Unit-1: Introduction to Distributed Operating Systems

Dr. Satyananda Champati Rai
Associate Professor
School of Computer Engineering
KIIT Deemed to be University
Bhubaneswar

UNIT-1: Fundamentals of Distributed OS

- Introduction to DOS
- Goals of DOS
- Hardware Concepts
- Software Concepts
- Networked OS
- True Distributed Systems
- Time Sharing Multiprocessor OS
- Design Issues
- System Architectures

#1: Introduction to DOS

- Background:
  - 1945 - 1985 : Beginning of modern computer era
  - Computers were expensive
  - Most organizations had only a handful of computers
  - Lack of communication among the computers
  - Most computers were operated independently
- Development of two advanced technologies
  - Powerful microprocessors 8-bit, 16-bit, 32-bit and 64-bit
  - High-speed computer networks (LAN, MAN, WAN)
- Subsequently, computing systems composed of large number of CPUs connected by a high-speed network were formed. It is known as **"Distributed Systems"**.

#1: Introduction to DOS Cont.

- Distributed System:
  - A distributed system is **a collection of independent computers** that appear to the users of the system as a single computer.
  - i.e. A Distributed system has two aspects – Hardware & Software
    1. **Hardware**: Machines are autonomous
    2. **Software**: **The users think of the system as a single computer**
- **Eg #1**: Network of workstations in an organization
- Eg: Pool of processors in the machine room that are not assigned to specific users but are allocated as needed.
- It might have a single file system, with all files accessible from all machines uniformly and using the same path name.
- A single command may be executed on user's own workstation/idle workstation / unassigned processors in the machine room.
- If the system as a whole looked and acted like a classical single-processor timesharing system, it would qualify as a **"Distributed System"**.

#1: Introduction to DOS Cont.

- **Eg #2**: A factory full of robots, each containing a powerful computer for handling vision, planning, communication, and other tasks.
- All the robots act like peripheral devices attached to the same central computer and the system can be programmed that way, it too count as a distributed system.

#1: Introduction to DOS Cont.

- **Eg #3**: Consider large bank with hundreds of branch offices all over the world. Each office has a master computer to store local accounts and handle local transactions. Each computer has the ability to communicate to all other branch computers and with a central computer at headquarters. If (transactions can be done **without regard to where a customer** or **account** is AND the **user do not notice any difference between this system and the old centralized mainframe** that it replaced) then it is a **"Distributed System"**.

#2: Motivation and Goals of Distributed Systems

- **Q**. Why a Distributed System is essential?
  - **Reasons**: Let us analyze the reasons of using Distributed System.
    1. **Economics:** The real driving force behind the trend toward decentralization is economics. Use of large number of chip CPUs together in a system potentially have a much better price/performance ratio than a single large centralized system would have.
    2. **Speed**: A collection of microprocessors cannot only give a better price/performance ration than a single mainframe, but may yield an absolute performance than no mainframe can achieve at any price. Multiple interconnected CPUs work together.
    3. **Inherent Distributed Application**: Eg- A supermarket chain. The completer system look like a single computer to the application programs, but implement decentrally, with one computer per store. This is a commercial distributed system. Eg #2: Computer Supported Cooperative Game. Eg #3: Cloud-based environment
    4. **Reliability**: if k% of the machines are down at any moment, the system should be able to continue to work with a k% loss in performance. For critical applications, such as control of nuclear reactor or aircraft using a DOS to achieve high reliability may be the dominant consideration.
    5. **Incremental Growth**: With a distributed system, we may add more processors to the system, thus allowing it to expand gradually as the need arises.
  - **Note: Grosch's law**: Computing power of CPU $\propto$ square of its price

Advantages of Distributed Systems over Independent PCs

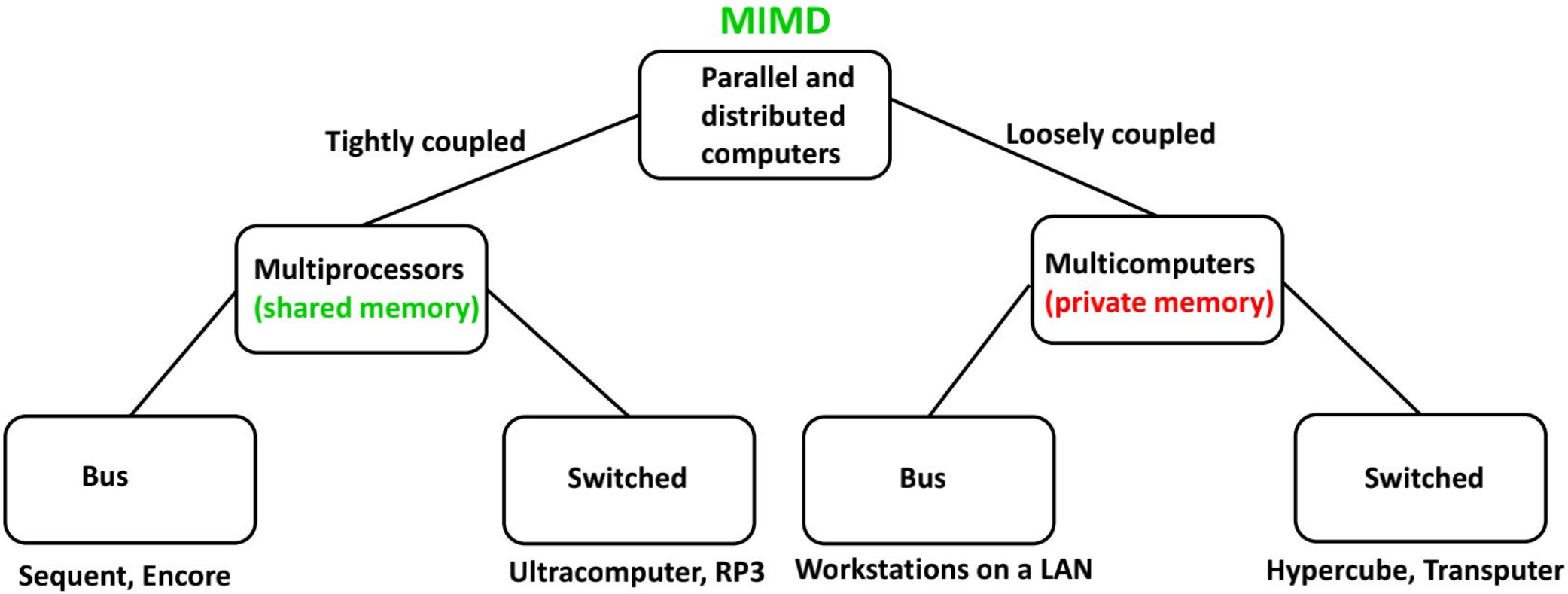
- **Data Sharing**: Allow many users access to a common data base
- **Device Sharing**: Allow many users to share expensive peripherals like color printer
- **Communication**: Make human-to-human communication easier (Eg: Email)
- **Flexibility**: Spread the work load over the available machines in the most cost effective way

Disadvantages of Distributed Systems

- **Software** : More software are required for different distributed applications for smooth functioning across the locations
- **Networking** : The network can saturate or cause other problems
- **Security**: Easy access also applies to secret data

#3: Hardware Concepts

- The multiple CPUs in distributed System - How the CPUs are interconnected and how they communicate ?



#3:Hardware Concepts Cont.

- Flynn's Archtecture (1972) – **Two** essential characteristics :
  - The number of instruction streams
  - The number of data streams
  - SISD : Eg. All traditional uniprocessor computers (i.e., those having **only one CPU**)
  - SIMD : Eg. Array processors (**Super Computers**) – one instruction unit that fetches an instruction, and then commands many data units to carry out in parallel, each with its own data.
  - MISD : No known computers fit this model
  - MIMD : Eg: A group of independent computers, each with its own program counter (PC), program, and data. **All distributed systems** are MIMD.

Difference between Multiprocessors and Multicomputers

Sl. (Parameters)	Multiprocessors	Multicomputers
1. Memory	Single virtual address space that is shared by all CPUs	Every machine has its own private memory
2a. Bus Architecture	There is a single network, backplane, bus, cable, or other medium that connects all the machines	
2b. Switched Architecture	There are individual wires from machine to machine, with many different wiring patterns in use. Eg. Worldwide public telephone system	
3. Coupling	Tightly coupled	Loosely coupled
4. Communication time	Short	Long
5. Data rate	High	Low

#3.1: Bus-Based Multiprocessors

- It consist of k≥2 number of CPUs all connected to a common bus, along with a memory module. A high-speed backplane or motherboard into which CPU and memory cards can be inserted. A typical **bus has 32 or 64 address lines**, **32 or 64 data lines** and **32 or 64 control lines**, all of which operate **in parallel**.

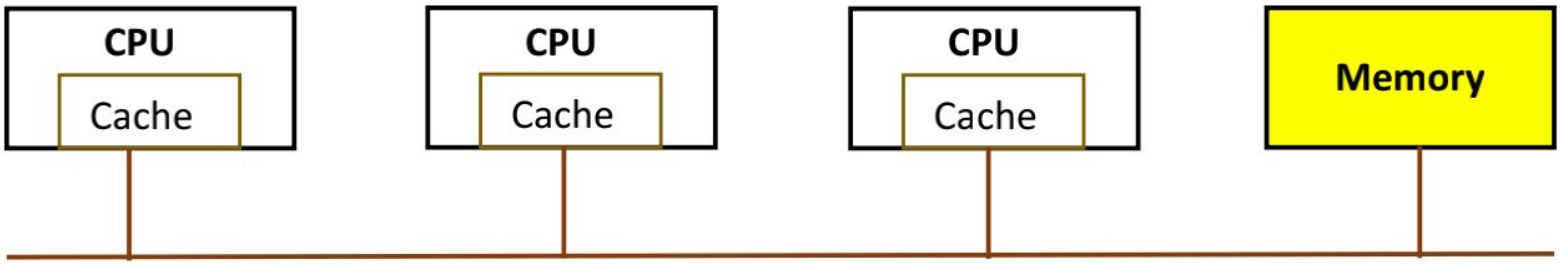


Fig: A bus-based multiprocessor

#3.1: Bus-Based Multiprocessors Cont.

- Characteristics:
  - Coherent memory
  - Bus is overloaded even with 4 or 5 CPUs, and performance will drop drastically
    1. Solu^{n}: Add a high-speed cache memory between the CPU and the bus.
    2. Cache sizes of 64K to 1M are used in general provides hit rate of 90%
    3. Write-through-cache
    4. Snooping-cache
    5. Snoopy-write-through-cache (combination of #3 and #4)

#3.2: Switched Multiprocessors

- When number of processors > 64, then bus-based multiprocessor is not suitable, in that case "Crossbar switch" or "omega switching" network is used.

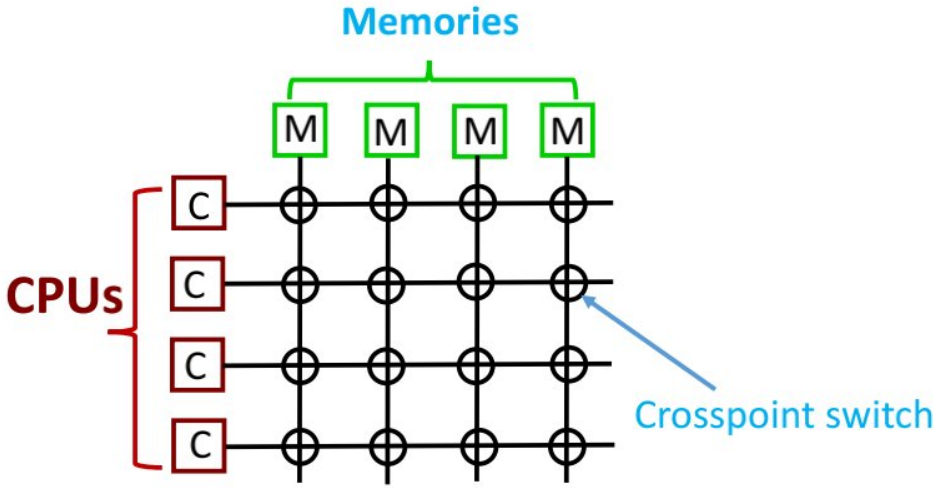


Fig. (a) A Crossbar switch

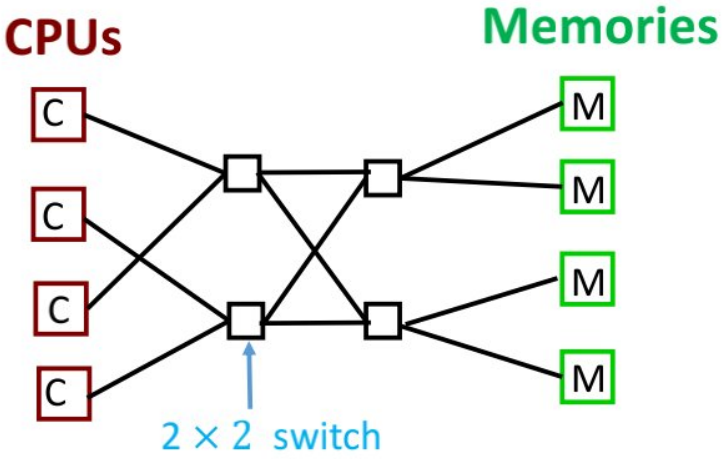


Fig. (b) An **Omega** switching network

#3.2: Switched Multiprocessors cont.

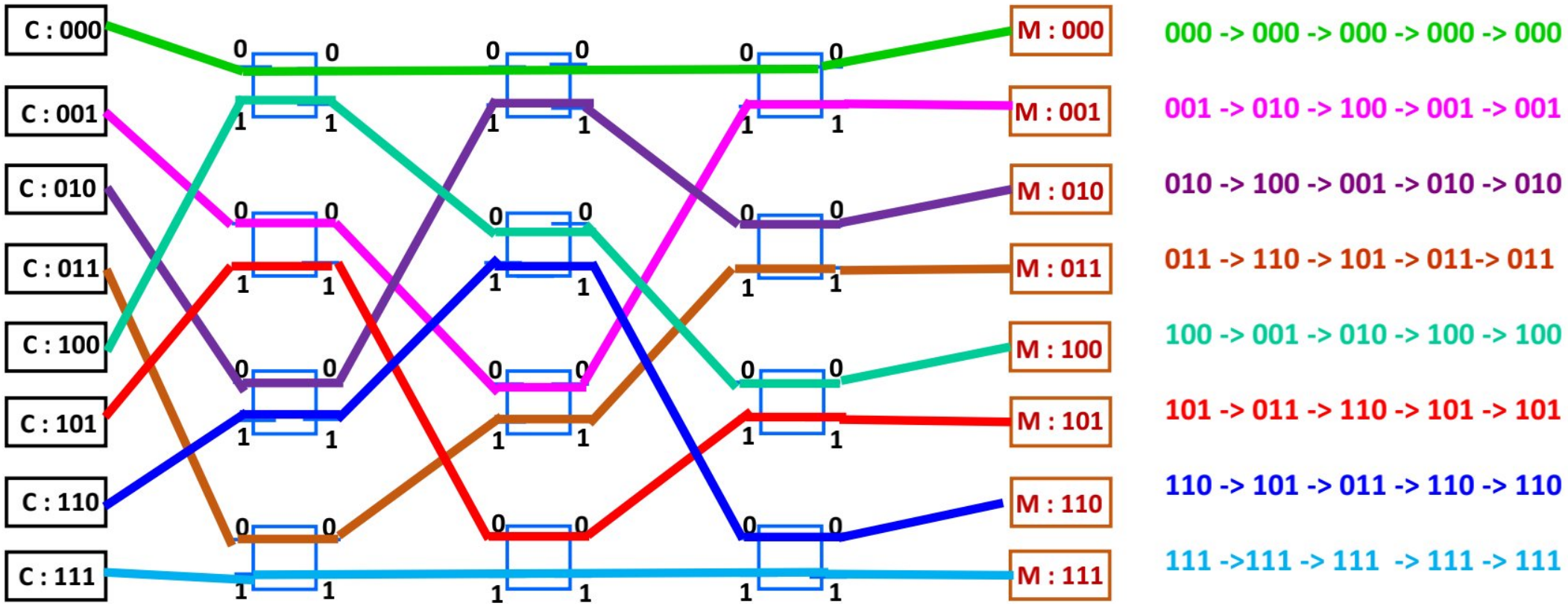
- Characteristics: Memory is divided into modules and connect them to the CPUs with a crossbar switch. (a tiny electronic crosspoint switch).
- A CPU can access a memory after **closing** the crosspoint switch
- If two CPUs try to access the same memory simultaneously, one of them will have to wait.
- Limitation (crossbar switch):
  - $\mathbf{n^2}$ crosspoint switches are required with n CPUs and n memories. **(Maximum 64 CPUs)**
- Remedy:
  - **Omega Network** (it contains four 2 × 2 switches, each having two inputs and two outputs)
  - With n CPUs and n memories, the omega network requires $\mathbf{\log_2 n}$ switching stages, each containing **n/2** switches, for a total of $\mathbf{(n \log_2 n)/2}$ switches.
- **Question**: Construct omega network for 8 computers with 8 memory. Explain the communication among any pair of computer to memory. How the conflict is resolved?

Problem: Omega Switched Network (8 × 8)

- **Question:** Construct 8 × 8 omega network for communication between 8 computers and 8 memory locations
- **Ans:** There will be $\log_2(8)$= 3 stages of switches
- Each stage has 8/2= 4 number of switches
- Total number of switches = 12
- In the switch the upper part is for 0 bit and lower part is for 1 bit
- For connection it must be rotate-left-shift operation. It is as follows
- 000 -> 000 -> 000 -> 000 -> 000
- 001 -> 010 -> 100 -> 001 -> 001
- 010 -> 100 -> 001 -> 010 -> 010
- 011 -> 110 -> 101 -> 011-> 011
- 100 -> 001 -> 010 -> 100 -> 100
- 101 -> 011 -> 110 -> 101 -> 101
- 110 -> 101 -> 011 -> 110 -> 110
- 111 ->111 -> 111 -> 111 -> 111

Problem: Omega Switched Network (8 × 8) cont.

- Explanation: 100 -> 001 -> 010 -> 100 -> 100 (rotate-left-shift operation for switches)



#3.2: Switched Multiprocessors cont.

- Limitations of Omega Switching Network
  - **Slow** : Because of multi-stage to-and-from communication
  - **Costly** : Due to high performance switches
- **Example**: n=1024, No. of switching stages=$\log_2(1024)=10$, To-and-fro stages=10 + 10 = 20, CPU is of modern RISC with chip running at 100 MIPS, So instruction execution time is = 10 nsec. The switching time must be 500 picosec (0.5 nsec). The complete multiprocessor required = log2(1024) * (1024/2) = 5120 number of 2 × 2 switches with 500-picosec switching capability. Hence **costly.**
- **Conclusion**: It is possible to build a large, tightly-coupled, shared memory multiprocessor, but it would be difficult and expensive.

#3.3: Bus-Based Multicomputers

- In bus-based multicomputer system it is required to communicate CPU-to-CPU. It need not be a high-speed backplane bus.
- It is a collection of workstations on a LAN, where each system has its own local memory. No shared memory.
- Speed is 10-100 Mbps

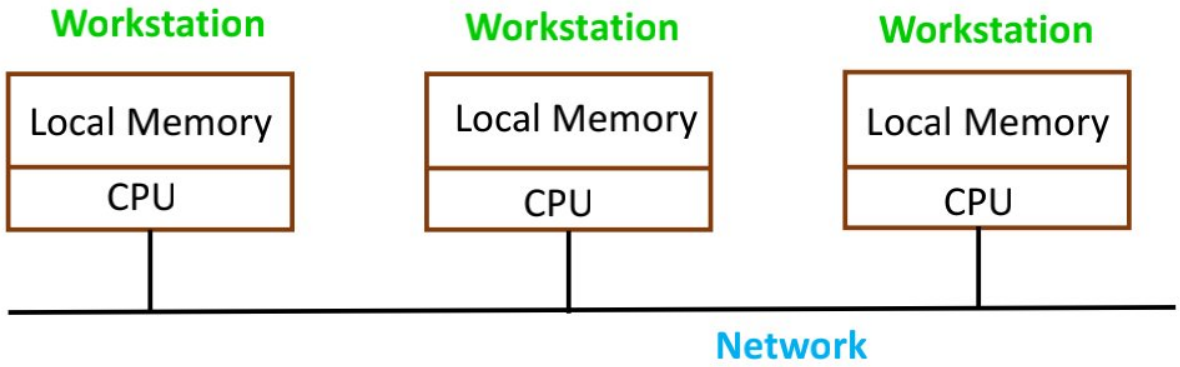


Fig. A multicomputer consisting of workstations on a LAN

#3.4: Switched Multicomputers

- Each CPU has direct and exclusive access to its own, private memory.
- Two popular topologies are:
  - Grid based switched multicomputer
  - Hypercube based switched multicomputer

