message is resent. Of course, the server needs to be able to deal with the possibility of duplicate requests.

The Five Classes of Failure (3)

- Server crashes are dealt with by implementing one of three possible implementation philosophies:
  - *At least once semantics*: a guarantee is given that the RPC occurred at least once, but (also) possibly more that once.
  - *At most once semantics*: a guarantee is given that the RPC occurred at most once, but possibly not at all.
  - *No semantics*: nothing is guaranteed, and client and servers take their chances!
- It has proved difficult to provide *exactly once semantics*.

The Five Classes of Failure (4)

- Lost replies are difficult to deal with.
- *Why* was there no reply? Is the server *dead*, *slow*, or did the reply just go *missing*? Emmmmm?

- A request that can be repeated any number of times without any nasty side-effects is said to be *idempotent*. (For example: a read of a static web-page is said to be idempotent).

- *Nonidempotent* requests (for example, the electronic transfer of funds) are a little harder to deal with. A common solution is to employ *unique sequence numbers*. Another technique is the inclusion of additional bits in a retransmission to identify it as such to the server.

The Five Classes of Failure (5)

- When a client crashes, and when an 'old' reply arrives, such a reply is known as an *orphan*.

- Four orphan solutions have been proposed:

  - *extermination* (the orphan is simply killed-off),
  - *reincarnation* (each client session has an *epoch* associated with it, making orphans easy to spot),
  - *gentle reincarnation* (when a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed), and,
  - *expiration* (if the RPC cannot be completed within a standard amount of time, it is assumed to have expired).

- In practice, however, none of these methods are desirable for dealing with orphans. Research continues …

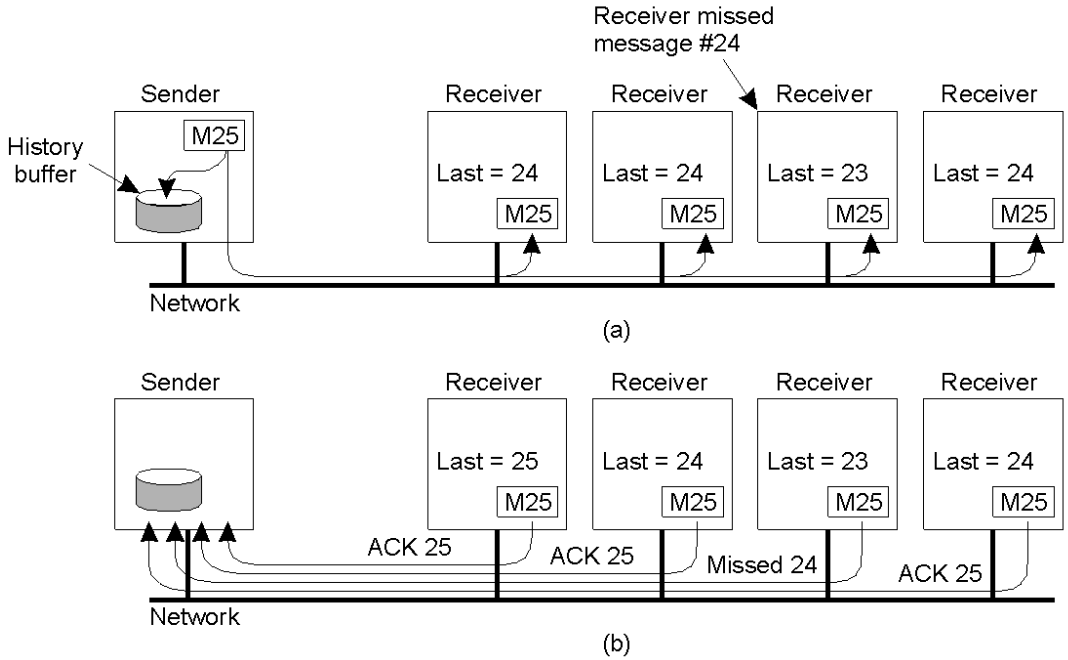
3. Reliable Group Communication

- Reliable multicast services guarantee that all messages are delivered to all members of a process group.

- Sounds simple, but is surprisingly *tricky* (as multicasting services tend to be *inherently* unreliable).

- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution *scales poorly* as the group membership grows. Also:
  - What happens if a process *joins* the group during communication?
  - Worse: what happens if the sender of the multiple, reliable point-to-point channels *crashes* half way through sending the messages?

Basic Reliable-Multicasting Schemes



- This is a simple solution to reliable multicasting when *all receivers are known* and are assumed *not to fail*. The sending process assigns a sequence number to outgoing messages (making it easy to spot when a message is missing).

a) Message transmission – note that the third receiver is expecting 24.

b) Reporting feedback – the third receiver informs the sender.