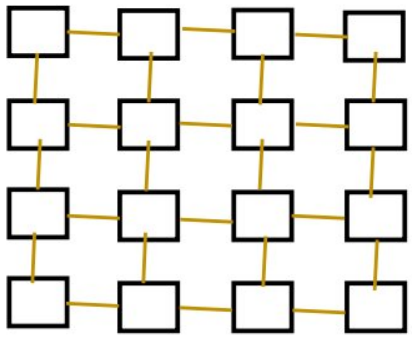


Fig: A Grid-based switched multicomputers

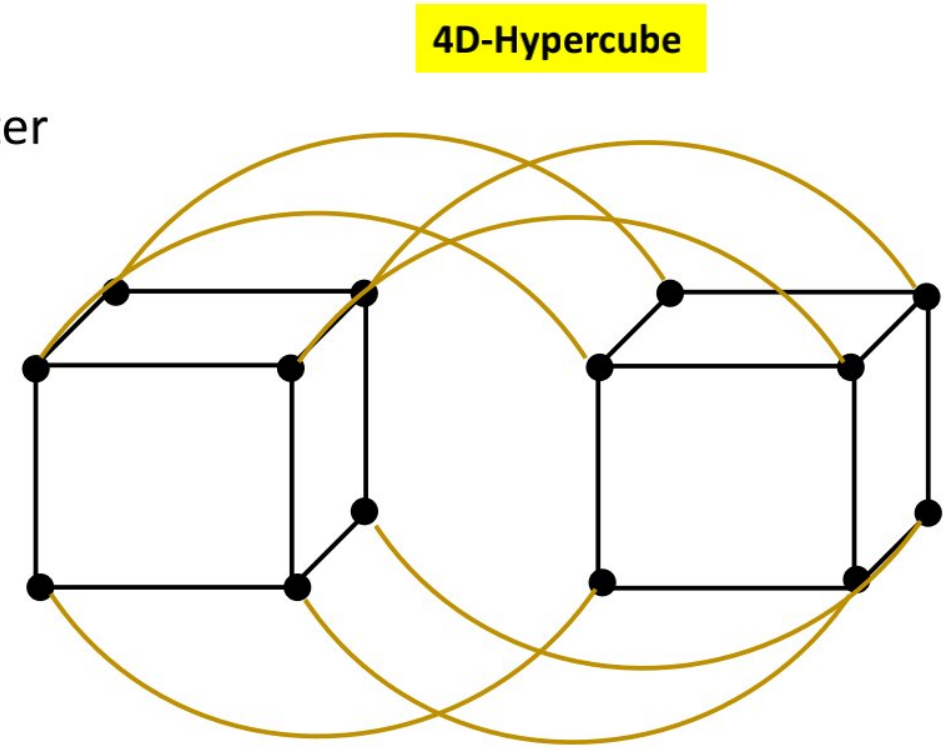


Fig: A Hypercube-based switched multicomputers

#4: Software Concepts

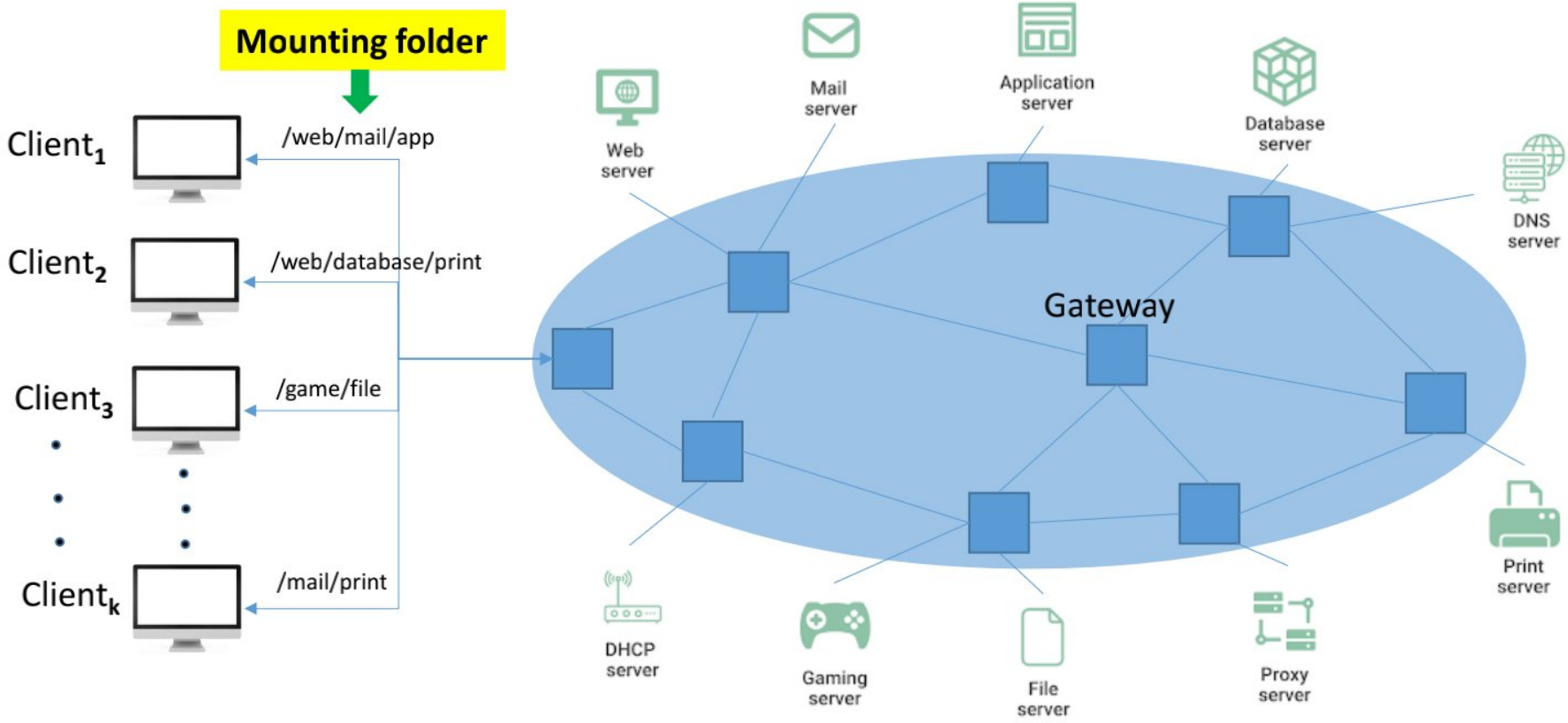
- Two types of OSs for multiple CPU systems
  - Loosely-coupled:
    1. A group of PCs, each has its own CPU, its own memory, Its own HDD, its own OS, but shares some resources, such as printers, databases, over a LAN/MAN/WAN.
    2. i.e. All the PCs are independent of one another.
    3. All the PCs are connected via some **network**
  - Tightly-coupled:
    1. All CPUs are at one place
    2. The microprocessors execute in **parallel**
- --

#5: Networked Operating Systems

- A network of workstations, must have their own OS, may have own HDD, connected by a LAN and execute all commands locally.
- Provision of one or more file sever(s) across the network
- Sometimes a user may log into another workstation remotely by using remote login command :
  - **$** rlogin IP_Address_of_Machine
- It also allows copy of any file from one machine to another machine:
  - **$ rcp machine_source/file1 machine_destination/file2**
- The **operating system** that is used in this kind of environment must manage the individual workstations and file servers and take care of the communication between them.

#5: Networked Operating Systems Cont.

- Accessing different servers by different clients through mounting



#6: True Distributed Systems: Fundamental Properties

- A distributed system is one that runs on a collection of networked machines but acts like **a virtual uniprocessor**.
- There must be a single, **global interprocess communication** mechanism so that any process can talk to any other process.
- Local and remote communication **must be same**.
- There must be a **global protection** scheme.
- **Process management** must also be same everywhere
- There must be **a single set of system calls** available in all machines and must make sense in distributed environment
- The **file system** must look same everywhere too. Every file should be visible at every location, subject to protection and security constraints.
- **Identical kernels** must run on all the CPUs in the system.

#7: Timesharing Multiprocessor system

- Tightly-coupled software on tightly-coupled hardware
- **Key characteristics**: Existence of a single run queue – A list of all the processes in the system that are logically unblocked and ready to run. A run queue is a **data structure** kept in the shared memory.
- The methods used on the multiprocessor to achieve the appearance of a virtual uniprocessor are not applicable to machines that do not have shared memory.
- **Eg**: Dedicated database machines

#7: Timesharing Multiprocessor system Cont.

- 3 CPUs and 5 processes that are ready to run. All the 5 processes are in the shared memory and 3 of them are currently running- A in CPU_1, B in CPU_2 and C in CPU_3 Speed is 10-100 Mbps
- A, B, C are executing, D & E are ready to run
- Mutual exclusion is achieved by Semaphore, Monitor
- If(currently running process has **I/O < θ**)
  - Then OS makes other process **busy waiting**

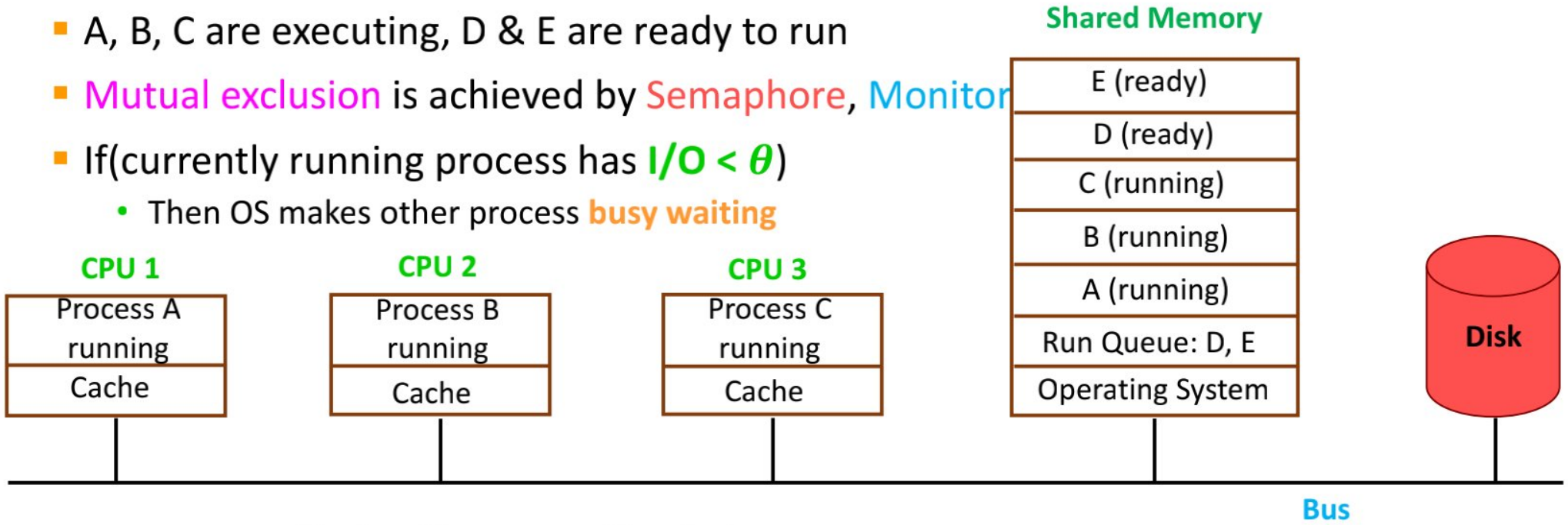


Fig. A multiprocessor with a single run queue

#7: Timesharing Multiprocessor system Cont.

- Comparison among three kinds of systems

Parameter	Network OS	Distributed OS	Multiprocessor OS
Does it look like a virtual uniprocessor	No	Yes	Yes
Do all have to run the same OS?	No	Yes	Yes
How many copies of the OS are there?	N	N	1
How is communication achieved?	Shared files	Messages	Shared memory
Are agreed upon network protocols required ?	Yes	Yes	No
Is there a single run queue?	No	No	Yes
Does file sharing have well-defined semantics?	Usually no	Yes	Yes

Fig: Comparison of three different ways of organizing n CPUs

#8: Design Issues: How to Build a DOS?

- How to maintain Transparency?
- How to make it Flexible?
- How to ensure Reliability?
- How to achieve desired Performance?
- How to adopt millions & billions of systems altogether to scale up?

#8.1: How to maintain Transparency in DOS?

- How to achieve the single-system image?
- How do the system designers fool everyone into thinking that the collection systems is simply an old-fashioned timesharing system? i.e. the working details about the systems must be hidden to the users.
- How to achieve transparency?
  - Hide the distribution from the users (It is a higher level work, easy to do)
    1. **$ make file_name // To recompile a large number of files in a directory**
  - Design the system call interface so that the existence of multiple processors is not visible. (It is a lower level work, difficult to achieve)
- **Transparency means** : Hiding all the distribution from the users and even from the application programs.

#8.1.1: Different Types of Transparency

1) **Location Transparency :** The users cannot tell where the resources are located ***(Eg: Location of Servers, PCs, Printers, Files, data bases, S/Ws not known to the users)***
2) **Migration Transparency:** Resources can move at will without changing their names. ***(Eg. File, Directory, S/Ws can be transferred from one server to other server without change of name)***
3) **Replication Transparency:** The users cannot tell how many copies exist ***(Eg. Server is free to replicate the heavily used files and provide the service without the knowledge of the users)***
4) **Concurrency Transparency:** Multiple users can share resources automatically. ***(Eg. If two or more users are accessing same resource, then one should not see others access)***
5) **Parallelism Transparency:** Activities can occur in parallel without the knowledge of the users. **(Eg. Programmers must not know the number of CPUs available in the whole system)**

#8.2: How to make DOS Flexible?

- **Monolithic kernel** (Eg. Centralized OS augmented with networking facilities and the integration of remote services)or **Microkernel** (Eg. Small OS with specific services)?
- Flexibility + Transparency = DOS

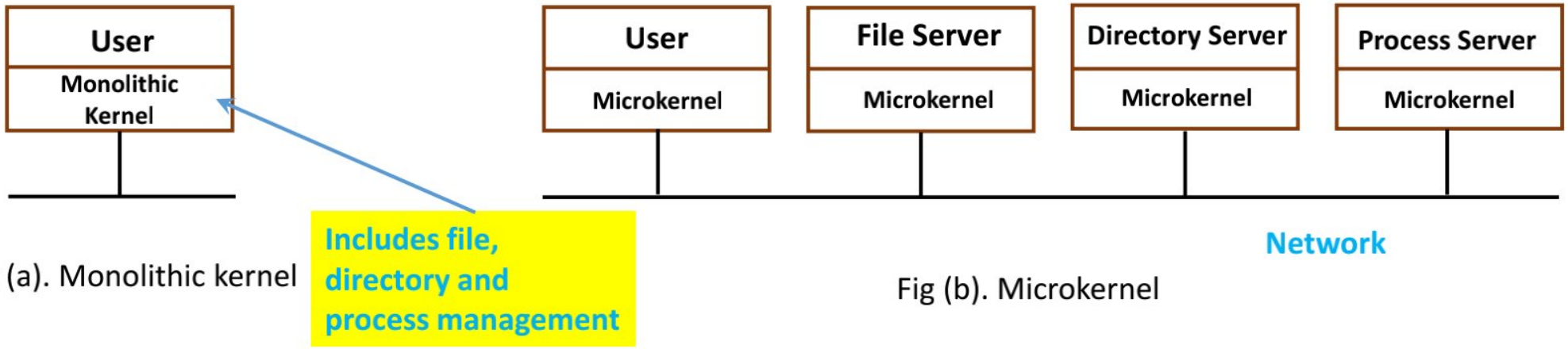


Fig (a). Monolithic kernel

Fig (b). Microkernel

#8.2 Monolithic kernel Vs Microkernel

Monolithic Kernel

- Traditional kernel that provides most services itself.
- It is less flexible
- Centralized OS, augmented with networking facilities and integration of remote services
- The system calls trap the kernel
- Own disks and local files
- DOS that are extension of UNIX OS use this approach
- Eg. **Sprite OS**

Microkernel

- Kernel should provide as little as possible; bulk services available from user level servers.
- It is more flexible & highly modular
- It provides four services – IPC, Memory management, Process management, and scheduling, Low-level I/O
- It does not provide the file system, directory system, process management
- Other services are user level service
- It is highly modular
- Eg. **Amoeba OS**

#8.3: Reliability

- **Availability** of a system is defined as the fraction of time that the system is usable.
- Tool to improve availability is **redundancy** – i.e. keep h/w and s/w replicas, so that if one of them fails the other will take over its place.
- A highly reliable system must be highly available, but that is not enough. **No data loss**, **not grabled**, **consistent across**, **security**, **protection** from unauthorized access.
- System should be **fault tolerance** (Capability to mask failures)
- All the separate services should be arranged in such a manner that it should **not** add substantial **overhead** to the system.

#8.4: Performance of DOS

- Response Time
- Throughput
- System utilization
- Network capacity consumed
- Communication overhead **dwarfs** the extra CPU cycles gained
- **Fine-grained parallelism** (Large number of small computations)
- **Coarse-grained parallelism** (Large computations, low interaction rates, and little data)
- Cost time

#8.5: Scalability

- IPV4 ---- 2^{32} : Number of unique IP addresses for unique systems
- IPV6 ---- $2^{128} \approx 3.4 \times 10^{38}$: Number of unique systems to accommodate IoT
- Centralized database (without mirror) are almost as bad as centralized components – vulnerable to failure
- Any algorithm that operates by collecting information from all sites, sends it to a single machine for processing, and then distributes the results **must be avoided**.
- Only decentralized algorithms should be used.
- The **characteristics of these algorithms** are
  - No machine has complete information about the system state
  - Machines make decisions based only on local information
  - Failure of one machine does not spoil the algorithm
  - There is **no** implicit assumption that **a global clock exist**

8.5: Scalability cont.

- **Netflix's** own custom global CDN (Content delivery network)
- Distributed GIS technology enables modern online mapping systems (such as **Google Maps** and Bing Maps), Location-based services (LBS), web-based GIS (such as ArcGIS Online) and numerous map-enabled applications.
- Examples of distributed OS are **Solaris**, **AIX**, **OSF**, etc.
- Messaging systems like **WhatsApp** are known as distributed systems. They are designed to operate on a distributed infrastructure, allowing messages to be sent and received across multiple servers and devices.
- With **AWS** High-Performance Computing (HPC), you can accelerate innovation with fast networking and virtually unlimited **distributed computing** infrastructure.
- **Facebook** uses thousands of distributed systems and microservices to power their ecosystem. In order to communicate with each other, these microservices rely on a message queue. Facebook Ordered Queueing Service (**FOQS**) is an internal Facebook tool that fills that role.

#8.5 Scalability Cont.

- **Dropbox's key management** infrastructure is designed with operational, technical, and procedural security controls with very limited direct access to keys. Encryption key generation, exchange, and storage is distributed for decentralized processing.
- **Spotify** uses a distributed network of servers to store and deliver music and podcasts to its users. The company has multiple server locations worldwide, which work together to ensure the content is provided quickly and reliably.
- Social media applications, such as **Facebook**, **X**, and **Instagram**, are designed to be accessed by users from different devices and locations, and their data and processing are distributed across multiple servers and data centers.

8.5: Scalability Cont.

- **Azure** Service Fabric is an example of a distributed systems platform that is designed specifically for building microservices-based applications.
- **MongoDB** is a general-purpose, document-based, distributed database management system built for modern application developers.
- The **Hadoop** distributed file system acts as the master server and can manage the files, control a client's access to files, and overseas file operating processes such as renaming, opening, and closing files.
- **Bitcoin** is a secure, distributed system that manages consensus about the state of accounts and the authorized transactions among them.

8.5: Scalability Cont.

- **Git** is a Distributed Version Control System (DVCS) used to save different versions of a file (or set of files) so that any version is retrievable at will
- **Zoom** builds its own distributed cloud-native infrastructure. Cloud-Native means the system is architected to use cloud technology from the ground up. The microservices allow developers to seamlessly grow capacity.

Q&A

Homework Questions

1) A multicomputer with 256 CPUs is organized as a 16 × 16 grid. What is the worst-case delay (in hops) that a message might have to take?
2) Consider a 256- CPU hypercube. What is the worst-case delay in terms of hops?
3) A multiprocessor ha 4096 50-MIPS CPUs connected to memory by an omega network. How fast do the switches have to be allow a request to go to memory and back in one instruction time?
4) An experimental file server is up 3/4 of the time and down 1/4 of the time, due to bugs. How many times does this file server have to be replicated to give an availability of at least 99 percent?
5) Show the communication path for sending a message from computer 3 to memory 6 in a 8×8 omega network.
6) Explain the principle to avoid any conflict in a 8×8 omega network.
7) Construct a 16×16 omega network and explain its connection mechanisms.

Thank You!

Unit-2
Communication in Distributed Systems

Dr. Satyananda Champati Rai
Associate Professor
School of Computer Engineering
KIIT Deemed to be University
Bhubaneswar

UNIT-2: Communications in Distributed Systems

- Basics of Communication Systems
- Layered Protocols
- ATM Models
- Client-Server Model
- Blocking and Non-Blocking Primitives
- Buffered and Un-Buffered Primitives
- Reliable and Unreliable Primitives
- Message passing
- Remote Procedure Calls

#1. Basics of Communication Systems

- **Issues** : In a uniprocessor, most interprocess communication implicitly assumes the existence of shared memory. Eg: Producer-Consumer
- In Distributed system there is **no** shared memory
- **Protocols:** These are the rules adhere by the communicating processes
- In Wide-Area Distributed System: These protocols have multiple layers
- Platforms for Interprocess Communication
  - Open Systems Interconnection (OSI)
  - Asynchronous Transfer Mode (ATM)
  - Remote Procedure Calls (RPC)
  - Group Communication

#2. Layered Protocols

- **Background** : Due to the absence of shared memory, all communication in distributed systems is based on message passing.
- Communication Network:
  - DATA COMMUNICATIONS
  - NETWORKS
  - NETWORK TYPES
  - **PROTOCOL LAYERING**
  - **TCP/IP PROTOCOL SUITE**
  - **THE OSI MODEL**

Types of Connection

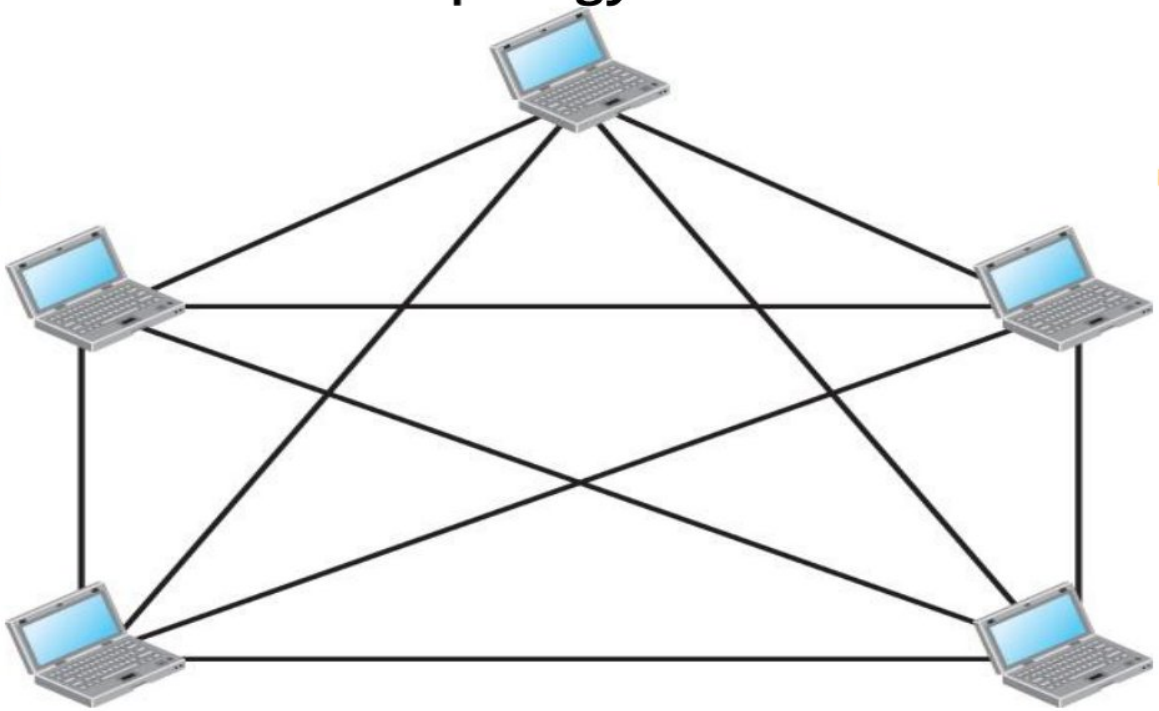
- A network is two or more devices connected through links.
- A link is a communications pathway that transfers data from one device to another.
- There are two possible types of connections: point-to-point and multipoint
- Point-to-point communication is a method in which the channel of communication is shared only between two devices or nodes.
- Multi-point communication is a form of communication in which the channel is shared among multiple devices or nodes.
- Bus Topology is a common example of Multipoint Topology.

Physical Topology

- The term physical topology refers to the way in which a network is laid out physically.
- Two or more devices connect to a link; two or more links form a topology.
- The topology of a network is the geometric representation of the relationship of all the links and linking devices (usually called nodes) to one another.
- There are four basic topologies possible: mesh, star, bus, and ring.

A fully connected mesh topology

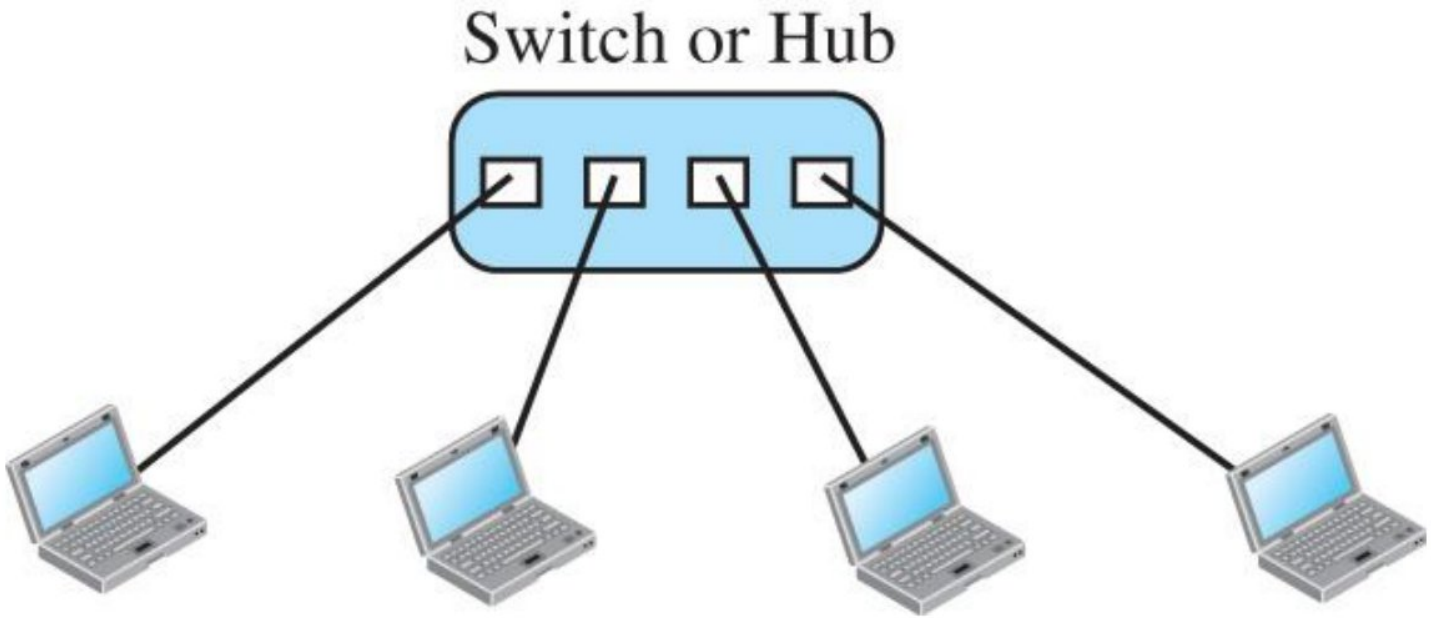
- n = 5



- 10 links

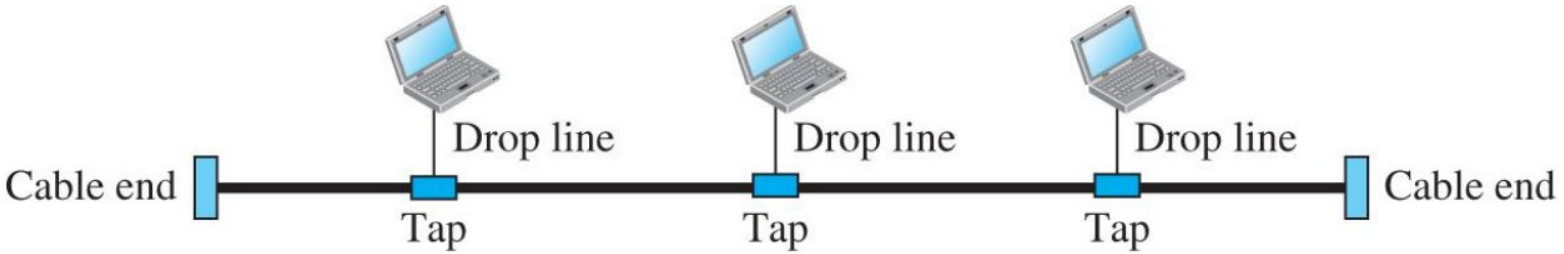
- Suppose, N number of devices are connected in a mesh topology, then the total number of dedicated links required to connect them is $\mathbf{Nc}_2$ i.e. **N(N-1)/2.**
- In Figure 1.4, there are 5 devices connected, hence the total number of links required is 5*4/2 = 10

A star topology



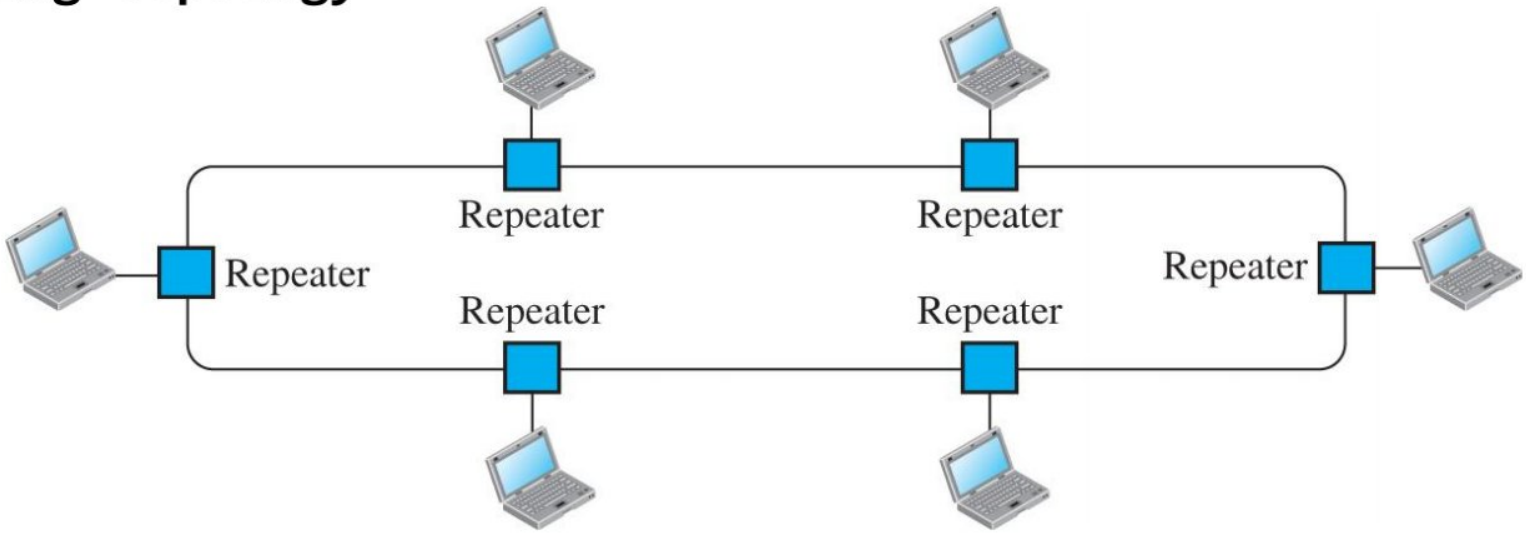
- A Star topology is a type of network topology in which all the devices or nodes are physically connected to a central node such as a router, switch, or hub.
- The central node (hub) acts as a server, and the connecting nodes act as clients.

A bus topology



- Bus topology, alternatively known as line topology, is a type of network topology where all devices on a network are connected to a single cable, called a bus or backbone.
- This cable serves as a shared communication line, allowing all devices (computers, printers, etc.) to receive the same signal simultaneously.

A ring topology

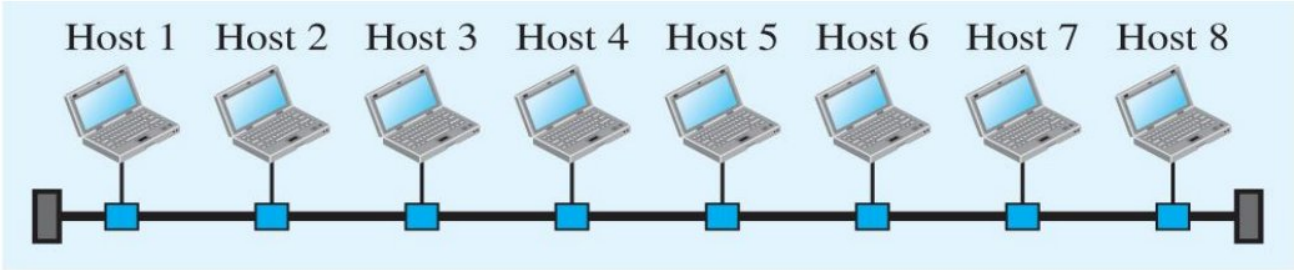


- Ring topology is a network configuration where devices are connected in a circular structure, forming a closed loop.
- In this topology, each device is connected to exactly two other devices, one on either side, creating a single continuous pathway for data transmission.
- Data travels in only one direction around the ring, passing through each device until it reaches its destination.

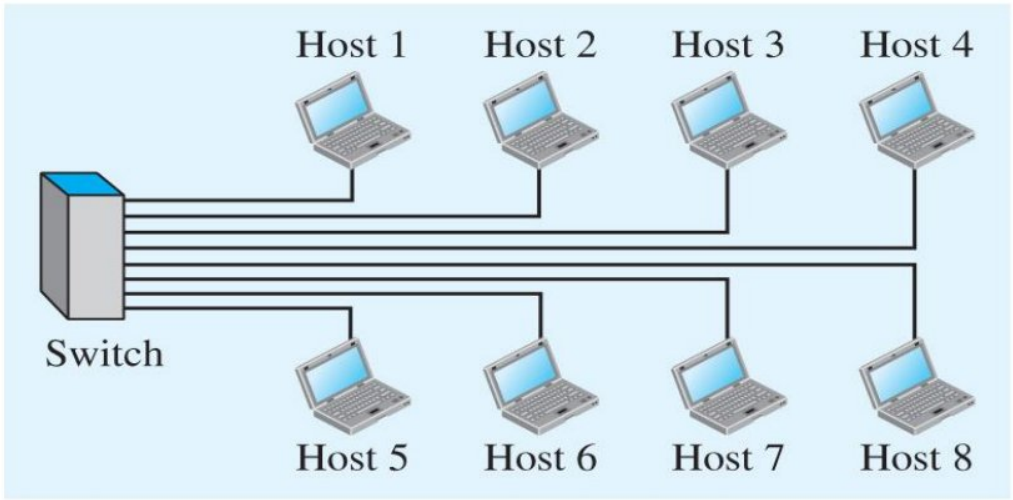
Local Area Network (LAN)

- A local area network (LAN) is a collection of devices connected together in one physical location, such as a building, office, or home.
- A LAN can be small or large, ranging from a home network with one user to an enterprise network with thousands of users and devices in an office or school.
- A LAN comprises cables, access points, switches, routers, and other components that enable devices to connect to internal servers, web servers, and other LANs via wide area networks.

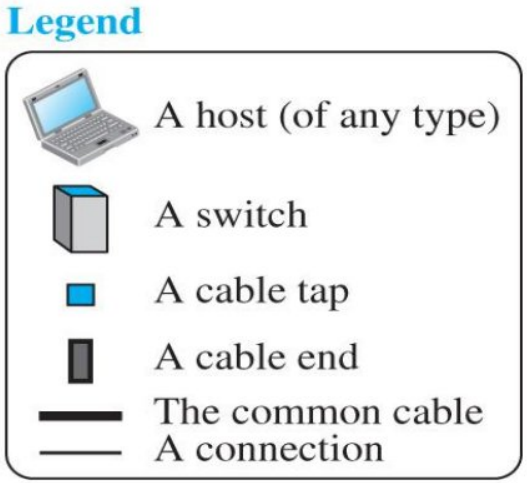
An isolated LAN in the past and today



a. LAN with a common cable (past)



b. LAN with a switch (today)



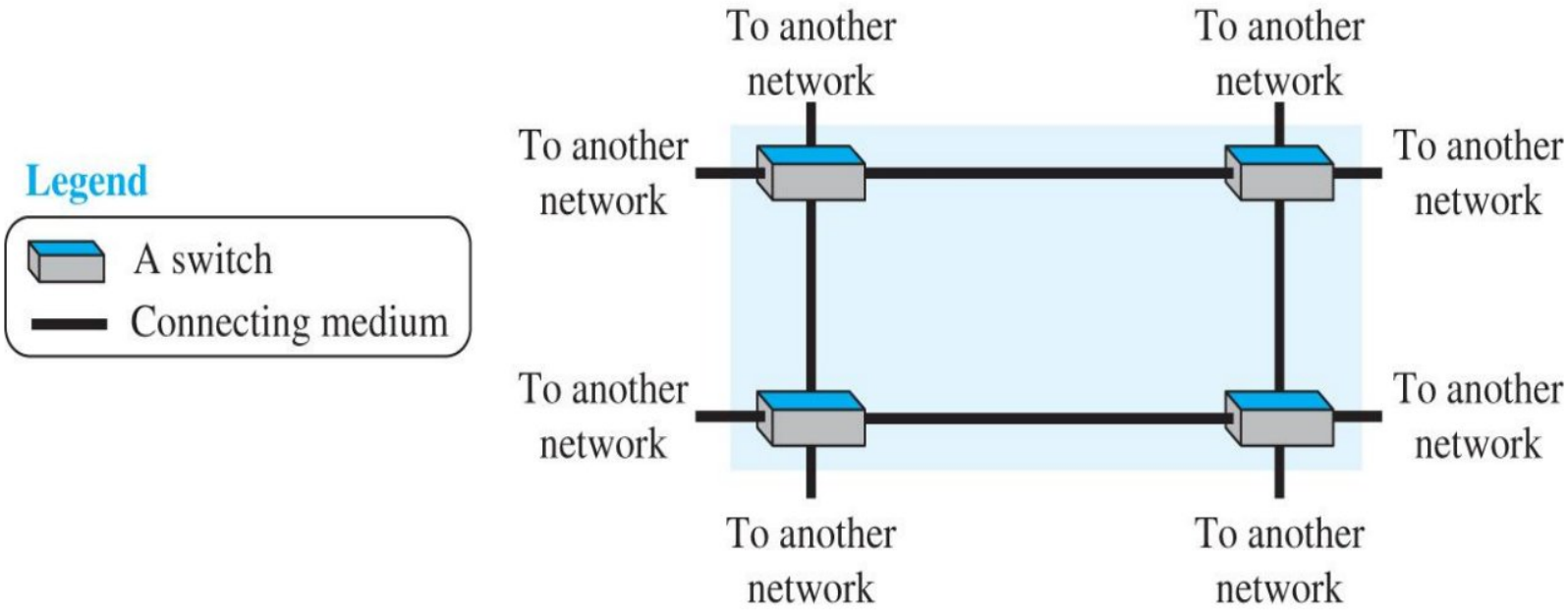
Wide Area Network (WAN)

- A wide area network (WAN) is also a connection of devices capable of communication.
- In its simplest form, a wide-area network (WAN) is a collection of local-area networks (LANs) or other networks that communicate with one another. A WAN is essentially a network of networks, with the Internet the world's largest WAN.
- Types of WAN technologies:Packet switching,TCP/IP protocol suite, RouterOverlay network(network virtualization)..

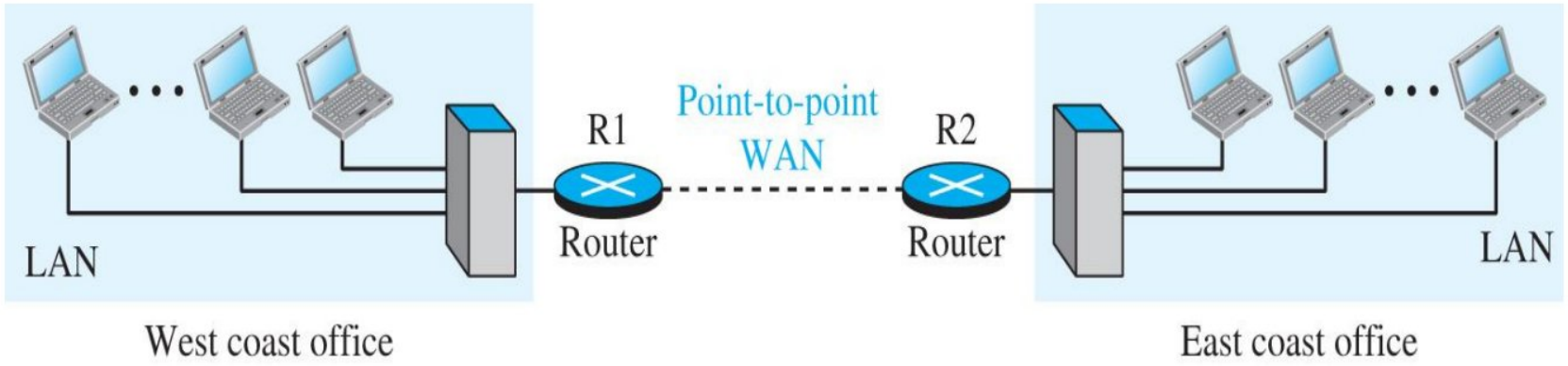
Switched WAN

- A switched WAN is a network with more than two ends.
- A switched WAN is used in the backbone of global communication today.
- Figure 1.10 shows an example of a switched WAN.

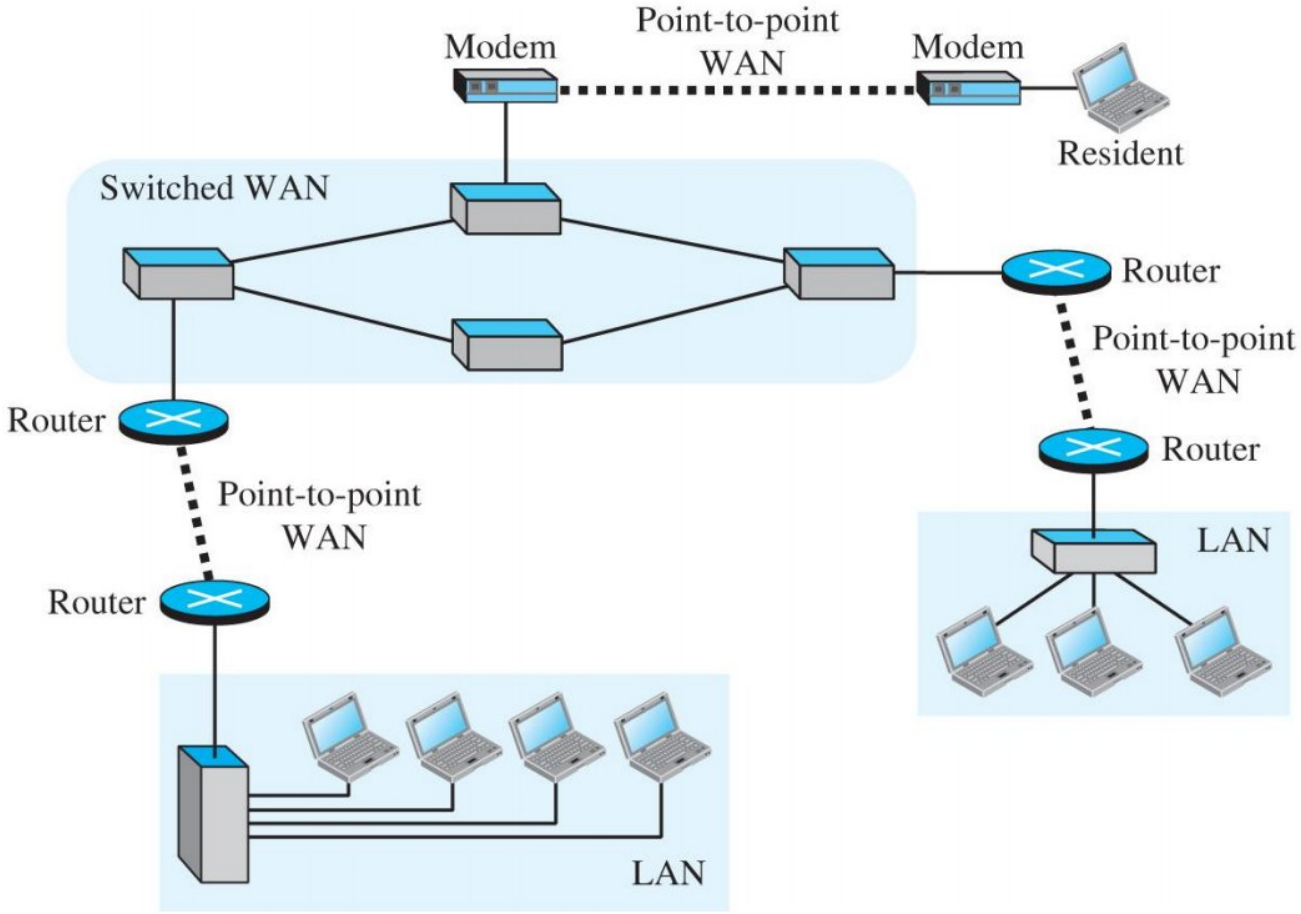
A switched WAN



An internetwork made of two LANs and one WAN



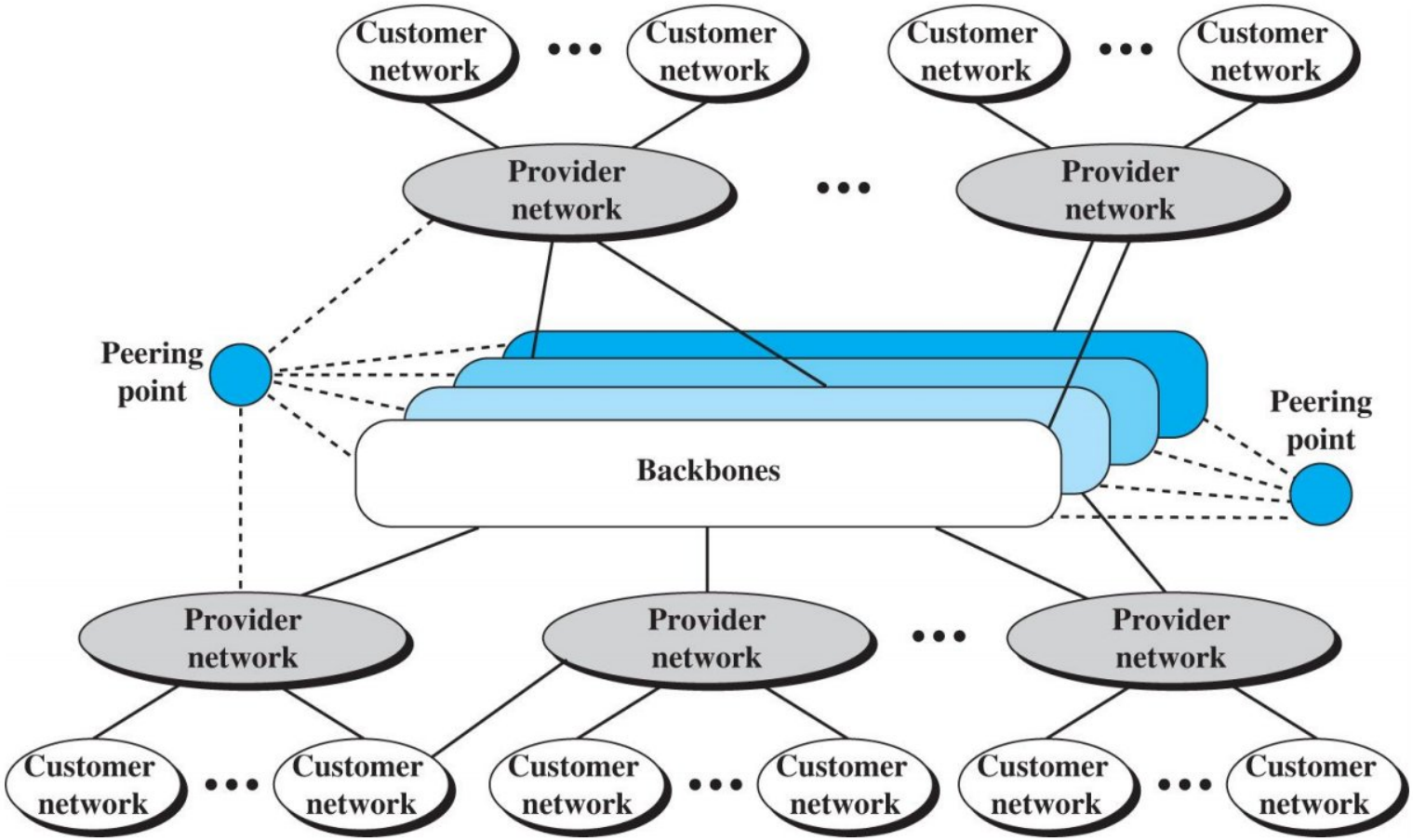
A heterogeneous network made of WANs and LANs



The Internet

- An internet (note the lowercase i) is two or more networks that can communicate with each other.
- The most notable internet is called the Internet (uppercase I) and is composed of millions of interconnected networks.
- Figure 1.13 shows a conceptual (not geographical) view of the Internet.