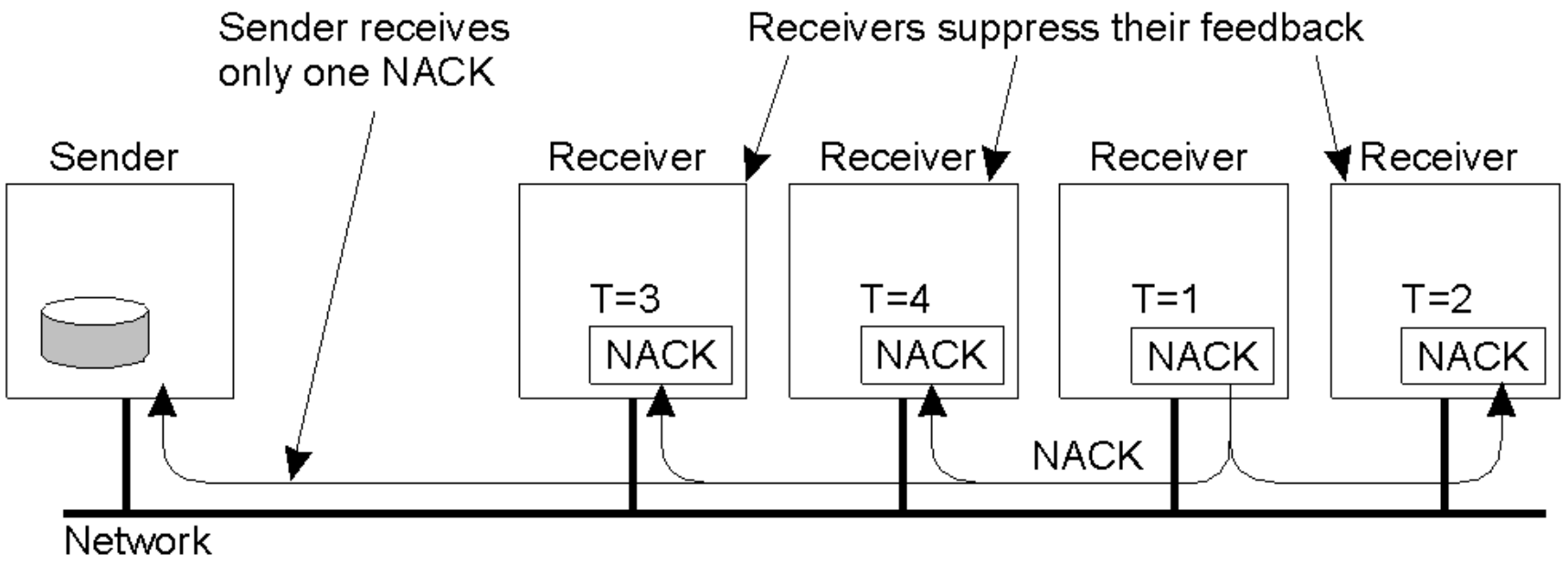
c) But, how long does the sender keep its *history-buffer* populated?

d) Also, such schemes **perform poorly** as the group grows … there are *too many* ACKs.

SRM: Scalable Reliable Multicasting

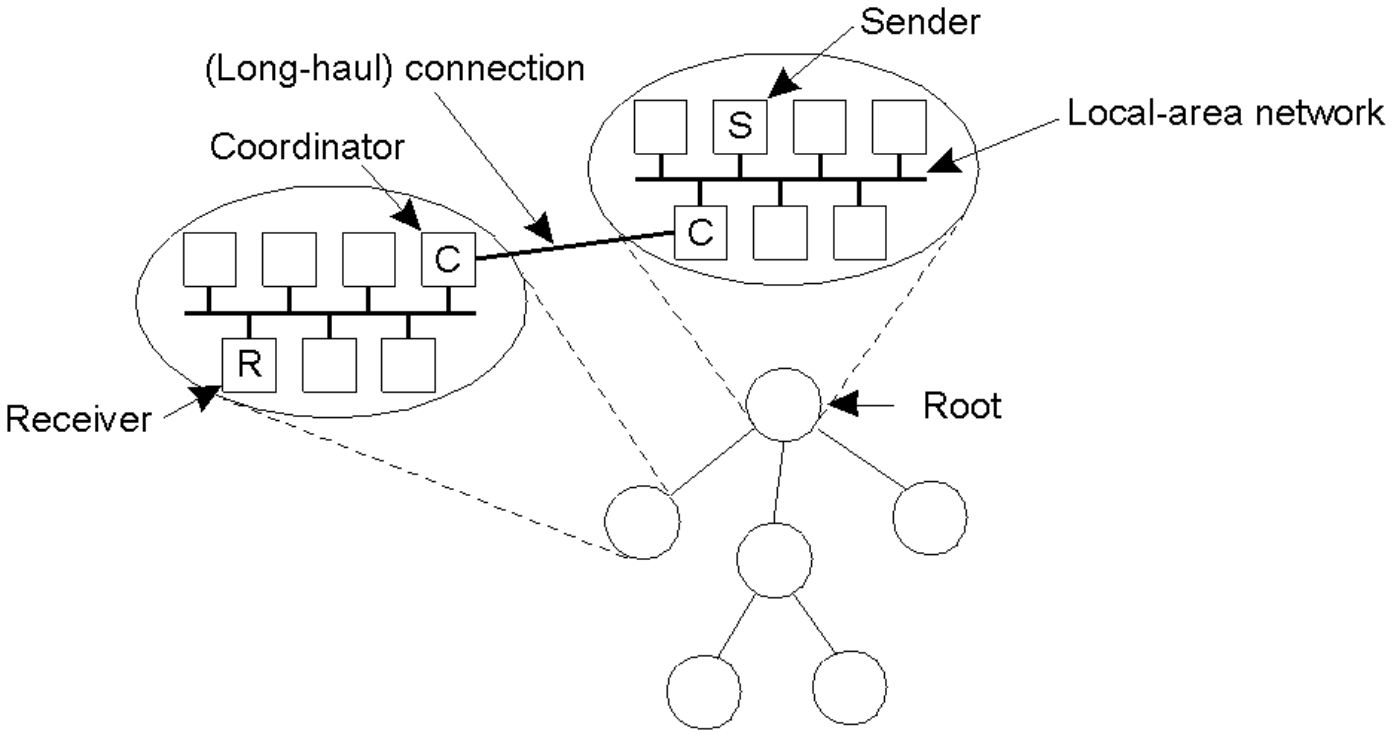
- Receivers *never* acknowledge successful delivery.
- **Only missing messages are reported.**
- NACKs are multicast to all group members.
- This allows other members to suppress their feedback, if necessary.
- To avoid "retransmission clashes," each member is required to wait a random delay prior to NACKing.

Nonhierarchical Feedback Control



- *Feedback Suppression* – reducing the number of feedback messages to the sender (as implemented in the *Scalable Reliable Multicasting Protocol*).
- Successful delivery is never acknowledged, only missing messages are reported (NACK), which are multicast to all group members. If another process is about to NACK, this feedback is suppressed as a result of the first multicast NACK. In this way, only a **single** NACK is delivered to the sender.

Hierarchical Feedback Control



- Hierarchical reliable multicasting is another solution, the main characteristic being that it supports the creation of **very large groups**.

a) Sub-groups within the entire group are created, with each *local coordinator* forwarding messages to its children.

b) A local coordinator handles retransmission requests *locally*, using any appropriate multicasting method for small groups.

Atomic Multicasting

- There often exists a requirement where the system needs to ensure that all processes get the message, or that none of them get it.
- An additional requirement is that all messages arrive at all processes in sequential order.
- This is known as the “atomic multicast problem”.

4. Distributed COMMIT

- **General Goal:**

- *We want an operation to be performed by all group members, or none at all*.

- [In the case of atomic multicasting, the operation is the delivery of the message.]

- There are three types of “commit protocol”: single-phase, two-phase and three-phase commit.

Commit Protocols

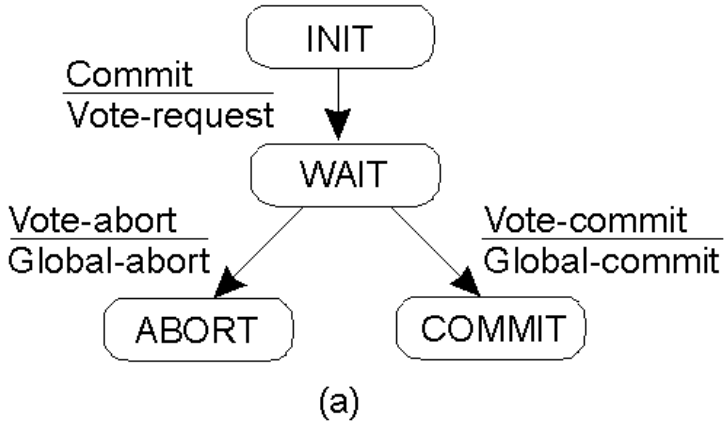
- **One-Phase Commit Protocol**:
  - An elected co-ordinator tells all the other processes to perform the operation in question.
  - But, what if a process cannot perform the operation? There's no way to tell the coordinator! Whoops …
- **The solutions**:
  - The *Two-Phase* and *Three-Phase Commit Protocols*.