The Internet today



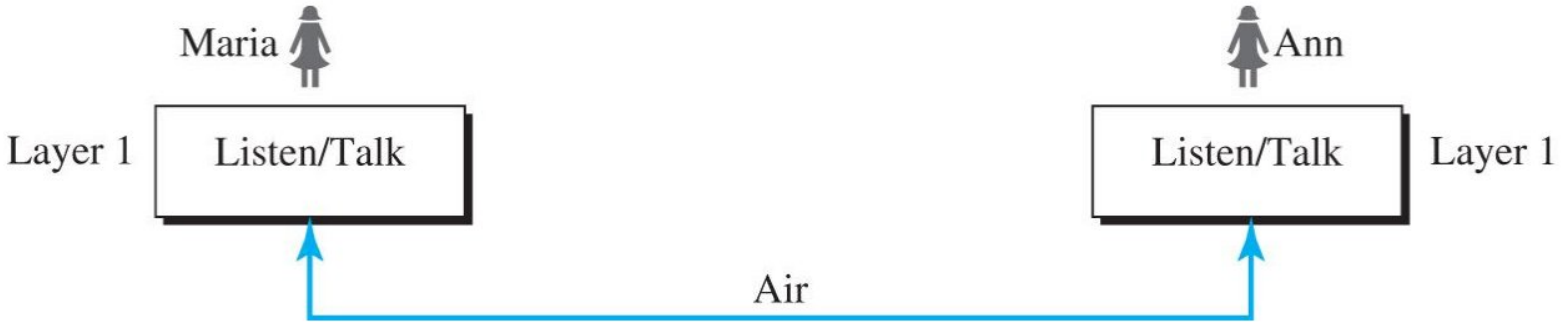
PROTOCOL LAYERING

- We defined the term **protocol** before. In data communication and networking, a protocol defines the rules that both the sender and **receiver** and all intermediate devices need to follow to be able to communicate directly.

First Scenario

- A large organization or a large corporation can itself become a local ISP and be connected to the Internet.
- This can be done if the organization or the corporation leases **a high-speed WAN** from **a carrier provider** and **connects itself to a regional ISP**.

A single-layer protocol

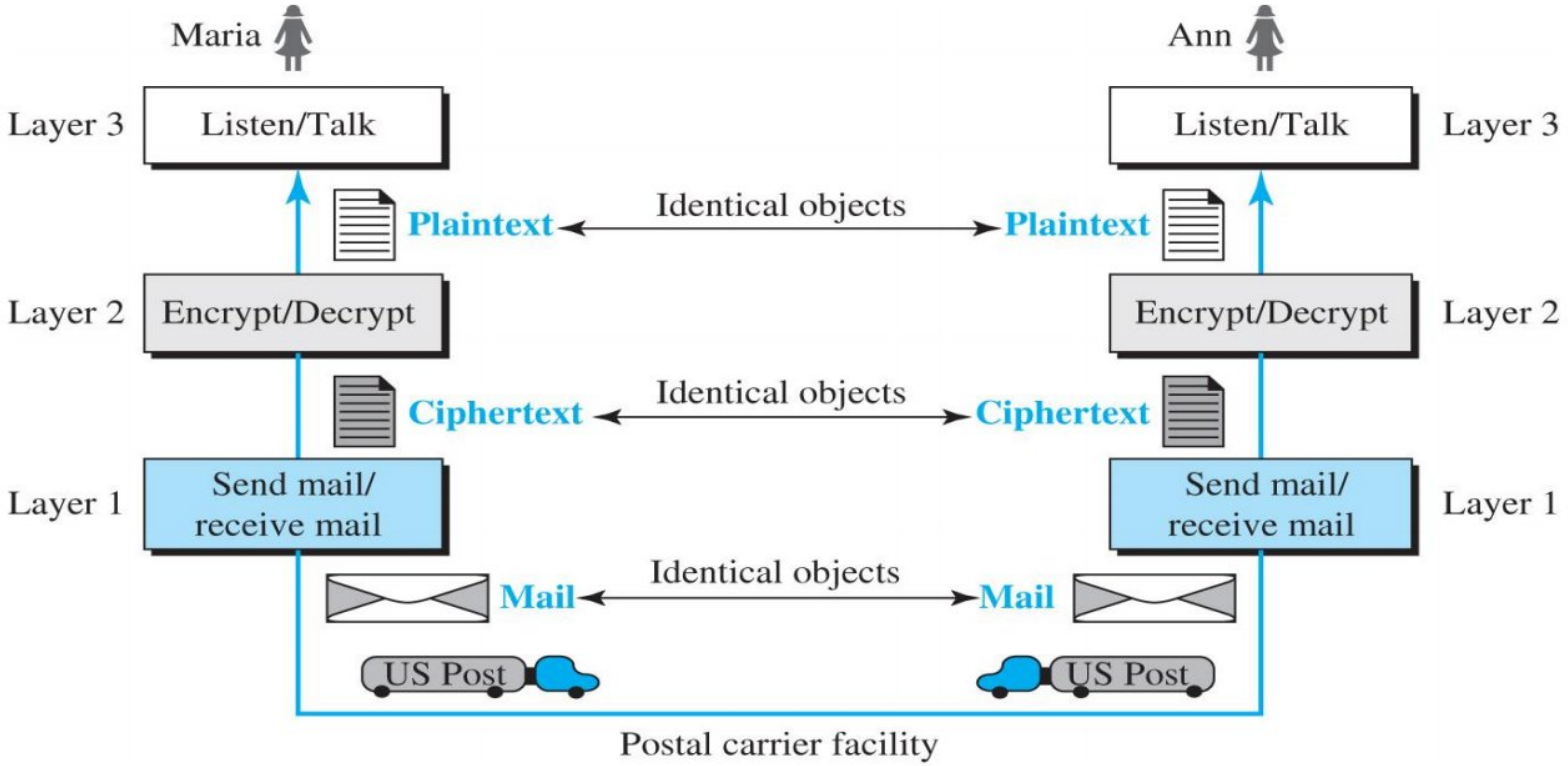


In single layer protocol, the two persons named Maria on one side and Ann on another side communicates (listen or talk) through air medium. The listen or talk actions on both sides are represented as layer 1.

Second Scenario

- In the second scenario, we assume that **Ann is offered a higher-level position** in her company, but needs to move to another branch located in a city very far from Maria.
- They decide to continue their conversion using regular mail through the post office. However, they **do not want their ideas to be revealed** by other people if the letters are intercepted. They use an **encryption/decryption** technique.

A three-layer protocol

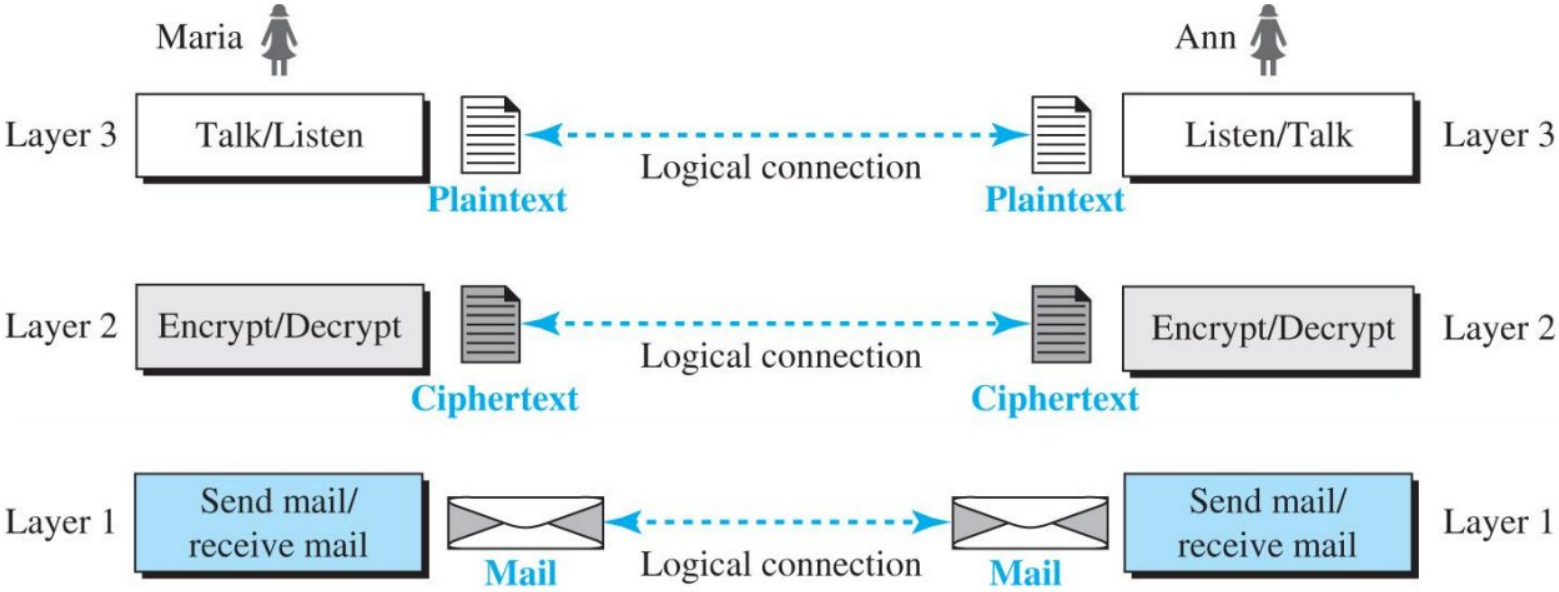


In three layer protocol, the communication between Maria and Ann is shown in three layers. The layer 1 is sending mail or receiving mail. The layer 2 is encrypting or decrypting the mail to a Ciphertext. The layer 3 is listening or talking where the ciphertext is converted to a plaintext. Each peer layers share identical objects that is the mail, ciphertext, and plaintext in both sides (Maria and Ann) are identical objects.

Principles of Protocol Layering

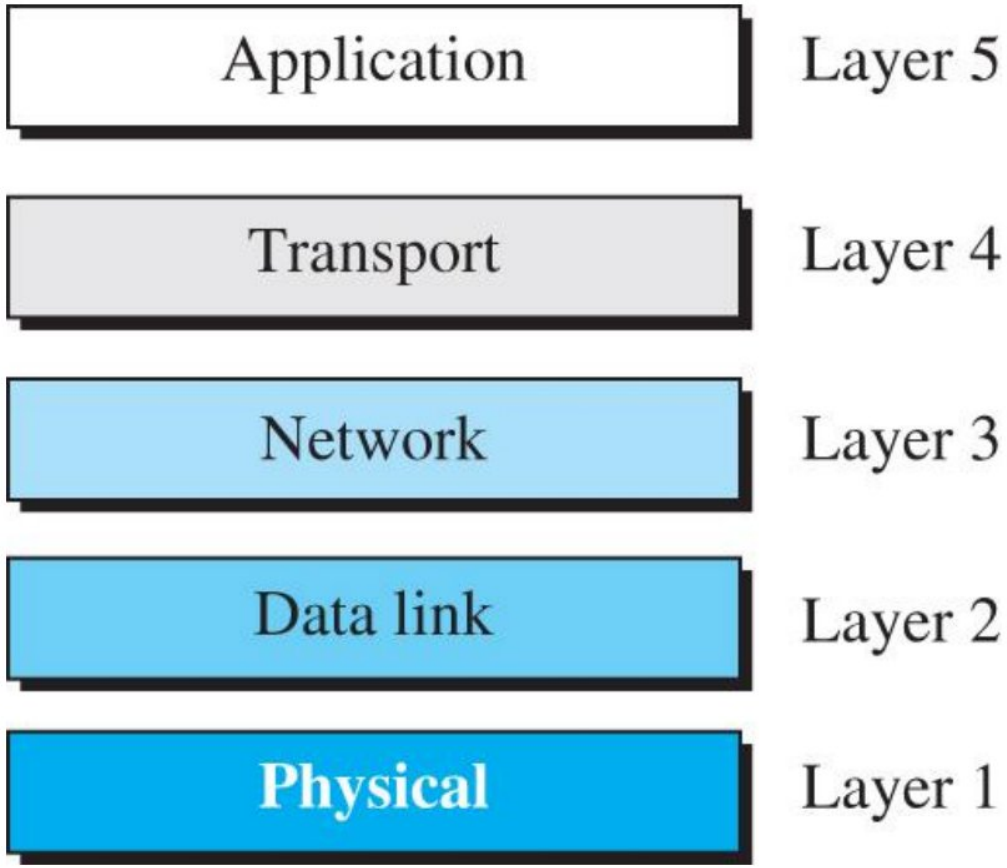
- #1. The first principle dictates that we need to make each layer to perform **two opposite task** in each direction.
- #2. The **second principle** dictates that **two objects** under each layer **should be identical.**

Logical connection between peer layers



A three layer protocol communication between Maria and Ann is shown is demonstrated. The layer 1 is sending mail or receiving mail. The layer 2 is encrypting or decrypting the mail to a Ciphertext. The layer 3 is listening or talking where the ciphertext is converted to a plaintext. **Each peer layer establishes a logical connection** between the objects they share to the successive layers.

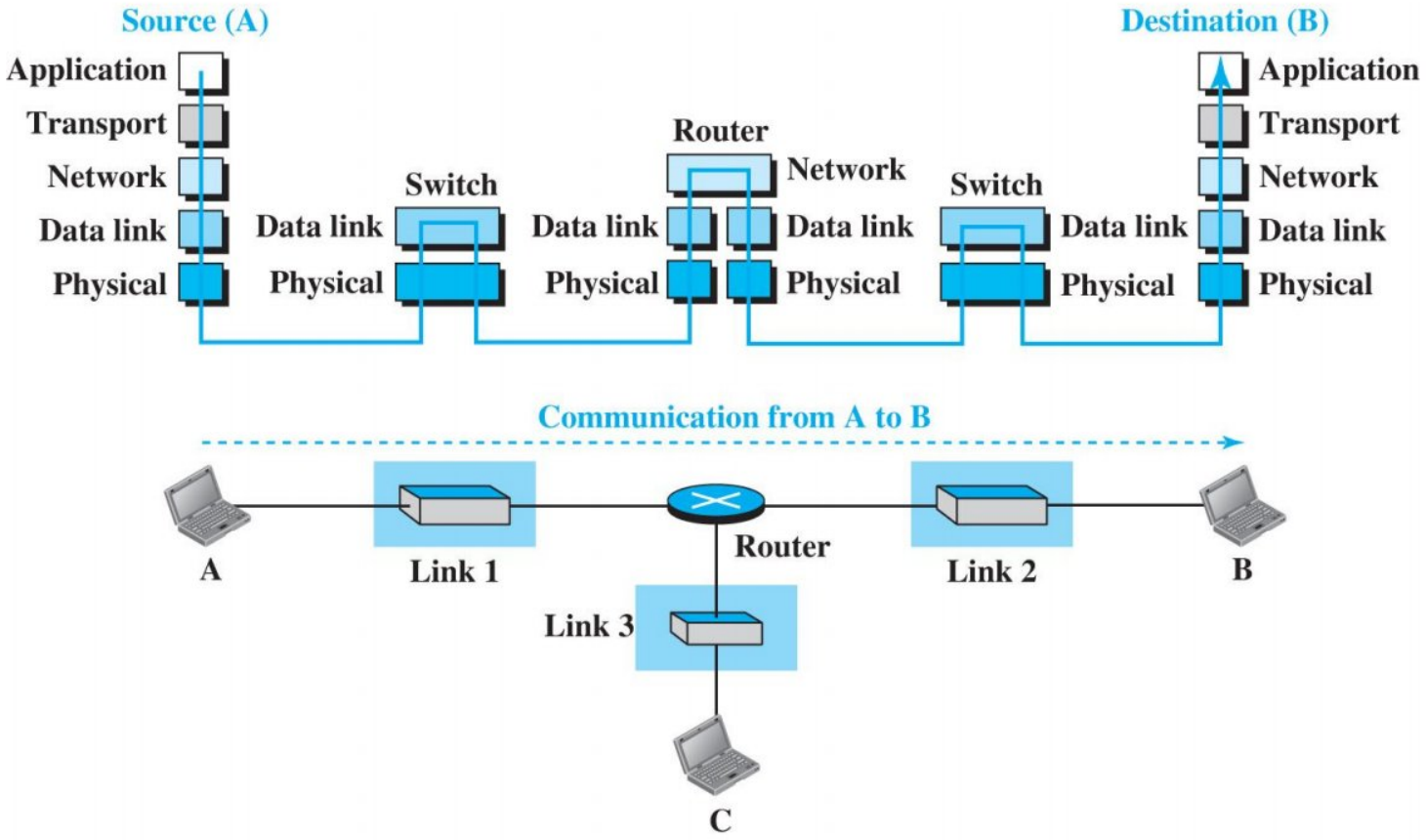
Layers in the TCP/IP protocol suite



Layered Architecture

- To show how the layers in the TCP/IP protocol suite are involved in communication between two hosts, we assume that we want to use the suite in a small internet made up of three LANs (links), each with a link-layer switch. We also assume that the **links are connected by one router.**

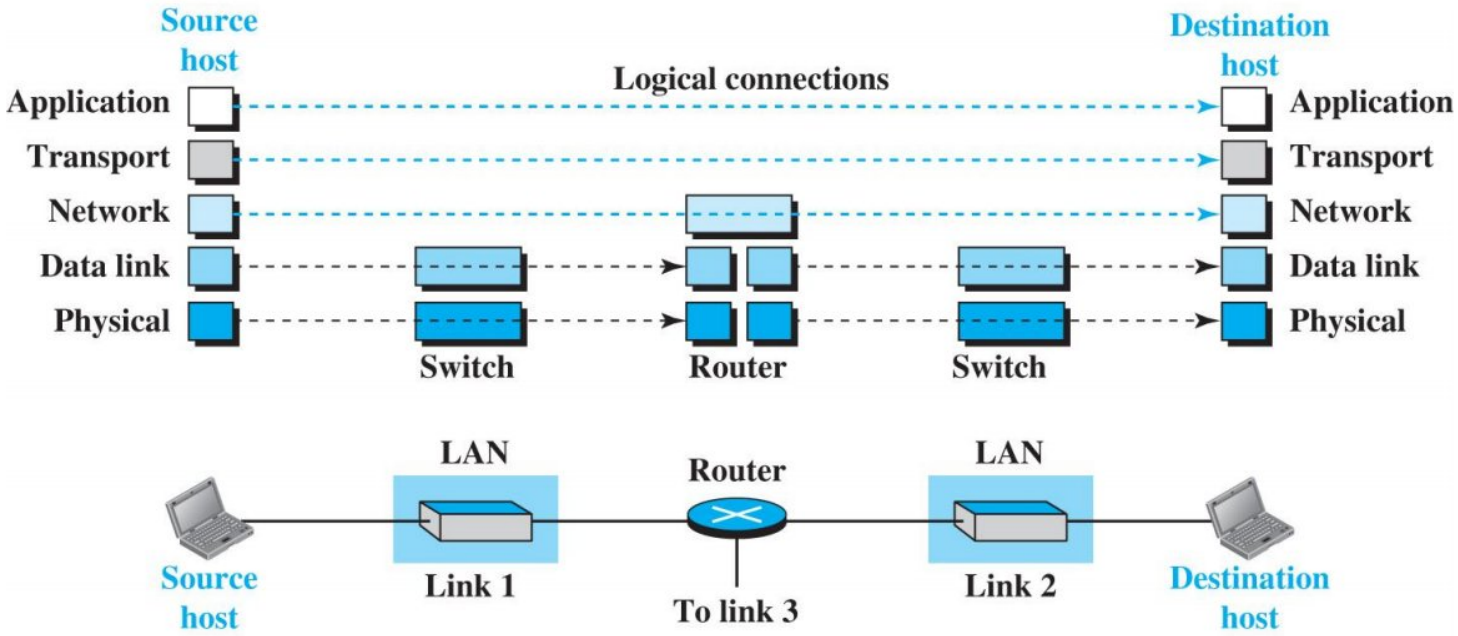
Communication through an internet



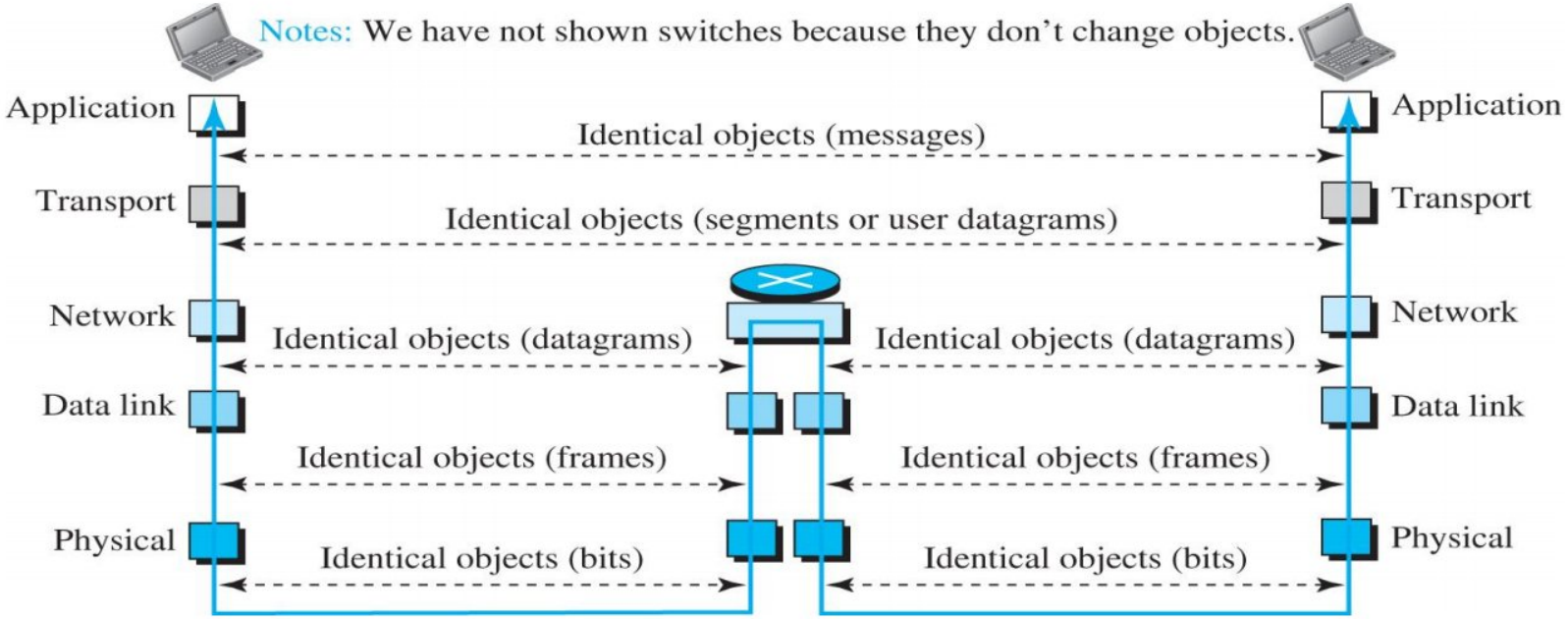
Brief Description of Layers

- After the above introduction, we briefly discuss the functions and duties of layers in the TCP/IP protocol suite. To better understand the duties of each layer, we need to think about the logical connections between layers.
- .

Logical connections between layers in TCP/IP



Identical objects in the TCP/IP protocol suite



An illustration shows that the first laptop and the second laptop have physical data link, networks, transport, and application. Reversible communication between the first and second laptops is shown. The application of first and second laptops shares identical objects (**messages**). The transport of first and second laptops shares identical objects (**segments or user datagrams**). The network of first and second laptops shares identical objects (**datagram**s) through the router. The data link of the first and second laptops shares identical objects (**frames**) through the router. The physical of first and second laptops share identical objects (**bits)** through the router.

Characteristics of Different Layers

- **#1.** We can say that the **physical layer** is responsible for carrying individual bits in a frame across the link.
- The physical layer is the lowest level in the TCP/IP protocol suite, the communication between two devices at the physical layer is still a logical communication because there is another, hidden layer, the transmission media, under the physical layer.
- **#2**. We have seen that an internet is made up of several links (LANs and WANs) connected by routers. When the next link to travel is determined by the router, the **data-link layer** is responsible for taking the datagram and moving it across the link.
- **#3.** The **network layer** is responsible for creating a connection between the source computer and the destination computer.
- The communication at the network layer is host-to-host.
- However, since there can be several routers from the source to the destination, the routers in the path are responsible for choosing the best route for each packet.