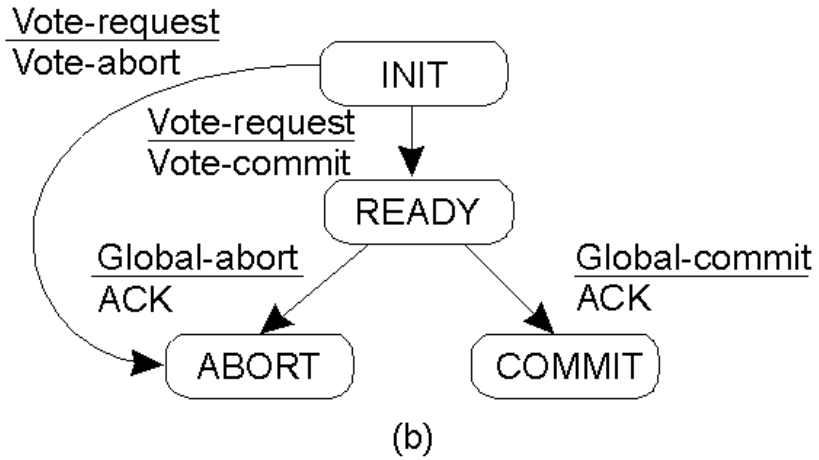
The Two-Phase Commit Protocol

- First developed in 1978!!!
- *Summarized: GET READY, OK, GO AHEAD.*

1. The coordinator sends a *VOTE_REQUEST* message to all group members.
2. The group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
3. All votes are collected by the coordinator. A *GLOBAL_COMMIT* is sent if all the group members voted to commit. If one group member voted to abort, a *GLOBAL_ABORT* is sent.
4. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

Two-Phase Commit Finite State Machines





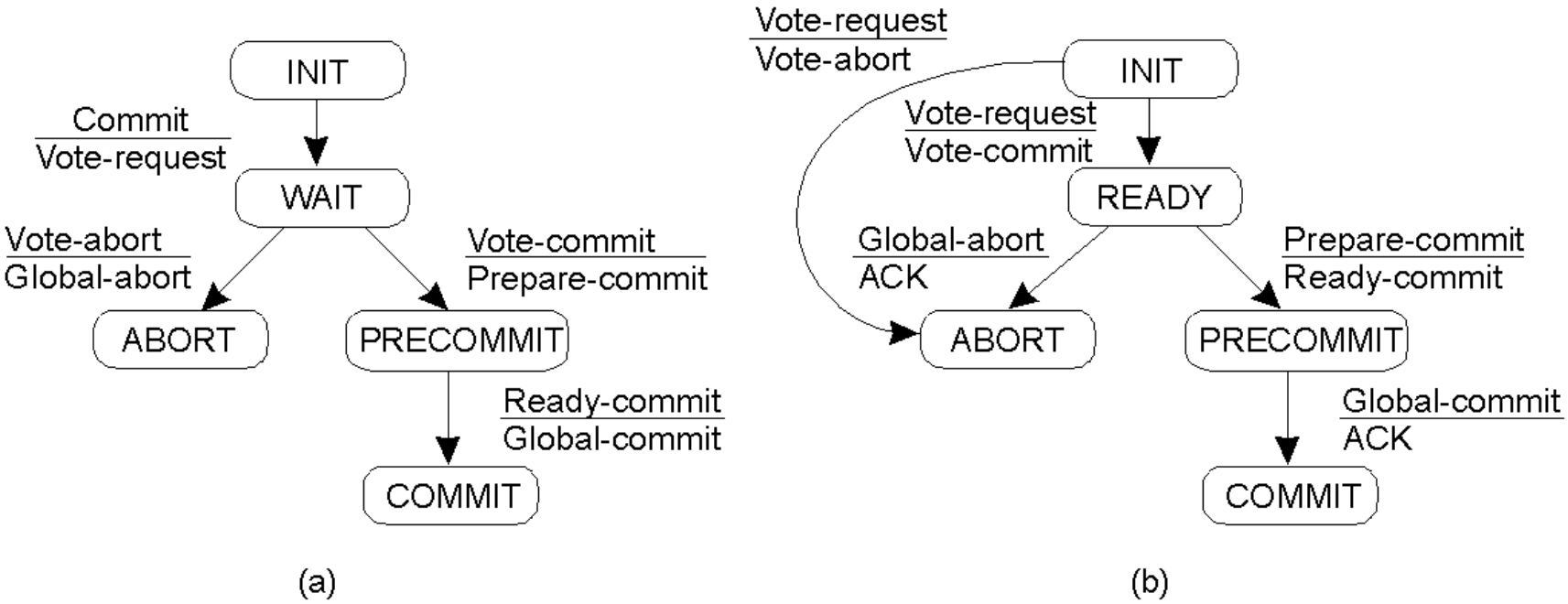
a) The finite state machine for the coordinator.

b) The finite state machine for a participant (group member).

Big Problem with Two-Phase Commit

- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.
- If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers …*
- Two-Phase Commit is known as a **blocking-commit protocol** for this reason.
- The solution? *The Three-Phase Commit Protocol.*

Three-Phase Commit



a) Finite state machine for the coordinator.

b) Finite state machine for a group member.

c) **Main point**: although 3PC is generally regarded as *better* than 2PC, it is *not applied often in practice*, as the conditions under which 2PC blocks rarely occur.

d) Refer to the textbook for details on how this works.

5. Recovery Strategies

- Once a failure has occurred, it is essential that the process where the failure happened *recovers* to a correct state.

- Recovery from an error is *fundamental* to fault tolerance.

- Two main forms of recovery:

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.
2. **Forward Recovery**: bring the system into a correct state, from which it can then continue to execute.