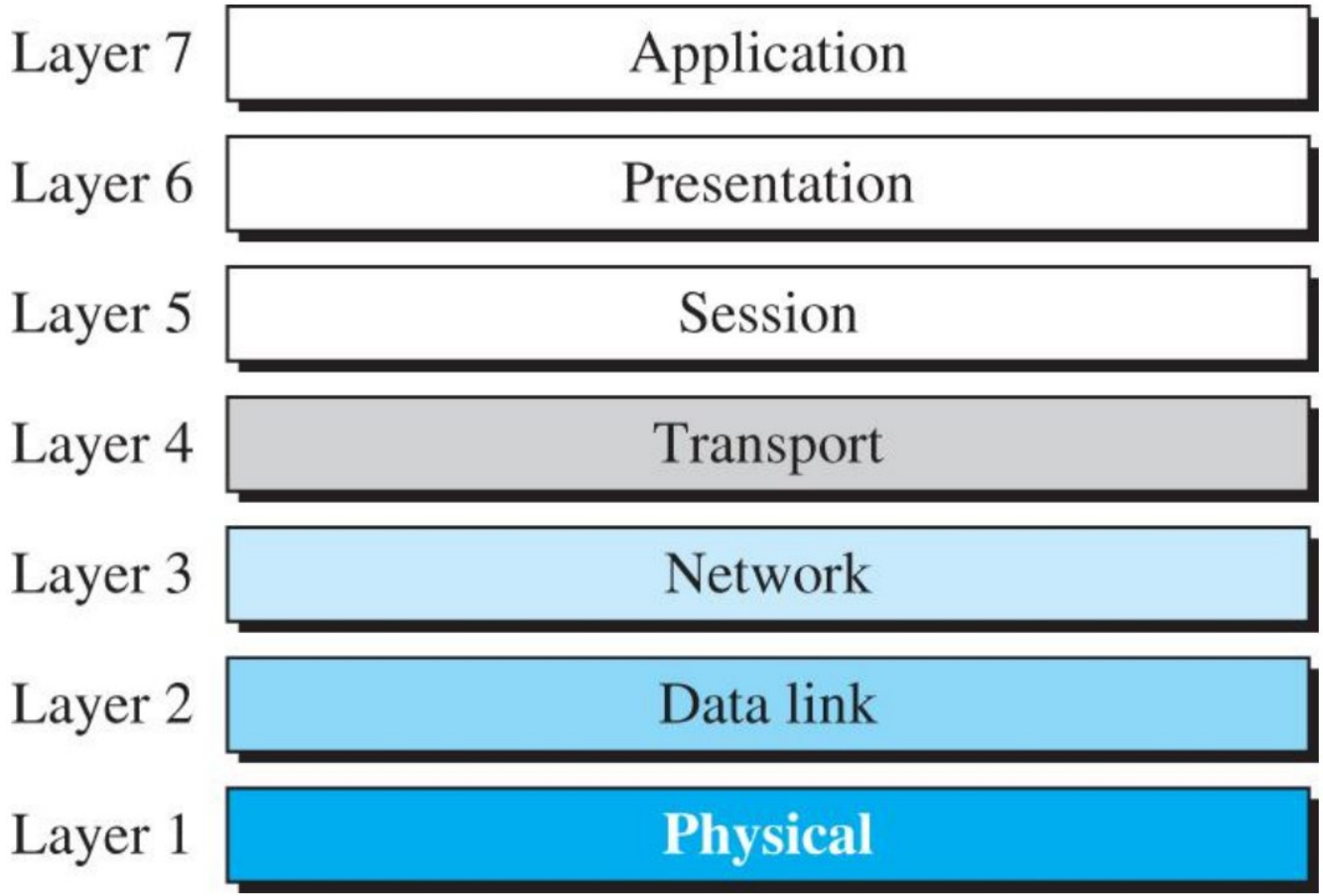
Characteristics of Different Layers Cont.

- **#4**. The logical connection at the **transport layer** is also end-to-end. The transport layer at the source host gets the message from the application layer, **encapsulates** it in a transport-layer packet.
- In other words, the transport layer is responsible for giving services to the application layer: to get a message from an application program running on the source host and deliver it to the corresponding application program on the destination host. transmits user datagrams without first creating a logical connection.
- **#5.** The logical connection between the two **application layers** is end-to-end.
- The two application layers exchange messages between each other as though there were a bridge between the two layers. However, we should know that the communication is done through all the layers. Communication at the application layer is between two processes (two programs running at this layer).
- To communicate, a process sends a request to the other process and receives a response. Process-to-process communication is the duty of the application layer.

Open System interconnection (OSI) MODEL

- Although, when speaking of the Internet, everyone talks about the TCP/IP protocol suite, this suite is not the only suite of protocols defined.
- Established in 1947, the International Organization for Standardization (ISO) is a multinational body dedicated to worldwide agreement on international standards.
- Almost three-fourths of the countries in the world are represented in the ISO. An ISO standard that covers all aspects of network communications is the Open Systems Interconnection (OSI) model. It was first introduced in the late 1970s.

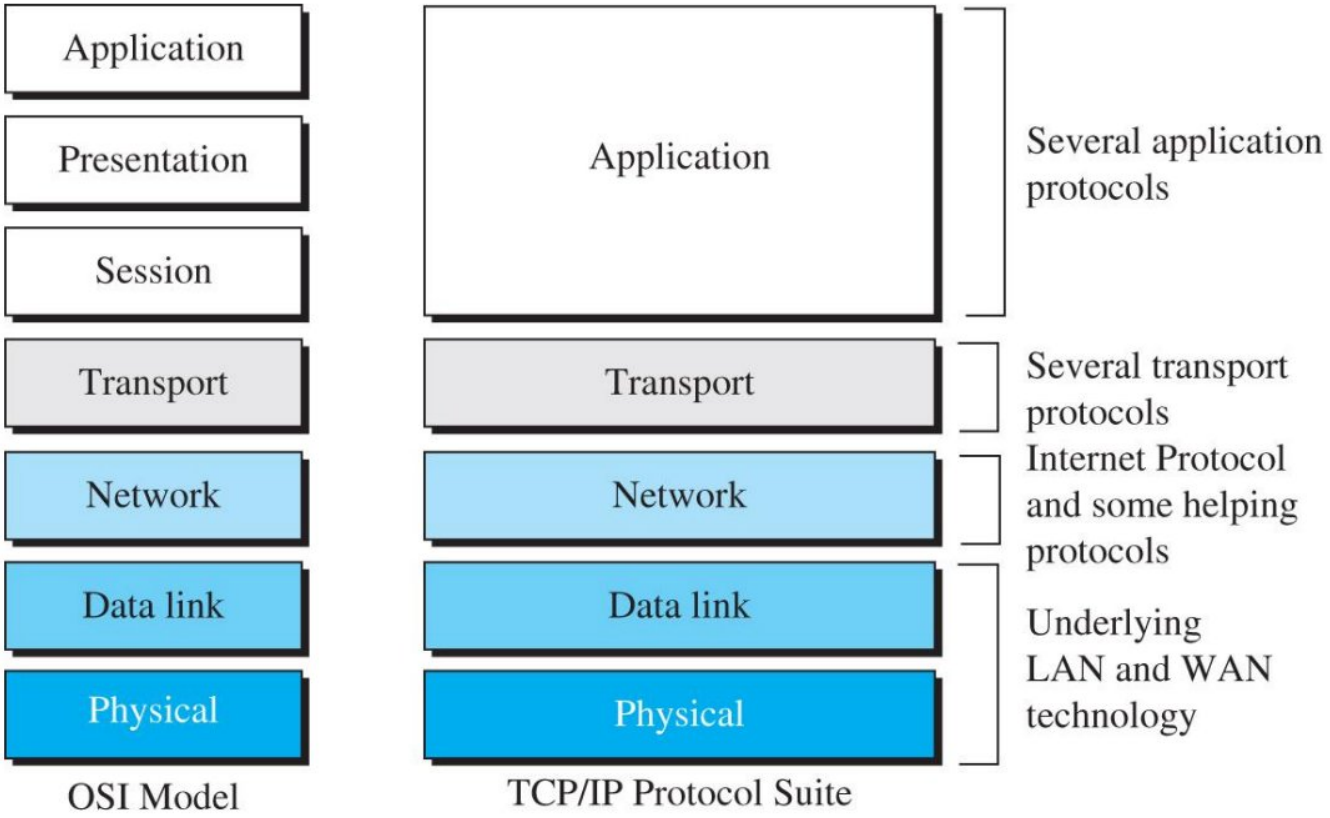
The OSI model



OSI versus TCP/IP

- When we compare the two models, we find that two layers, session and presentation, are missing from the TCP/IP protocol suite. These two layers were not added to the TCP/IP protocol suite after the publication of the OSI model.
- The application layer in the suite is usually considered to be the combination of three layers in the OSI model.

TCP/IP and OSI model



UNIT-2: Communications in Distributed Systems

- Basics of Communication Systems
- Layered Protocols
- ATM Models
- Client-Server Model
- Blocking and Non-Blocking Primitives
- Buffered and Un-Buffered Primitives
- Reliable and Unreliable Primitives
- Message passing
- Remote Procedure Calls

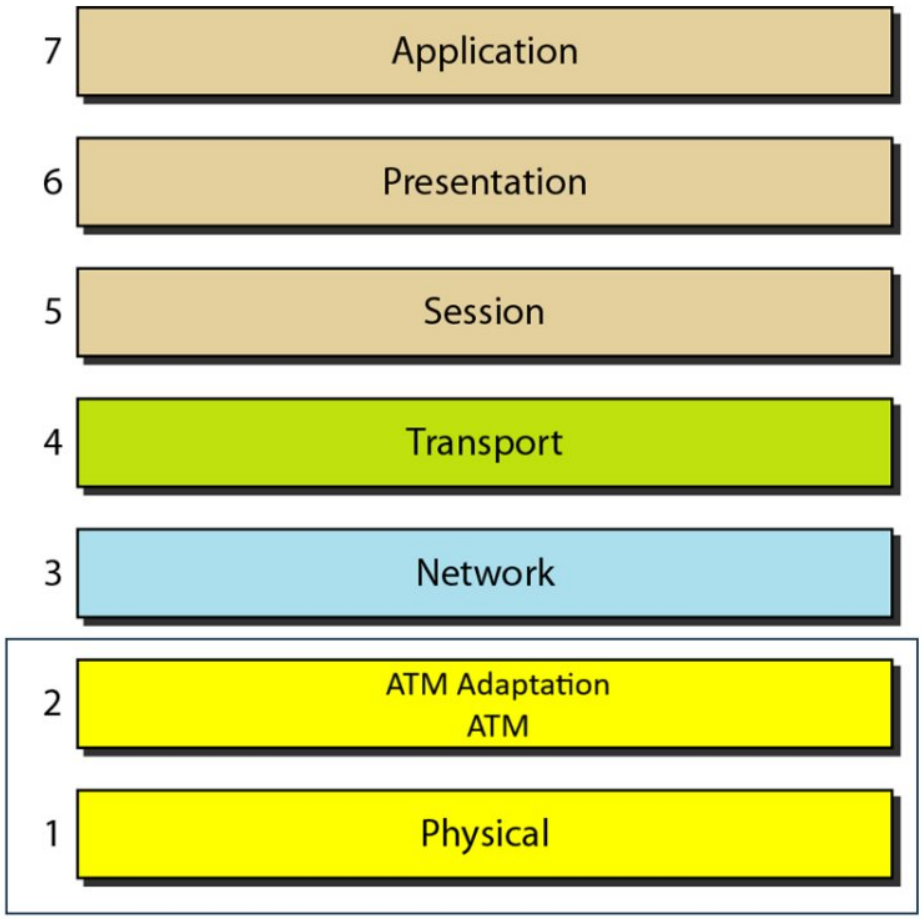
#3: Asynchronous Transfer Mode (ATM)

- ATM is the **cell relay** protocol designed by ATM forum and adopted by ITU-T
- It is a cell switching and multiplexing technology that combines benefits of both **circuit switching** and **packet switching**
- ATM working principles
  - **Sender first establishes a connection (virtual circuit) to the receiver(s)**
  - A route is determined from sender to receiver
  - Routing information is stored in the switches along the way
  - Packets can be sent through this connection by sender
  - **Packets are chopped into small fixed-sized units (cell) by hardware**
  - **Routing information purged from switches when connection is not required**

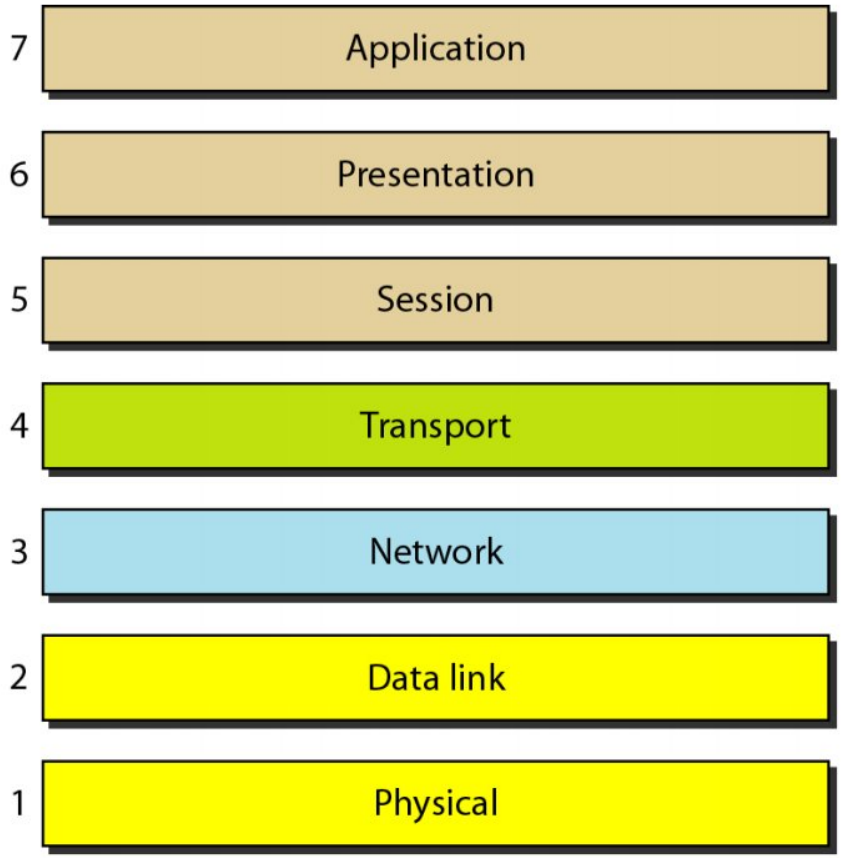
ATM Advantages

- A single network is used to transport voice, data, broadcast television, videotapes, radio.
- It is used for video conferencing, video-on-demand, teleconferencing, access to thousands of remote databases
- Cost saving
- ATM uses cell switching which handles both point-to-point and multicasting efficiently
- ATM allows **rapid switching** as its cell (or packet) **size is fixed**

#3.1 ATM Layers



ATM Reference Model



ISO/OSI Model

3.2: ATM Physical Layer

- In the Physical Layer, **ATM is synchronous as it transmits empty cells** while no data to be send.
- It uses **SONET** (Synchronous Optical NETwork) in physical Layer.
- In SONET, frame size is 810 bytes (overhead: 36 bytes, payload: 744 bytes), gross data rate 51.840 Mbps.
- Basic 51.840 Mbps channel is called **Optical Carrier** (OC-1)
- OC-12 and OC-48 are used for **long-haul** transmission.

3.3: ATM Layer

- Generic Flow Control (GFC) is used for flow control.
- Virtual Path Identifier (VPI) and Virtual Channel Identifier (VCI) together identify path and circuit of a cell
- Payload type distinguishes data cells from control cells
- Cell Loss Priority (CLP) identifies the less important cells which drop if congestion occurs
- Cyclic Redundancy Check (CRC) identifies redundancy and correct it

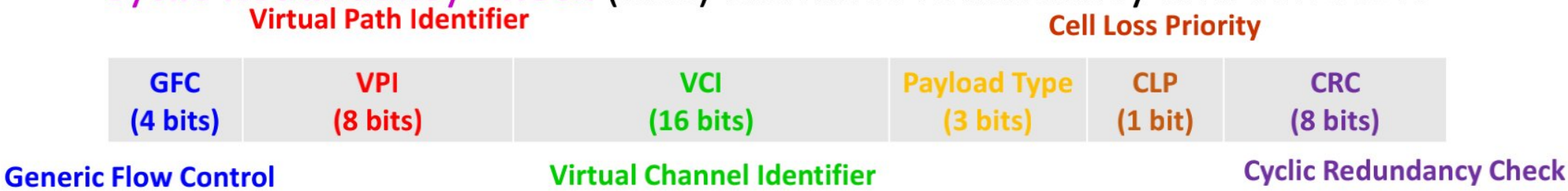


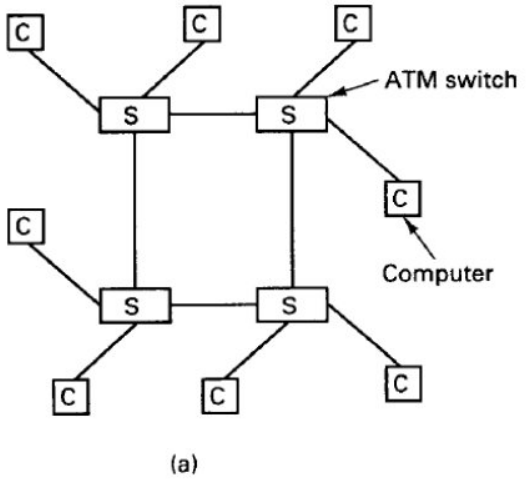
Figure: User-to-Network Cell Header Layout

3.4: ATM Adaptation Layer

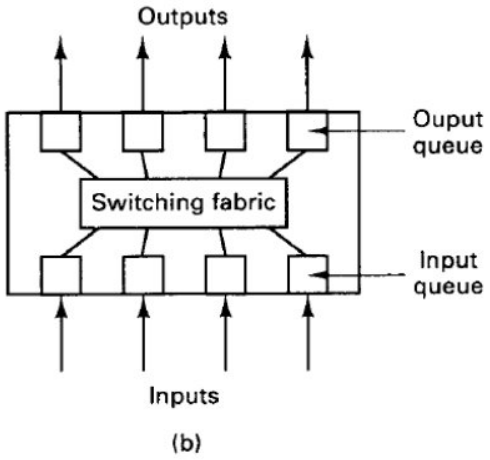
- Adaptation Layer has **four classes**
  - Constant bit rate traffic (for audio and video) : **CBR Traffic**
  - Variable bit rate traffic but with bounded delay: VBR Traffic
  - Connection-oriented data traffic
  - Connectionless data traffic
- Simple and Efficient Adaptation Layer (SEAL)
  - 1 bit of ATM *header*, 1 bit of *Payload Type*
  - Payload Type field is set to 1 for last cell, otherwise 0
  - Last cell contains 8 bytes tailer with four fields
  - Tailer contains packet length (2 bytes), checksum (4 bytes)
  - There are no use of first two fields (1 byte each field)

#3.5: ATM Switching

- Network built with 4 switches and 8 computers
- Cells can be switched different computers by traversing switches
- Switching fabric connects input and output lines and ensures **parallel switching**
- **Head-of-line blocking problem**
- **Solution: Keep copy of a cell in a output buffer queue**



(a) ATM Switching Network



(b) Inside of One Switch

#3.6: ATM Implications for Distributed Systems

- High-speed network but latency remains
- Flow control
- Transcontinental Delay
- Cell drops during congression

#3.7: ATM Advantages

- High-speed, fast-switched integrated data, voice, and video communication.
- A standards-based solution formalized by the International Telecommunication Union (ITU)
- Interoperability with standard LAN/WAN technologies
- **QoS technologies(The parameters are: End-to-end delay, Delay Jitter, PLR, Bandwidth usage etc.)** that enable a single network connection to reliably carry voice, data, and video simultaneously.

UNIT-2: Communications in Distributed Systems

- Basics of Communication Systems
- Layered Protocols
- ATM Models
- Client-Server Model
- Blocking and Non-Blocking Primitives
- Buffered and Un-Buffered Primitives
- Reliable and Unreliable Primitives
- Message passing
- Remote Procedure Calls

4.1. Interprocess Communication

- In a distributed system, it is completely different from a uniprocessor system as there is no shared memory.
- Certain rules need to be followed for interprocess communication, called Protocols.
- For wide-area distributed systems, these protocols take the form of **multiple layers,** such as OSI and ATM.
- The OSI model addresses only a small aspect of the communication - sending bits from the sender to the receiver, with much overhead.

4.2 Client – Server Model

- It is based on simple **connectionless** request / reply protocol.
- Client sends a request message to the server and the server **returns the data requested** **or** an error code indicating the reason of failure.
- It is simple. No connection to be established before use and no connection to be closed after use.
- Simplicity leads to efficiency. Only **three levels** of protocol are needed.

4.2 Client – Server Model Cont.

- Physical and datalink protocol take care of getting the packets from client to server and back.
- No routing and no connections - layers 3 & 4 not needed.
- Layer 5 is the request/ reply protocol. No sessions required.
- Communication provided by the micro-kernels using two system calls -
  - *send (dest, &mptr)*
  - *receive(addr, &mptr)*

mptr - message pointer

dest - destination process

addr - source address

Example : Client and Server Program

```
/* Definitions needed by clients and servers.                    */
#define TRUE              1
#define MAX_PATH          255   /* maximum length of file name          */
#define BUF_SIZE          1024  /* how much data to transfer at once    */
#define FILE_SERVER       243   /* file server's network address        */

/* Definitions of the allowed operations */
#define CREATE            1     /* create a new file                    */
#define READ              2     /* read data from a file and return it  */
#define WRITE             3     /* write data to a file                 */
#define DELETE            4     /* delete an existing file              */

/* Error codes. */
#define OK                0     /* operation performed correctly        */
#define E_BAD_OPCODE      -1    /* unknown operation requested          */
#define E_BAD_PARAM       -2    /* error in a parameter                 */
#define E_IO              -3    /* disk error or other I/O error        */

/* Definition of the message format. */
struct message {
      long source;              /* sender's identity                    */
      long dest;                /* receiver's identity                  */
      long opcode;              /* requested operation                  */
      long count;               /* number of bytes to transfer          */
      long offset;              /* position in file to start I/O        */
      long result;              /* result of the operation              */
      char name[MAX_PATH];      /* name of file being operated on       */
      char data[BUF_SIZE];      /* data to be read or written           */
};
```

Example : Client and Server Program

```
#include <header.h>
void main(void) {
     struct message ml, m2;                  /* incoming and outgoing messages          */
     int r;                                  /* result code                             */

     while(TRUE) {                           /* server runs forever                     */
          receive(FILE_SERVER, &ml);         /* block waiting for a message             */
          switch(ml.opcode) {                /* dispatch on type of request             */
               case CREATE:    r = do_create(&ml, &m2); break;
               case READ:      r = do_read(&ml, &m2); break;
               case WRITE:     r = do_write(&ml, &m2); break;
               case DELETE:    r = do_delete(&ml, &m2); break;
               default:        r = E_BAD_OPCODE;
          }
          m2.result = r;                     /* return result to client                 */
          send(ml.source, &m2);              /* send reply                              */
     }
}
```

Example : Client and Server Program

(a)

```
#include <header.h>
int copy(char *src, char *dst){                  /* procedure to copy file using the server   */
     struct message ml;                          /* message buffer                            */
     long position;                              /* current file position                     */
     long client = 110;                          /* client's address                          */

     initialize( );                              /* prepare for execution                     */
     position = 0;
     do {
          ml.opcode = READ;                      /* operation is a read                       */
          ml.offset = position;                  /* current position in the file              */
          ml.count  = BUF_SIZE;                                                                /* how many bytes to read*/
          strcpy(&ml.name, src);                 /* copy name of file to be read to message   */
          send(FILESERVER, &ml);                 /* send the message to the file server       */
          receive(client, &ml);                  /* block waiting for the reply               */

          /* Write the data just received to the destination file.                            */
          ml.opcode = WRITE;                     /* operation is a write                      */
          ml.offset = position;                  /* current position in the file              */
          ml.count  = ml.result;                 /* how many bytes to write                   */
          strcpy(&ml.name, dst);                 /* copy name of file to be written to buf    */
          send(FILE_SERVER, &ml);                /* send the message to the file server       */
          receive(client, &ml);                  /* block waiting for the reply               */
          position += ml.result;                 /* ml.result is number of bytes written      */
     } while( ml.result > 0 );                   /* iterate until done                        */
     return(ml.result >= 0 ? OK : ml result);    /* return OK or error code                   */
}
```

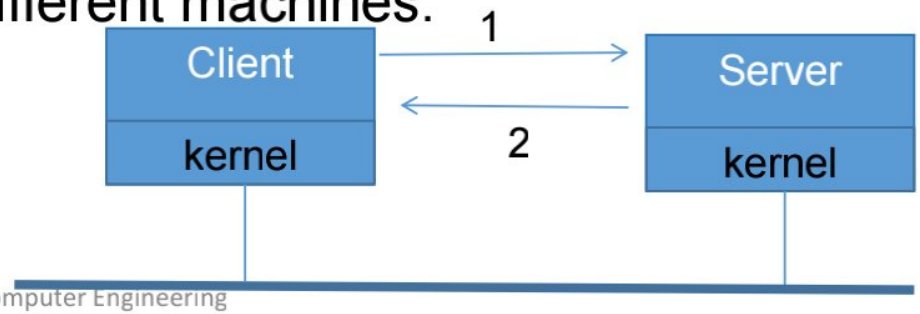
4.3 Addressing

- One way of mentioning server address is to mention it in ***header.h*** as a constant.
- Sending kernel can extract it (ex - 243) from message structure and use it for sending packets to server.
- Ambiguity arises if multiple processes are running on the same server.