Forward and Backward Recovery

- **Disadvantage of Backward Recovery**:
  - Check pointing (can be very expensive (especially when errors are very rare).
  - [Despite the cost, backward recovery is implemented more often. The "logging" of information can be thought of as a type of check pointing.].

- **Disadvantage of Forward Recovery**:
  - In order to work, all potential errors need to be accounted for *up-front*.
  - When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

Recovery Example

- **Consider as an example: Reliable Communications.**

- *Retransmission* of a lost/damaged packet is an example of a backward recovery technique.

- When a lost/damaged packet can be reconstructed as a result of the receipt of other successfully delivered packets, then this is known as *Erasure Correction*. This is an example of a forward recovery technique.

Stable Storage

- Stable storage is an example of group masking at the disk block level
- Designed to ensure permanent data is recoverable after a system failure during a disk write operation or after a disk block has been damaged

Stable Storage

- Provided by a Careful Storage Service
  - Unit of storage the stable block
  - Each stable block is represented by two careful blocks that hold the contents of the stable block in duplicate
  - Write operation writes one careful block ensuring it is correct before writing the second block
  - Careful blocks are disk blocks stored with a checksum to mask value failures, the blocks are located on different disk drives with independent failure modes
  - Value failures are converted to omission failures
  - The Read operation reads one of the pair of careful blocks, if an omission failure occurs then it reads the other, thus masking the omission failures of the Careful Storage Service

Stable Storage

- Stable Storage - Crash Recovery
  - When a server is restarted after a crash, the pair of careful blocks (representing the stable block) will be in one of the following states:
    - both good and the same
    - both good and different
    - one good, one bad
- What does the recovery procedure do in each of the above cases ?

Summary (1 of 2)

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

Summary (2 of 2)

- Fault Tolerance is generally achieved through use of *redundancy* and *reliable multitasking protocols*.
- Processes, client/server and group communications can all be "enhanced" to tolerate faults in a distributed system. Commit protocols allow for fault tolerant multicasting (with *two-phase* the most popular type).
- *Recovery* from errors within a Distributed System tends to rely heavily on **Backward Recovery** techniques that employ some type of *check pointing* or *logging* mechanism, although **Forward Recovery** is also possible.

Interconnection Networks

Multiprocessors interconnection networks (INs) can be classified based on a number of criteria. These include

(1) Mode of operation (synchronous versus asynchronous),
(2) Control strategy (centralized versus decentralized),
(3) Switching techniques (circuit versus packet),
(4) Topology (static versus dynamic).

1- Mode of Operation

According to the mode of operation, INs are classified as synchronous versus asynchronous. In synchronous mode of operation, a single global clock is used by all components in the system such that the whole system is operating in a lock–step manner.

Interconnection Networks

Asynchronous mode of operation, on the other hand, does not require a global clock. Handshaking signals are used instead in order to coordinate the operation of asynchronous systems. While synchronous systems tend to be slower compared to asynchronous systems, they are race and hazard-free.

2- Control Strategy

According to the control strategy, INs can be classified as centralized versus decentralized. In centralized control systems, a single central control unit is used to oversee and control the operation of the components of the system. In decentralized control, the control function is distributed among different components in the system.

3- Switching Techniques

Interconnection networks can be classified according to the switching mechanism as circuit versus packet switching networks. In the circuit switching mechanism, a complete path has to be established prior to the start of communication between a source and a destination.

- In a packet switching mechanism, communication between a source and destination takes place via messages that are divided into smaller entities, called packets.
- On their way to the destination, packets can be sent from a node to another in a store-and-forward manner until they reach their destination.
- While packet switching tends to use the network resources more efficiently compared to circuit switching, it suffers from variable packet delays.

4- Topology

An interconnection network topology is a mapping function from the set of processors and memories onto the same set of processors and memories.

In other words, the topology describes how to connect processors and memories to other processors and memories.

A fully connected topology, for example, is a mapping in which each processor is connected to all other processors in the computer. A ring topology is a mapping that connects processor k to its neighbors, processors (k - 1) and (k + 1).

4- Topology

In general, interconnection networks can be classified as static versus dynamic networks.

In static networks, direct fixed links are established among nodes to form a fixed network, while in dynamic networks, connections are established as needed.