Alternative 1 -

- Send messages to processes , not machines.
- Process identification - two part names - machine + process no.
  Ex - 243.4 or 4@243
- Each machine can number its processes starting from 0. So there is no confusion between process 'n' of different machines.
- No global coordination is required.

1. **Request to 243.0**
2. **Reply to 199.0**



4.3 Addressing Cont.

Alternative 2 -

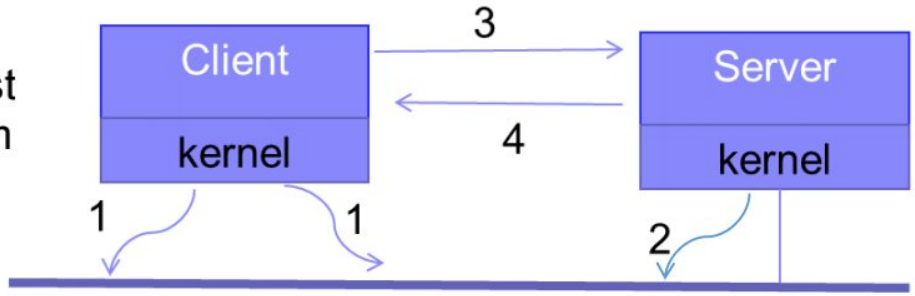
- use ***machine.local-id*** instead of ***machine.process***
- Each process is assigned a *local-id* and informs kernel that it listens to *local-id*
- Problem - user is aware of the location of the machine (243). If the machine is down, compiled programs with ***header.h*** **will not work**, although another machine (365) is available. **No transparency**.

Alternative 3-

- Each process has a unique address that doesn't contain machine number.
- A centralized **process address allocator** maintains a counter. Upon receiving a request, it returns the current value of the counter.
- Problem - Such centralized components **do not scale** to large systems.

4.3 Addressing Cont.

1. Broadcast
2. Here I am
3. Request
4. Reply

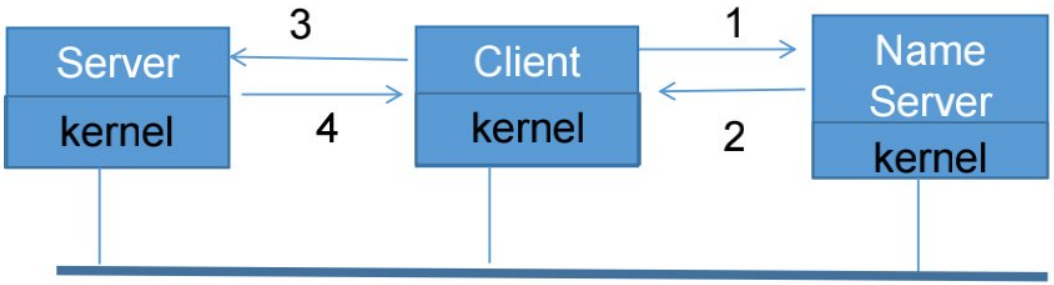


Alternative 4 -

- Each process picks its identifier from a large address space (a space of 64-bit binary integer).
- **It is scalable**.
- Identification of machine -Sender broadcasts a special **Locate packet** containing the address of the destination process. All machines on the network will receive it. The matched kernel responds with a message “Here I am” along with the machine number. So the sending kernel uses this machine number for further communication.
- Problem - **Broadcasting** is **an overload** to the system.

4.3 Addressing

1.Lookup
2. NS reply
3. Request
4. Reply



Alternative 5 -

- **Overload can be avoided** by providing **an extra machine** to map high-level (ASCII) service names to machine address.
- These **names are embedded in the programs**, not binary machine numbers.
- For the first time client sends a query to the Name server, asking the machine number where the server is currently located. Then the request can be sent directly to the machine address.
- Problem - If the name server is replicated, consistency problem may arise.

UNIT-2: Communications in Distributed Systems

- Basics of Communication Systems
- Layered Protocols
- ATM Models
- Client-Server Model
- Blocking and Non-Blocking Primitives
- Buffered and Un-Buffered Primitives
- Reliable and Unreliable Primitives
- Message passing
- Remote Procedure Calls

Blocking versus Nonblocking Primitives

- **Message Passing**: A message-passing system gives a collection of message-based IPC (Inter-Process Communication) protocols.
- The **send()** and **receive()** communication primitives are used by processes for interacting with each other.
- **For example**, Process A wants to communicate with Process B then Process A will send a message with send() primitive and Process B will receive the message with receive() primitive.
- **Synchronization Semantics**: The following are the two ways of message passing between processes:
- **Blocking (Synchronous)**
- **Non-blocking (Asynchronous)**

Cont..

- In case of blocking primitive (also called as **synchronous primitives**), when a process calls ***send*** it specifies a destination and a buffer to send to that destination.
- While the message is being sent, the sending process is **blocked** (i.e., suspended).
- The instruction following the call to ***send* is not executed until the message has been completly sent**, as shown in figure below.
- **Similarly, a call to *receive* does not return control until the message has actually been received and put in the message buffer**.

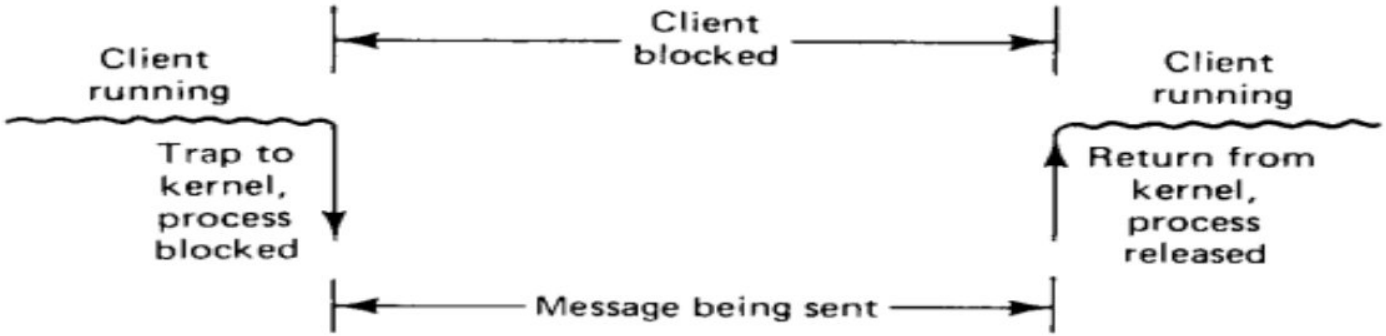


Figure 1: A blocking *send* primitive

Cont..

- An alternative to blocking primitives are **nonblocking primitives** (also called as **asynchronous primitives**).
- If ***send*** is nonblocking, it returns control to the caller immediately, before the message is sent.
- The **advantage** of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle.
- The disadvantage of this scheme is that the sender cannot modify the message buffer until the message has been sent.
- The sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer.
- **There are two possible solutions to this problem:**
- The **first solution** is to have the **kernel copy the message to an internal kernel buffer** and then allow the process to continue, as shown in Figure 2 in the next slide.
- The **disadvantage** of this method is that every outgoing message has to be copied from user space to kernel space.

Cont..

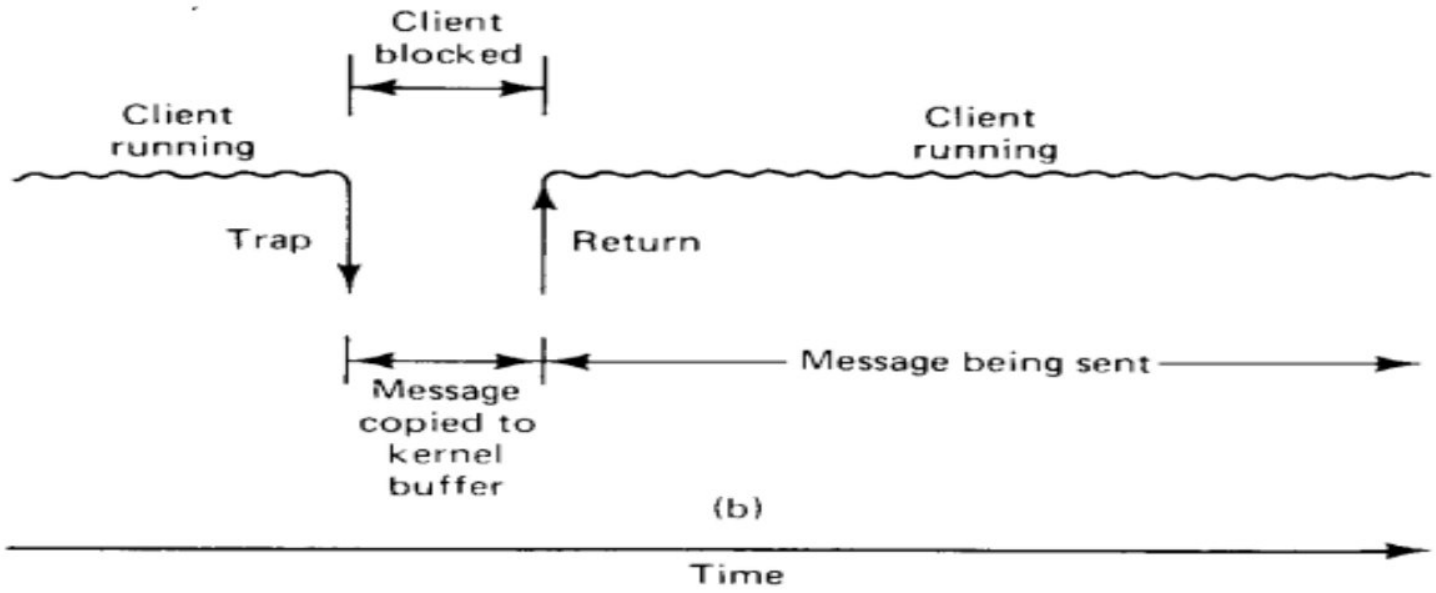


Figure 2: A nonblocking *send* primitive

Cont..

- The **second solution** is to **interrupt the sender** when the message has been sent to inform it that the buffer is once again available.
- No copy is required here, which saves time, but programs based on user-level interrupts are difficult to write and debug.

- **Blocking and nonblocking *send* primitives:**
- **Blocking send() primitive:** The blocking send() primitive refers to the blocking of sending process.
- The process remains blocking until it receives an acknowledgment from the receiver side that the message has been received after the execution of this primitive.
- **Non-blocking send() primitive:** The non-blocking send() primitive refers to the non-blocking state of the sending process that implies after the execution of send() statement, the process is permitted to continue further with its execution immediately when the message has been transferred to a buffer.

Cont..

- Just as ***send*** can be blocking and nonblocking, so can ***receive***.
- **Blocking receive() primitive:** when the receive statement is executed, the receiving process is halted until a message is received.
- **Nonblocking receive() primitive:** The non-blocking receive() primitive implies that the receiving process is not blocked after executing the receive() statement, control is returned immediately after informing the kernel of the message buffer's location.

- **The issue in a nonblocking *receive()* primitive is when a message arrives in the message buffer, how does the receiving process know?**
- One of the following two procedures can be used for this purpose:
- **Polling**: In the polling method, the receiver can check the status of the buffer when a test primitive is passed in this method.
- The receiver regularly polls the kernel to check whether the message is already in the buffer.

Cont..

- **Interrupt**: A software interrupt is used in the software interrupt method to inform the receiving process regarding the status of the message i.e. when the message has been stored into the buffer and is ready for usage by the receiver.
- So, here in this method, **receiving process keeps on running without having to submit failed test requests**.

Buffered versus Unbuffered Primitives

Unbuffered Primitives:

- Unbuffered primitives involve direct communication without any intermediate storage.
- In these primitives, the sender and receiver need to be **synchronized** for the communication to take place.
- A call to the primitive ***receive(addr, &m)*** tells the kernel of the machine on which it is running that the calling process is listening to the address *addr* and is prepared to receive one message sent to that address.
- A single message buffer, pointed to by *m*, is provided to hold the incoming message.
- When the message comes in, the receiving **kernel copies it to the buffer** and **unblocks the receiving process**, as shown by Figure 3 in the next slide.

Cont..

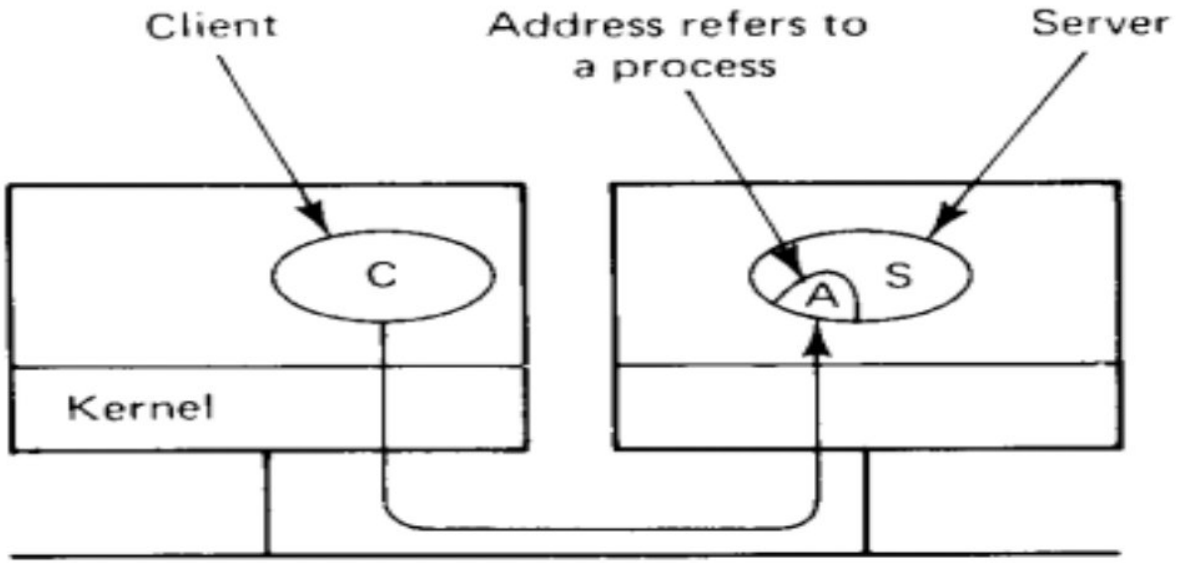


Figure 3: Unbuffered message passing

Cont..

- What are the problems that occurs when the client calls *send* primitive before the server calls *receive* primitive in unbuffered message passing mechanism?
- **The problems are as follows:**
- How does the server's kernel knows which of its process is using the address in the newly arrived message?
- How does the server's kernel knows where to copy the message?
- To avoid such problems, it's crucial to ensure that the *receive* primitive is called in a timely manner. Some strategies to handle this include:
- **Pre-emptive Design**: Design the system so that the *receive* is always invoked before or concurrently with ***send*** to avoid blocking.
- **Timeouts and Error Handling**: Implement timeouts or error handling mechanisms to manage situations where a ***send*** operation might block indefinitely.
- **Buffered Communication:** Use buffered message passing where messages are stored in a buffer temporarily, allowing the sender and receiver to operate asynchronously and reducing the risk of blocking.

Cont..

- **Buffered primitives:**
- In order to deal with the buffer management issues, a new data structure called a "**mailbox**" is defined.
- A process that is interested in receiving messages tells the kernel to create a mailbox for it, and specifies an address to look for the network packets.
- Henceforth, all incoming messages with that address are put in the mailbox.
- The call to receive removes on message from the mailbox, or blocks if none is present.
- In this way, the kernel knows what to do with incoming messages and has a place to put them.
- This technique is referred as **buffered primitive**, as shown by Figure 4, in the next slide.

Cont..

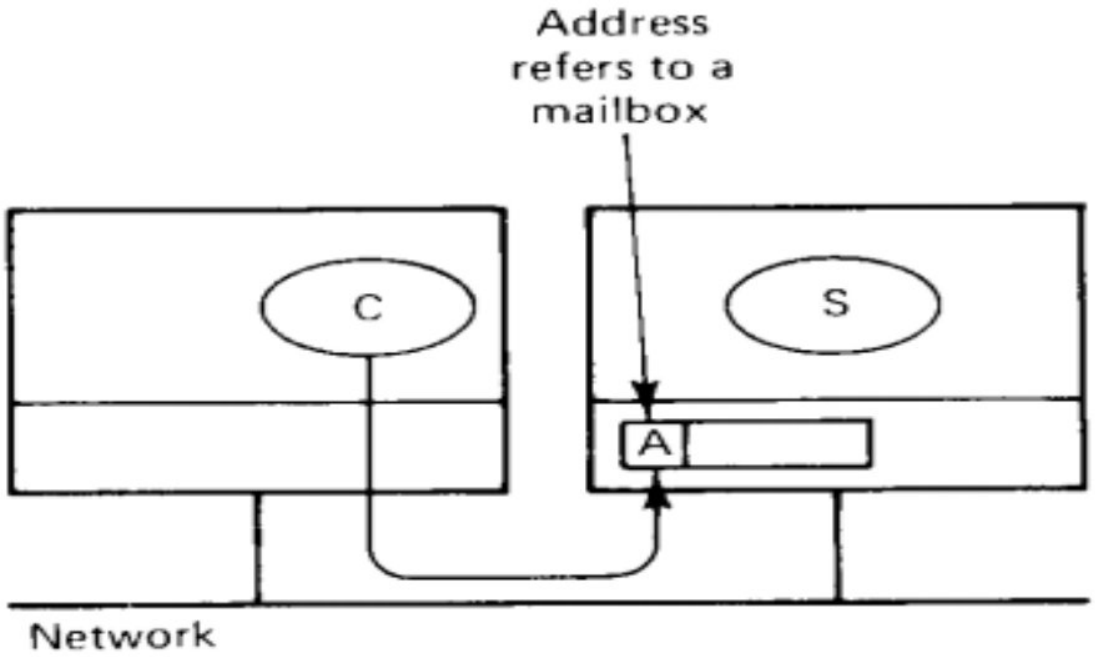


Figure 4: Buffered message passing

Reliable vs Unreliable Primitives

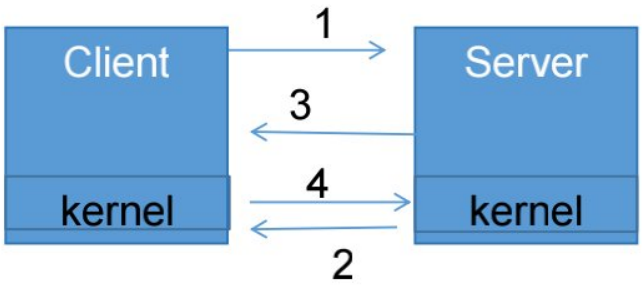
Using blocking primitives, the client process gets suspended after sending a message. When it is restarted, there is no guarantee that message has been delivered. The message might have been lost.

- **Alternative solution1** - Redefine the semantics of *send* to be unreliable. The system gives no guarantee about message delivery.
- **Alternative solution 2** - Kernel on the receiving machine sends an acknowledgement back to the kernel on sending machine.
- Sending kernel free the process after receiving this acknowledgement.
- Similarly, the reply from server back to

client is acknowledged by client's kernel.

- Acknowledgement goes from kernel to kernel.

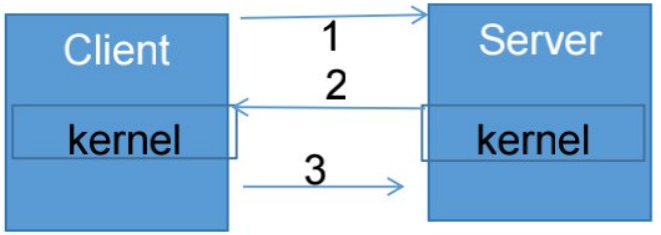
1. Request
2. ACK (kernel to kernel)
3. Reply
4. ACK (kernel to kernel)



Reliable vs Unreliable Cont.

- **Alternative solution3** - Client is blocked after sending message and the server's reply act as an acknowledgement.
- If the reply takes too long, the sender can resend the mesage to guard against lost message.
- An acknowledgement from client's kernel to the server's kernel is sometimes used. Until this packet is received, the server's *send* does not complete and the server remains blocked.
- If the reply is lost and the request is retransmitted, then the server kernel sends reply again without waking up the server.

1. Request (client to server)
2. Reply (server to client)
3. ACK (kernel to kernel)



Implementing client-server model

Design issues for the communication primitives

Item	Option 1	Option 2	Option 3
Addressing	Machne address	Sparse process addresses	Names looked up via server
Blocking	Blocking primitives	Nonblocking with copy to kernel	Nonblocking with interrupt
Bufferring	Unbuffered, discarding unexpected messages	Unbuffered, temporarly keeping unexpected messages	Mailboxes
Reliability	Unreliable	Request-Ack-Reply-Ack	Request-Reply-Ack

How message passing is implemented depends on which choices are made.

Implementing client-server model

Issue of packet size - All **packets have a limit of packet size**. Messages larger than this must be split up into multiple packets and sent separately.

- Problem - some of these packets may be **lost or distorted**. They may even arrive in the **wrong order**.
- Solution - Assign a number to each message and put it in each packet belonging to that message, along with a sequence number indicating the order of the packet.

Issue of acknowledgement -Acknowledge each individual packet or Acknowledge the entire message.

case I - if a packet is lost, only one packet need to be retransmitted. But it will cause more acknowledgements.

Case II - Fewer packets, but more complicated to recover if a packet is lost.

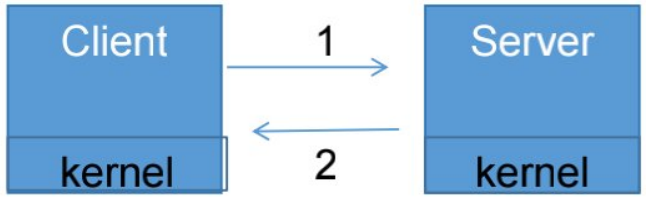
Implementing client-server model

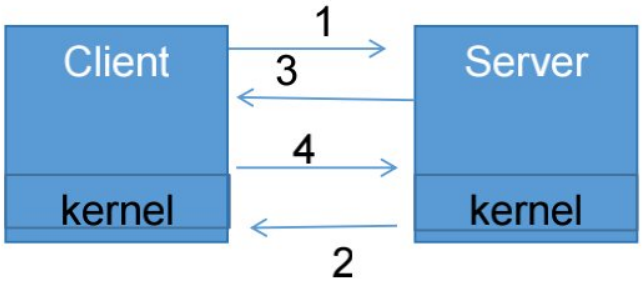
Issue of underlying protocol -

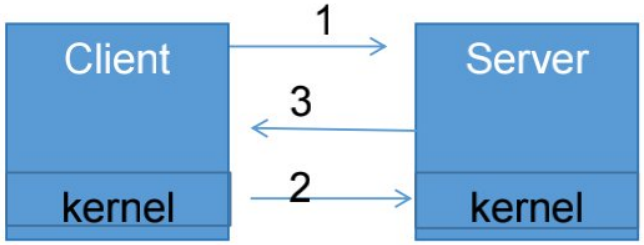
- AYA - to check whether request is complicated or the server is crashed
- TA - if request packet cannot be accepted

Code	Packet type	From	To	Description
REQ	Request	Client	Server	Client wants service
REP	Reply	Server	Client	Reply from server to the client
ACK	Acknowledgement	Either	Other	Previous packet arrived
AYA	Are you alive ?	Client	Server	Check if the server is crashed
IAA	I am alive	Server	Client	Server has not crashed.
TA	Try again	Server	Client	Server has no space
AU	Address unknown	Server	Client	No processis using this address

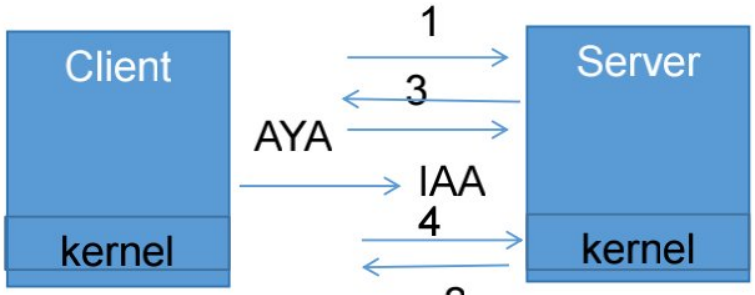
Implementing client-server model







1. Request
2. ACK (kernel to kernel)
3. Reply
4. ACK (kernel to kernel)



Examples of packet exchanges

UNIT-2: Communications in Distributed Systems

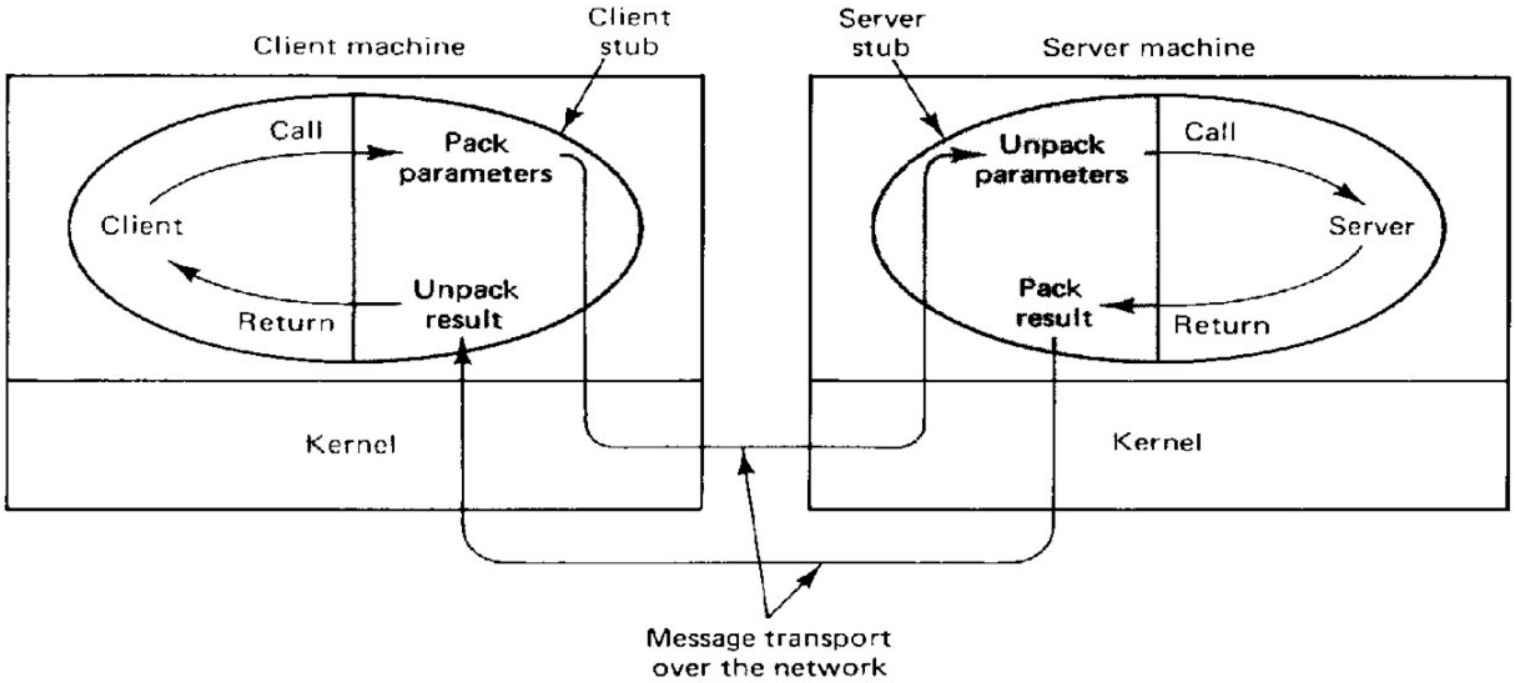
- Basics of Communication Systems
- Layered Protocols
- ATM Models
- Client-Server Model
- Blocking and Non-Blocking Primitives
- Buffered and Un-Buffered Primitives
- Reliable and Unreliable Primitives
- Message passing
- Remote Procedure Calls

Remote Procedure Call

Remote procedure call:- Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.

Calling and Called procedures run on different machines and they execute in different address spaces.

RPC is the widely used approach for Distributed Operating System.



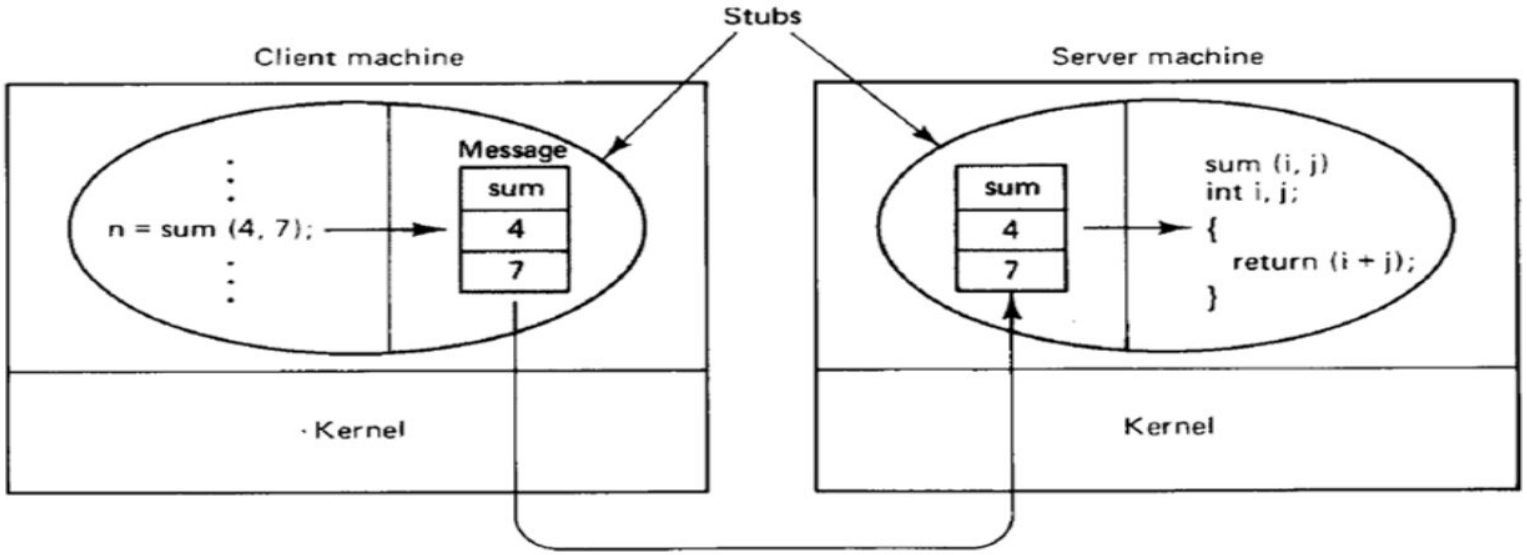
Steps for RPC Packing and Unpacking

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and traps to the kernel.
3. The kernel sends the message to the remote kernel.
4. The remote kernel gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and traps to the kernel.
8. The remote kernel sends the message to the client's kernel.
9. The client's kernel gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Parameter Passing

Parameter Marshaling:- Packing the parameters in the message.

The Client Stub takes the parameters and put them in a message. It also puts the number or name of the procedure to be called in the message. The server machine might support different calls.



- **Problem occurs when the system at client and server end is different.**
- Each machine has its own representation for numbers ,characters and other data items.
- IBM Mainframe machines :-EBCDIC character code.
- IBM personal Computer :- ASCII character code.

- Similar problem occurs with representation of integers and floating numbers.

- **Client End:-**The compiler reads the server specification and generate a client stub that packs its parameters into the officially approved message format.

- **Server End:-** The compiler can also produce a server stub that unpacks them and calls the server procedure.

- The **system is transparent** with respect to the differences in the internal representations of the data items.

Dynamic Binding.

- The client locates server in distributed syetem using Dynamic Binding.
- **Registering the Server to Binder:-**
- The server send a message to a program called a binder, to make its existance known.
- The server specifies its name, version number ,a unique identifier (32 bit long), and a handle used to locate it.
- The handle is system dependent (Ethernet Address, IP Address and X.500 Address, a sparse process identifier).
- It **can deregister** with the binder when it is **no** longer prepared to **offer service**.

How client locates server?

- The client stub send message to the binder asking it to import version of the server interface.
- The binder checks to see if one or more servers have already exported an interface with the version and name . If no server is found the read call fails.
- **Otherwise** , the binder gives its handle and unique identifier to the client stub. The client stub uses the handle as the address to send the request message

RPC Semantics in the Presence of Failure

Five different classes of failure that can occur in RPC systems:

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

Client cannot locate the server

- The server might be down, for example. Alternatively, suppose that the client is compiled using a particular version of the client stub, and the binary is not used for a considerable period of time. In the meantime, the server evolves and a new version of the interface is installed and new stubs are generated and put into use. When the client is finally run, the binder will be unable to match it up with a server and will report failure.
- The problem remains of how this failure should be dealt with.

- Wth the server of Fig. 2-9(a), each of the procedures returns a value, with the code –1 conventionally used to indicate failure. For such procedures, just returning –1 will clearly tell the caller that something is amiss.
- In UNIX, a global variable, errno, is also assigned a value indicating the error type. In such a system, adding a new error type "Cannot locate server" is simple.

- The trouble is, this solution is not general enough. Consider the sum procedure of Fig. 2-19. Here –1 is a perfectly legal value to be returned, for example, the result of adding 7 to –8. Another error-reporting mechanism is needed.

- One possible candidate is to have the error raise an exception. In some languages (e.g., Ada), programmers can write special procedures that are invoked upon specific errors, such as division by zero.

- This approach, too, has drawbacks. To start with, not every language has exceptions or signals.

Lost Reply Messages

- Lost replies are considerably more difficult to deal with. The obvious solution is just to rely on the timer again. If no reply is forthcoming within a reasonable period, just send the request once more.
- The trouble with this solution is that the client's kernel is not really sure why there was no answer. Did the request or reply get lost, or is the server merely slow? It may make a difference.
- In particular, some operations can safely be repeated as often as necessary with no damage being done. A request such as asking for the first 1024 bytes of a file has no side effects and can be executed as often as necessary without any harm being done. A request that has this property is said to be idempotent.
- Now consider a request to a banking server asking to transfer a million dollars from one account to another. If the request arrives and is carried out, but the reply is lost, the client will not know this and will retransmit the message. The bank server will interpret this request as a new one, and will carry it out too. Two million dollars will be transferred. Heaven forbid that the reply is lost 10 times. Transferring money is not idempotent.
- One way of solving this problem is to try to structure all requests in an idem-potent way. In practice, however, many requests (e.g., transferring money) are inherently nonidempotent, so something else is needed.
- Another method is to have the client's kernel assign each request a sequence number. By having each server's kernel keep track of the most recently received sequence number from each client's kernel that is using it, the server's kernel can tell the difference between an original request and a retransmission and can refuse to carry out any request a second time.
- An additional safeguard is to have a bit in the message header that is used to distinguish initial requests from retransmissions (the idea being that it is always safe to perform an original request; retransmissions may require more care).

Server Crashes

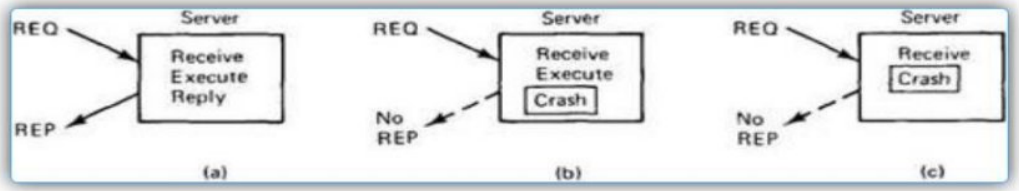


Fig. 2-24. (a) Normal case. (b) Crash after execution. (c) Crash before execution.

- The next failure on the list is a server crash. It too relates to idempotency, but unfortunately it cannot be solved using sequence numbers.
- Fig. 2-24(a)- A request arrives, is carried out, and a reply is sent. Now consider Fig. 2-24(b). A request arrives and is carried out, just as before, but the server crashes before it can send the reply. Finally, look at Fig. 2-24(c). Again a request arrives, but this time the server crashes before it can even be carried out.
- The annoying part of Fig. 2-24 is that the correct treatment differs for (b) and (c). In (b) the system has to report failure back to the client (e.g., raise an exception), whereas in (c) it can just retransmit the request. The problem is that the client's kernel cannot tell which is which. All it knows is that its timer has expired.
- Three schools of thought exist on what to do here. One philosophy is to wait until the server reboots (or rebinds to a new server) and try the operation again. The idea is to keep trying until a reply has been received, then give it to the client. This technique is called at least once semantics and guarantees that the RPC has been carried out at least one time, but possibly more.
- The second philosophy gives up immediately and reports back failure. This way is called at most once semantics and guarantees that the rpc has been carried out at most one time, but possibly none at all.
- The third philosophy is to guarantee nothing. When a server crashes, the client gets no help and no promises. The RPC may have been carried out anywhere from 0 to a large number of times. The main virtue of this scheme is that it is easy to implement.

Client Crashes

- What happens if a client sends a request to a server to do some work and crashes before the server replies? At this point a computation is active and no parent is waiting for the result. Such an unwanted computation is called an **orphan**.
- Orphans can cause a variety of problems. As a bare minimum, they waste CPU cycles. They can also lock files or otherwise tie up valuable resources. Finally, if the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result.
- What can be done about orphans? In **solution 1**, before a client stub sends an RPC message, it makes a log entry telling what it is about to do. The log is kept on disk or some other medium that survives crashes. After a reboot, the log is checked and the orphan is **explicitly killed off**. This solution is called **extermination**.
- The disadvantage of this scheme is the horrendous expense of writing a disk record for every RPC. Furthermore, it may not even work, since orphans themselves may do RPCs, thus creating grandorphans or further descendants that are impossible to locate. Finally, the network may be partitioned, due to a failed gateway, making it impossible to kill them, even if they can be located.
- In **solution 2**, called **reincarnation**, all these problems can be solved without the need to write disk records. The way it works is to divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations are killed. Of course, if the network is partitioned, some orphans may survive. However, when they report back, their replies will contain an obsolete epoch number, making them easy to detect.
- **Solution 3** is a variant on this idea, but **less Draconian**. It is called gentle reincarnation. When an epoch broadcast comes in, each machine checks to see if it has any remote computations, and if so, tries to locate their owner. Only if the owner cannot be found is the computation killed.
- **Solution 4**, **expiration**, in which each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum, which is a nuisance. On the other hand, if after a crash the server waits a time T before rebooting, all orphans are sure to be gone. The problem to be solved here is choosing a reasonable value of T in the face of RPCs with wildly differing requirements.
- **In practice, none of these methods are desirable**. Worse yet, killing an orphan may have unforeseen consequences. For example, suppose that an orphan has obtained locks on one or more files or data base records. If the orphan is suddenly killed, these locks may remain forever. Also, an orphan may have already made entries in various remote queues to start up other processes at some future time, so even killing the orphan may not remove all traces of it.

Lost Request Messages

- The second item on the list is **dealing with lost request messages**.
- This is the **easiest one to deal with**: just have the kernel start a timer when sending the request. If the timer expires before a reply or acknowledgement comes back, the kernel sends the message again.
- If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine.
- Of course, so many request messages are lost that the kernel gives up and **falsely concludes** that the server is down, in which case we are back to "**Cannot locate server**."

#9: Remote Procedure Call - Performance

- **Success** or **failure** of a Distributed System **depends on** its **Performance.**
  - Performance is critically dependent on the ***"Speed of the Communication*".***
    1. ****: Stands (50+Δ)% & Falls (50- Δ)%*** *with its* ***implementations*** *rather than with its abstract principles.*
    2. *One should analyze where the* ***time*** *is spent?*
- **Implementation** of Distributed System
  - Protocol Selection
  - Acknowledgement
  - Critical Path
  - Copying
  - Timer Management

#9.1:RPC Protocol Selection

- Criteria of Selection: It should gets the bits from the client's kernel to the server's kernel
- Connection-oriented protocol vs Connectionless protocol
- Standard general-purpose protocol vs Specifically designed for RPC
- Length of Packet and Message

#9.1.1:Connection-oriented Vs Connectionless Protocol

Connection-Oriented Protocol	Connectionless Protocol
After connection establishment, the client is bound to the server	No principle of connection establishment for long period. However session-wise pairing between two neighboring entities is required.
Same connection is used by all the traffic, in both directions.	The path used by all the traffic might be different
Communication is easier	Communication is easier is LAN, where most of the connections are of one hop length
When a kernel sends a message, the possibility of lost of the message and receiving of its ACK is not worrisome for it.	Loss of message, loss of ACK need extra work
This approach is very strong in WAN	Reliable in LAN
This is not suitable in LAN (∵ The extra s/w are forming hindrance in the LAN)	Suitable in small building LAN
Conclusion: Connection-oriented in WAN	**Conclusion**: Connectionless in LAN

#9.1.2:Standard Protocol Vs Specialized RPC

Standard Protocol (IP or UDP)	Specialized RPC
The protocol is already designed. Saves substantial work.	It is need to be invented, implemented, tested and embedded in existing systems. Considerably more work.
Many implementations are available. Saves work and time.	More work and time
Communication is easier	Communication need to be tested in the networks.
Most of the UNIX systems accept the packets of these protocol for communication purpose	Needs integration into existing UNIX systems
Existing networks also support IP and UDP packets	Need to be tested across all types of networks
Writing, executing and testing code using these protocols are straightforward.	Several phases of software testing is required.
IP is not an end-to-end protocol. It is executed on top of reliable TCP. So, it bounce back several times in the network.	Specialized RPC would avoid bouncing back of the packets.
IP has 13 header fields. 3 are essential (Src_Addr, Dstn_addr, Pkt_len). Header checksum is time consuming	Number of header fields may differ, according to the requirement of the problem.

#9.1.3:Packet and Message Length

- Size of file is 64K in a single 64K RPC would be **efficient**
- Size of file is 64K in a 64 1K RPCs is *not efficient*
- **Large size file** with **Maxflow** should be supported by both **protocol** and **network**
  - Sun Microsystem's limit is 8K (System level **constraints**)
  - Ethernet's limit is 1536 bytes (Network level **constraints**)
- So, a single RPC is required to be split over multiple packets, is an **overhead**.

#9.2:Acknowledgements

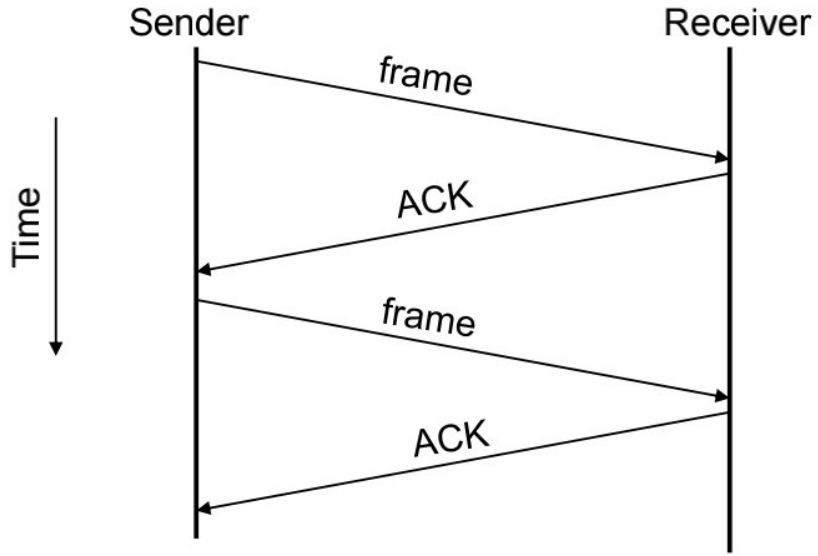
- When large RPCs have to be broken up into many small packets, then what should be the acknowledgement process?
  - Should individual packets be acknowledged? (***stop-and-wait- protocol***)
  - Acknowledge after receiving all the packets (***Blast Protocol***)
- ***

Automatic Repeat Request (ARQ) Algorithms

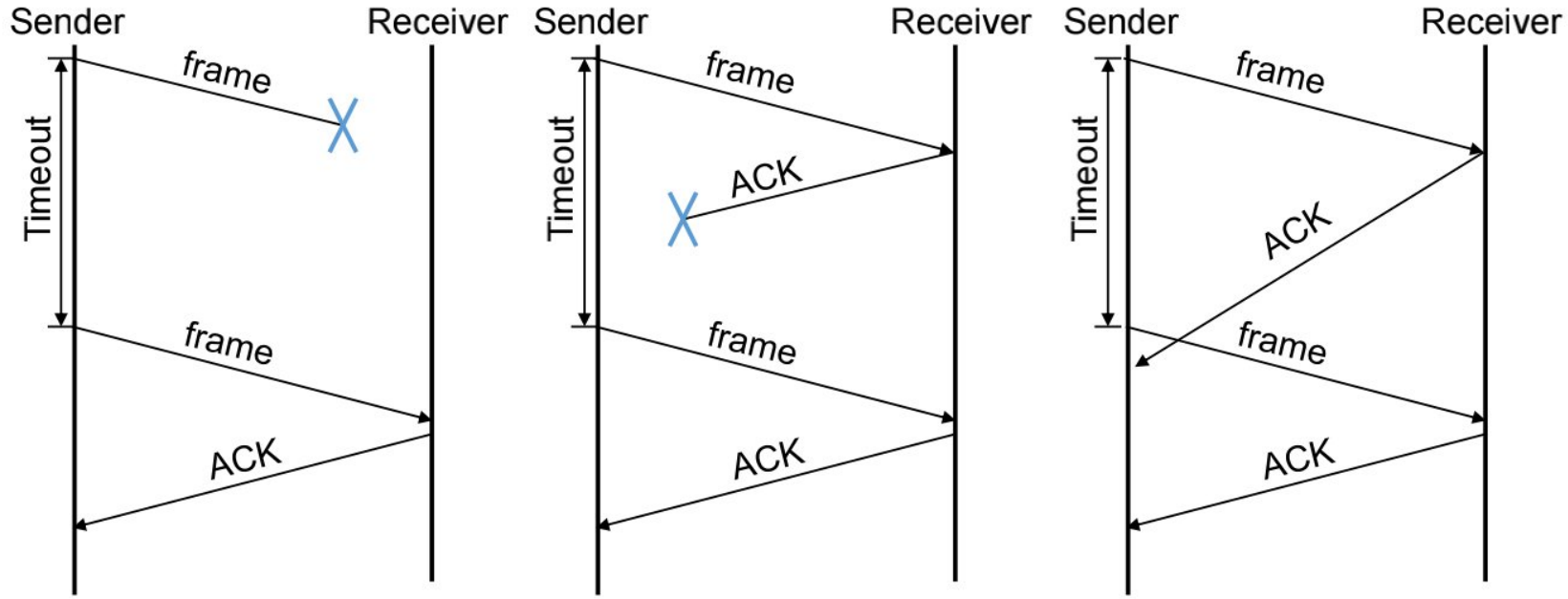
- Use two basic techniques:
  - Acknowledgements (ACKs)
  - Timeouts
- Two examples:
  - Stop-and-Wait
  - Sliding window

Stop-and-Wait

- Receiver: send an acknowledge (ACK) back to the sender upon receiving a packet (frame)
- Sender: excepting first packet, send a packet only upon receiving the ACK for the previous packet



What Can Go Wrong?