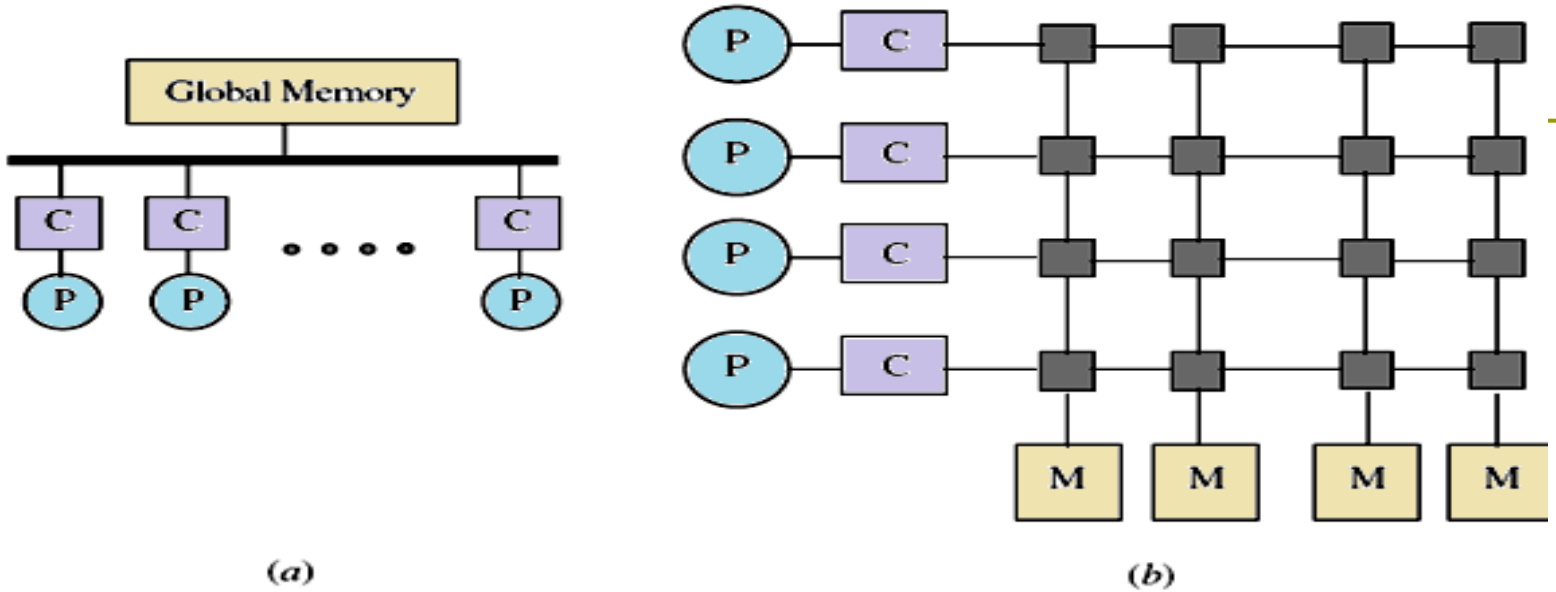
Switching elements are used to establish connections among inputs and outputs.

5- Memory Systems

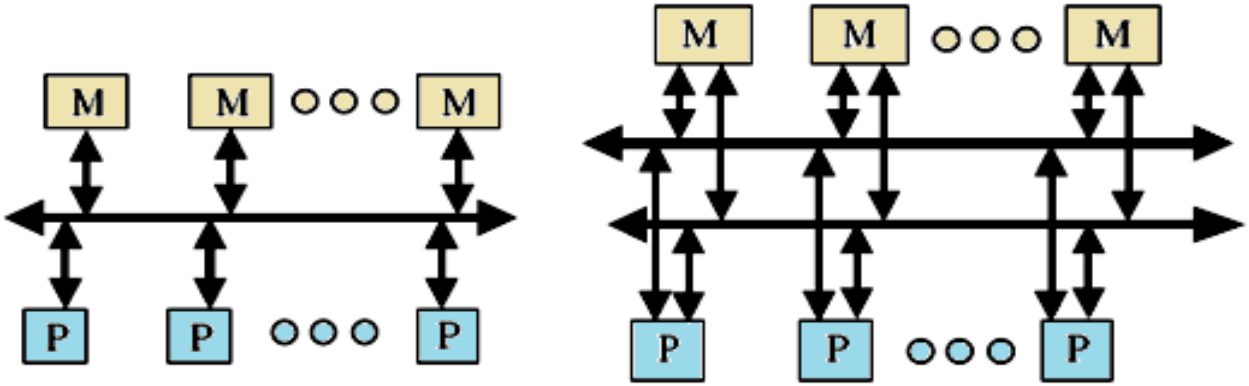
- Shared memory systems can be designed using bus-based or switch-based INs. The simplest IN for shared memory systems is the bus.
- However, the bus may get saturated if multiple processors are trying to access the shared memory (via the bus) simultaneously. A typical bus-based design uses caches to solve the bus contention problem.
- For example, a crossbar switch can be used to connect multiple processors to multiple memory modules. A crossbar switch, can be visualized as a mesh of wires with switches at the points of intersection.

5- Memory Systems

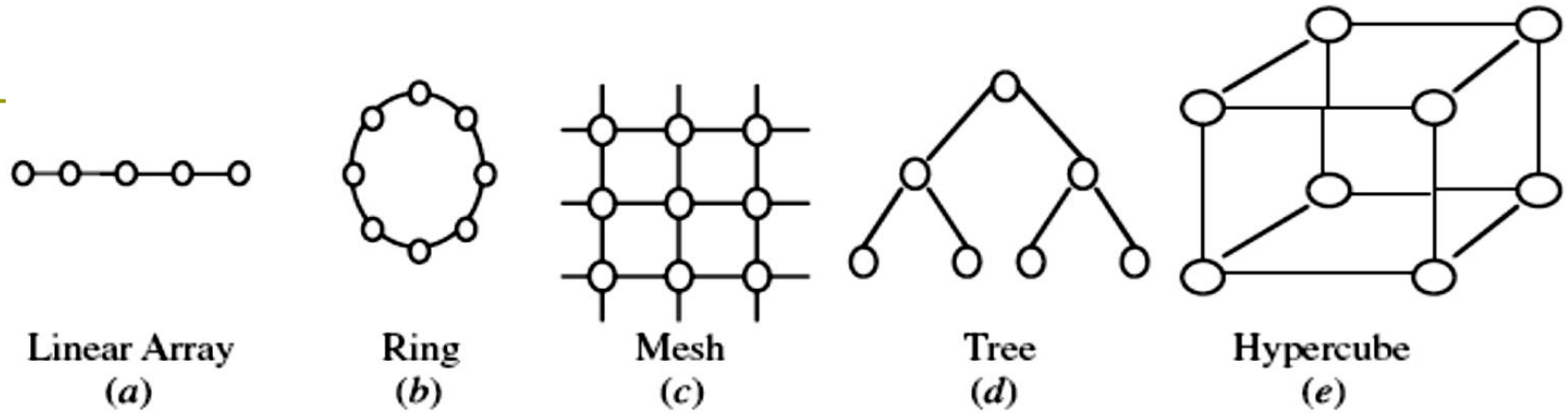
- Message passing INs can be divided into static and dynamic.
  - Static networks form all connections when the system is designed rather than when the connection is needed. In a static network, messages must be routed along established links.
  - Dynamic INs establish a connection between two or more nodes on the fly as messages are routed along the links. The number of hops in a path from source to destination node is equal to the number of point-to-point links a message must traverse to reach its destination.



Shared memory interconnection networks.
(a) bus-based and (b) switch-based shared mem



bus-based systems when a single bus is used versus the case when multiple buses are used

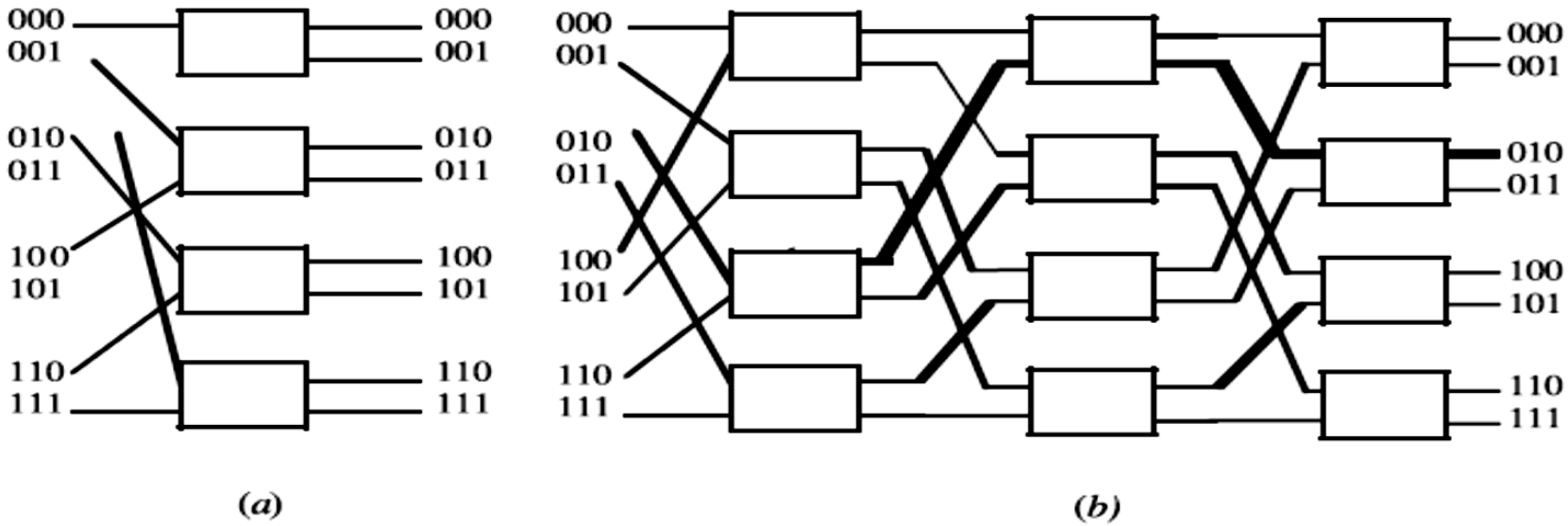


Examples of static topologies.

The single-stage interconnection network of Figure 1.10a is a simple dynamic network that connects each of the inputs on the left side to some, but not all, outputs on the right side through a single layer of binary switches represented by the rectangles. The binary switches can direct the message on the left-side input to one of two possible outputs on the right side. If we cascade enough single-stage networks together, they form a completely connected multistage interconnection network (MIN),

Dynamic INs:

(a)single-stage,
(b)multistage,



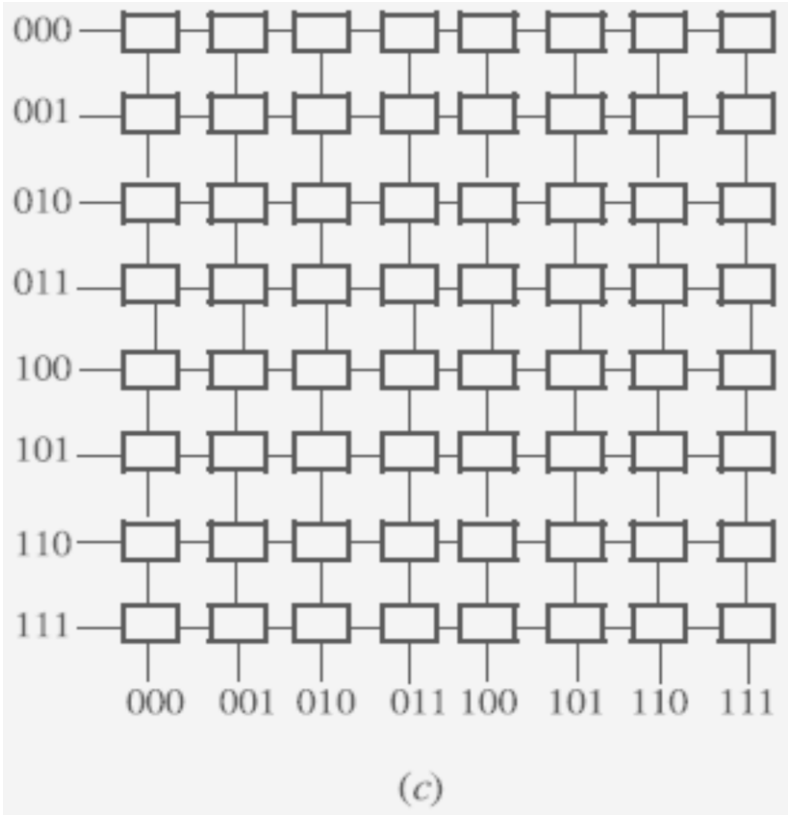
The omega MIN connects eight sources to eight destinations. The connection from the source 010 to the destination 010 is shown as a bold path.

Dynamic INs:

These are dynamic INs because the connection is made on the fly, as needed.
In order to connect a source to a destination, we simply use a function of the bits of the source and destination addresses as instructions for dynamically selecting a path through the switches.
For example, to connect source 111 to destination 001 in the omega network, the switches in the first and second stage must be set to connect to the upper output port, while the switch at the third stage must be set to connect to the lower output port (001).

(c) Crossbar switch

The crossbar switch of Figure c provides a path from any input or source to any other output or destination by simply selecting a direction on the fly. To connect row 111 to column 001 requires only one binary switch at the intersection of the 111 input line and 011 output line to be set. The crossbar switch clearly uses more binary switching components; for example, N^2 components are needed to connect NxN source/destination pairs. The omega MIN, on the other hand, connects NxN pairs with N/2(log N) components.



(c) Crossbar switch

The major advantage of the crossbar switch is its potential for speed. In one clock, a connection can be made between source and destination.

The diameter of the crossbar is one. (Note: Diameter, D, of a network having N nodes is defined as the maximum shortest paths between any two nodes in the network).

The omega MIN, on the other hand requires log N clocks to make a connection. The diameter of the omega MIN is therefore log N.

- Both networks limit the number of alternate paths between any source/destination pair.
- This leads to limited fault tolerance and network traffic congestion. If the single path between pairs becomes faulty, that pair cannot communicate.
- If two pairs attempt to communicate at the same time along a shared path, one pair must wait for the other. This is called blocking, and such MINs are called blocking networks. A network that can handle all possible connections without blocking is called a nonblocking network.

- TABLE shows a performance comparison among a number of different dynamic INs

Network	Delay	Cost (Complexity)
Bus	$O(N)$	$O(1)$
Multiple-bus	$O(mN)$	$O(m)$
MINs	$O(\log N)$	$O(N \log N)$

m = the number of multiple buses used,
N = the number of processors (memory modules) or input/output of the network.

- Table 3 shows a performance comparison among a number of static INs. In this table, the degree of a network is defined as the maximum number of links (channels) connected to any node in the network.

- The diameter of a network is defined as the maximum path, p, of the shortest paths between any two nodes. Degree of a node, d, is defined as the number of channels incident on the node.

Network	Degree	Diameter	Cost (#links)
Linear array	2	$N-1$	$N-1$
Binary tree	3	$2([\log_2 N]-1)$	$N-1$
n-cube	$\log_2 N$	$\log_2 N$	$nN/2$
2D-mesh	4	$2(n-1)$	$2(N-n)$