Frame lost → resent it on Timeout

ACK lost → resent packet

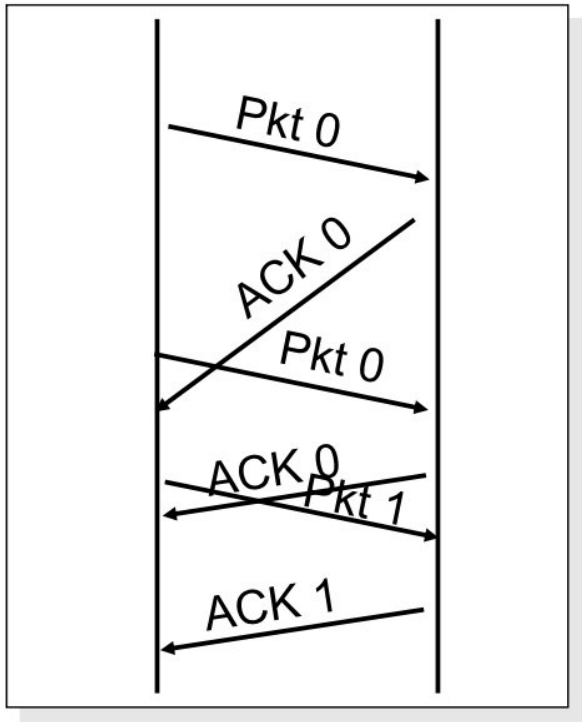
Need a mechanisms to detect duplicate packet

ACK delayed → resent packet

Need a mechanism to differentiate between ACK for current and previous packet

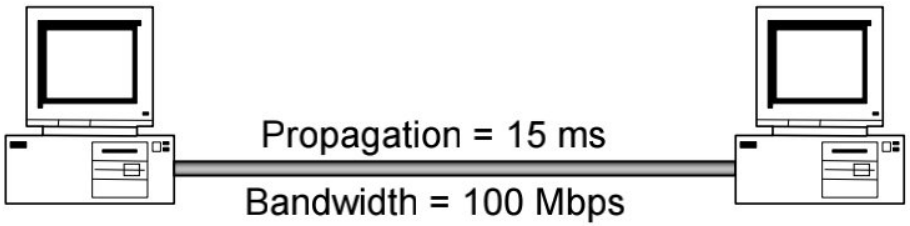
How to Recognize Retransmissions?

- Use sequence numbers
  - both packets and acks
- Sequence # in packet is finite → How big should it be?
  - For stop and wait?
- One bit – won't send seq #1 until received ACK for seq #0



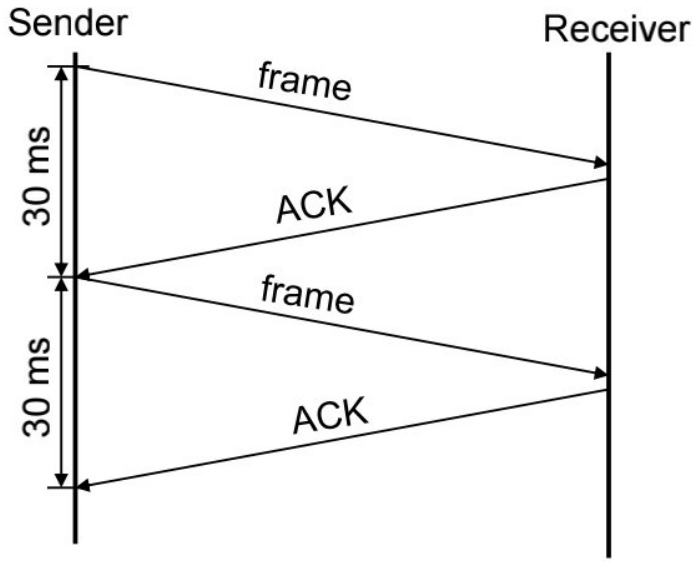
Stop-and-Wait Disadvantage

- May lead to inefficient link utilization
- Example: assume
  - One-way propagation = 15 ms
  - Bandwidth = 100 Mbps
  - Packet size = 1000 bytes → transmit = $(8*1000)/10^8$ = 0.08ms
  - Neglect queue delay → Latency = approx. 15 ms; RTT = 30 ms



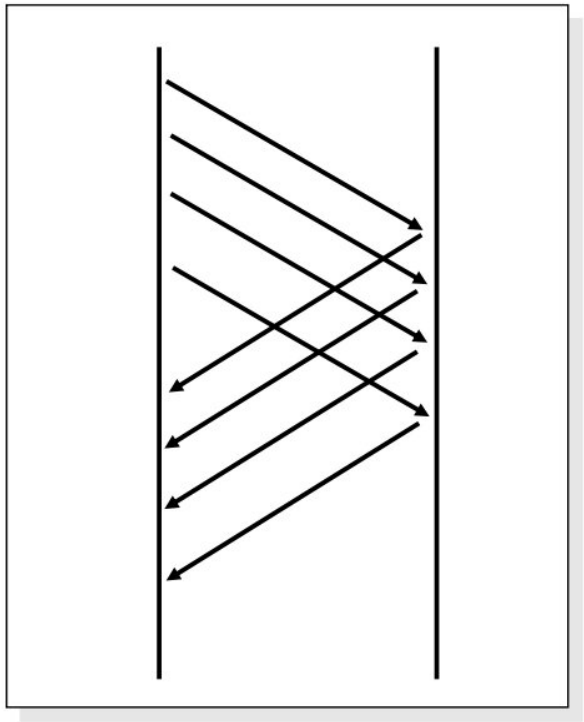
Stop-and-Go Disadvantage cont.

- Send a message every 30 ms → Throughput = (8*1000)/0.03 = 0.2666 Mbps
- Thus, the protocol uses less than 0.3% of the link capacity!



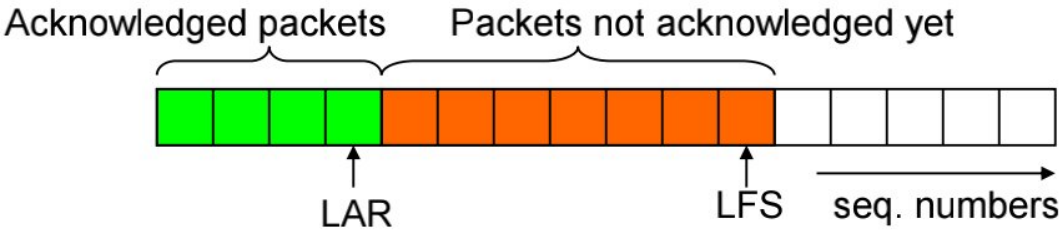
How to Keep the Pipe Full?

- Send multiple packets without waiting for first to be acked
  - Number of pkts in flight = window
- Reliable, unordered delivery
  - Several parallel stop & waits
  - Send new packet after each ack
  - Sender keeps list of unack'ed packets; resends after timeout
  - Receiver same as stop & wait
- How large a window is needed?



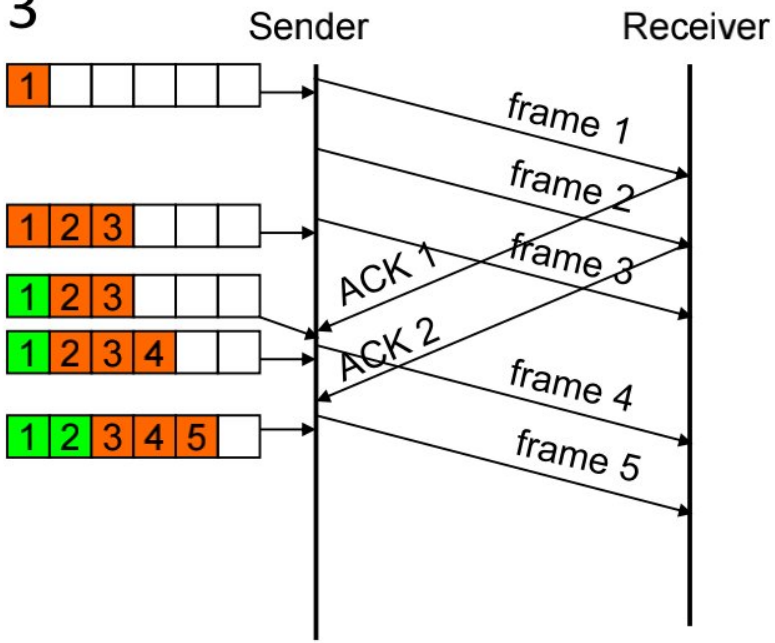
Sliding Window Protocol: Sender

- Each packet has a sequence number
  - Assume infinite sequence numbers for simplicity
- Sender maintains a window of sequence numbers
  - **SWS** (sender window size) – maximum number of packets that can be sent without receiving an ACK
  - **LAR** (last ACK received)
  - **LFS** (last frame sent)



Example

- Assume SWS = 3



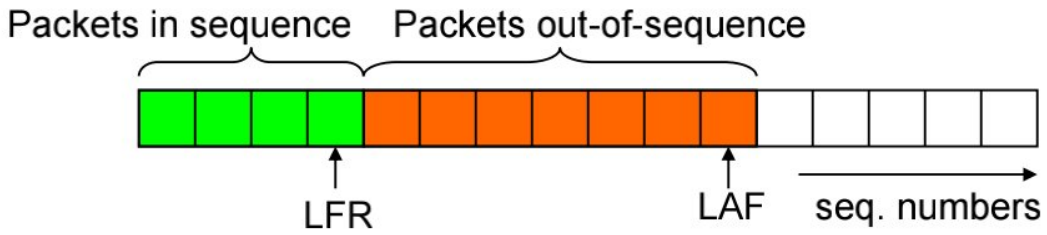
Note: usually ACK contains the sequence number of the first packet in sequence expected by receiver

Sliding Window Protocol: Receiver

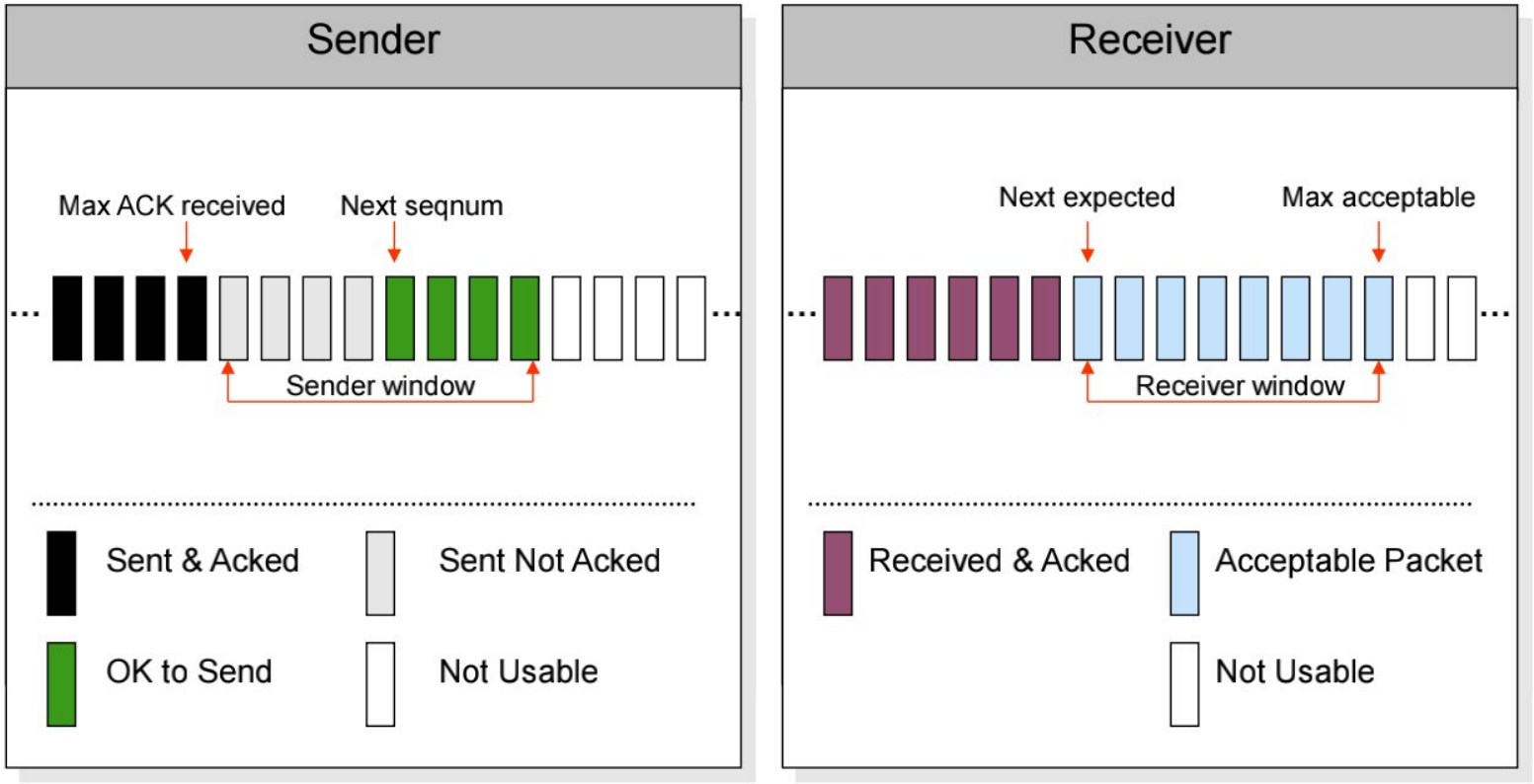
- Receiver maintains a window of sequence numbers
  - RWS (receiver window size) – maximum number of out-of-sequence packets that can received
  - LFR (last frame received) – last frame received in sequence
  - LAF (last acceptable frame)
  - LAF – LFR <= RWS

Sliding Window Protocol: Receiver

- Let seqNum be the sequence number of arriving packet
- If (seqNum <= LFR) or (seqNum >= LAF)
  - Discard packet
- Else
  - Accept packet
  - ACK largest sequence number seqNumToAck, such that all packets with sequence numbers <= seqNumToAck were received



Sender/Receiver State



Sequence Numbers

- How large do sequence numbers need to be?
  - Must be able to detect wrap-around
  - Depends on sender/receiver window size
- E.g.
  - Max seq = 7, send win=recv win=7
  - If pkts 0..6 are sent succesfully and all acks lost
    1. Receiver expects 7,0..5, sender retransmits old 0..6!!!
- Max sequence must be $\geq$ send window + recv window

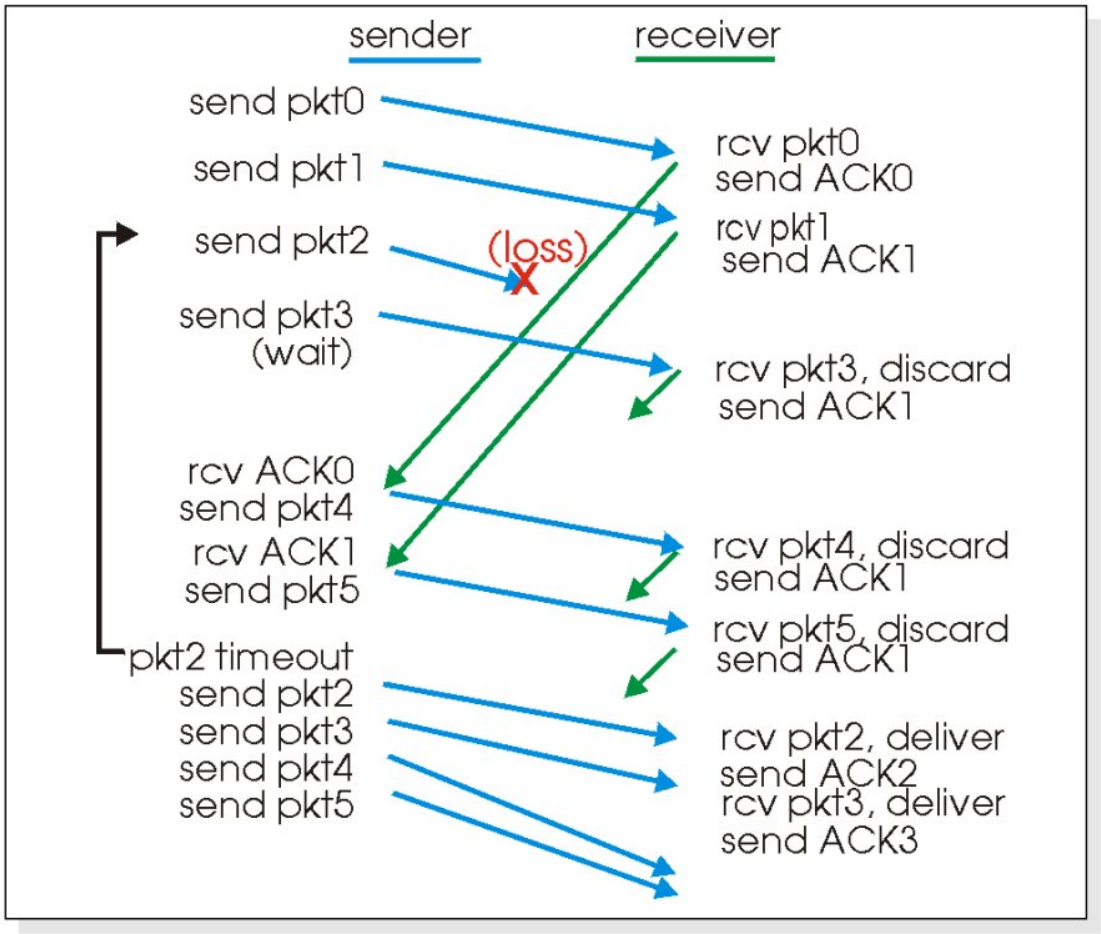
Cumulative ACK + Go-Back-N

- On reception of new ACK (i.e. ACK for something that was not acked earlier)
  - Increase sequence of max ACK received
  - Send next packet
- On reception of new in-order data packet (next expected)
  - Hand packet to application
  - Send cumulative ACK – acknowledges reception of all packets up to sequence number
  - Increase sequence of max acceptable packet

Loss Recovery

- On reception of out-of-order packet
  - Send nothing (wait for source to timeout)
  - Cumulative ACK (helps source identify loss)
- Timeout (Go-Back-N recovery)
  - Set timer upon transmission of packet
  - Retransmit all unacknowledged packets
- Performance during loss recovery
  - No longer have an entire window in transit
  - Can have much more clever loss recovery

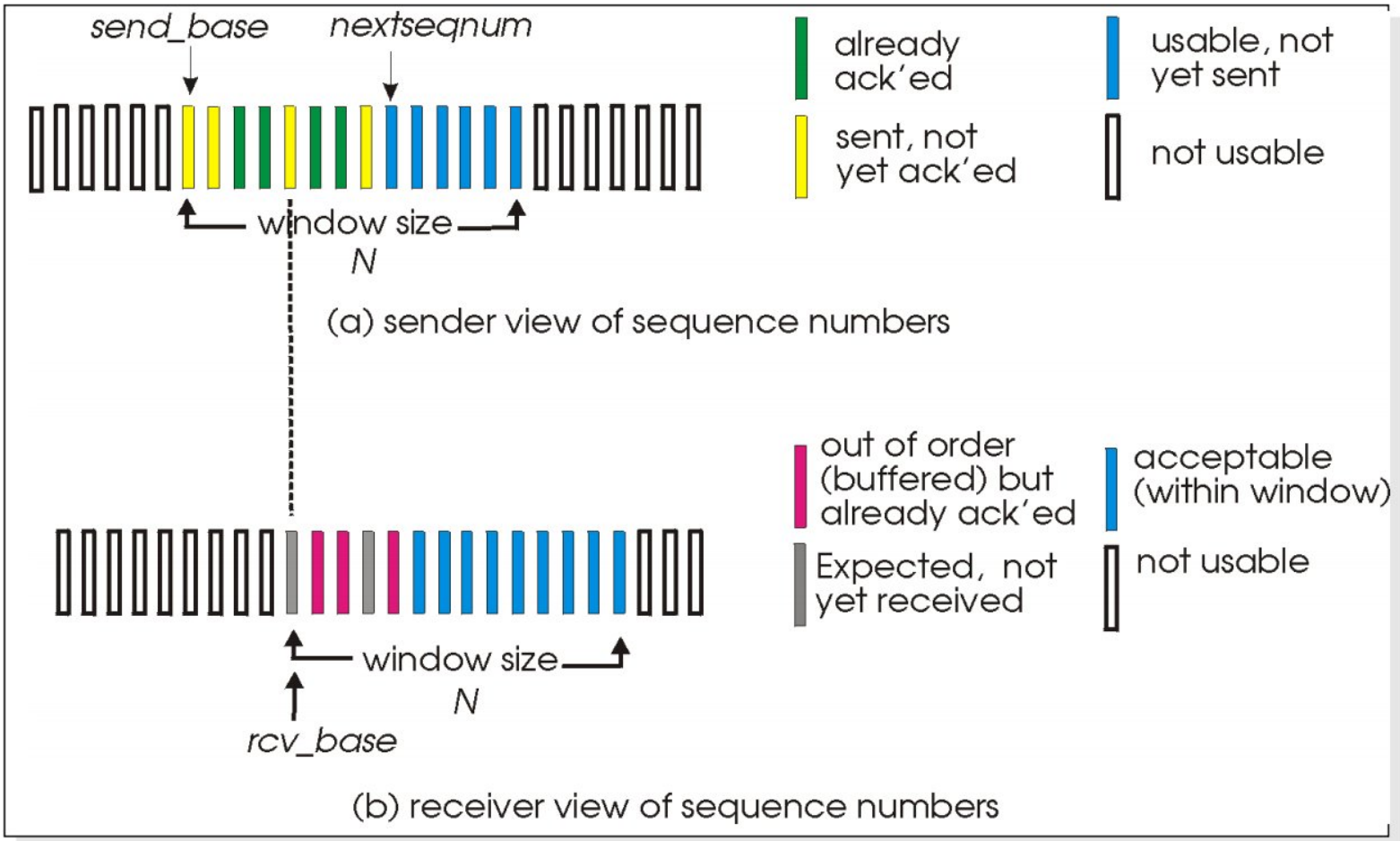
Go-Back-N in Action



Selective Ack + Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - Sender timer for each unACKed packet
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed packets

Selective Repeat: Sender, Receiver Windows



Summary of ARQ Protocols

- Mechanisms:
  - Sequence number
  - Timeout
  - Acknowledgement
- Sender window: fill the pipe
- Receiver window: handle out-of-order delivery

Many Nuances

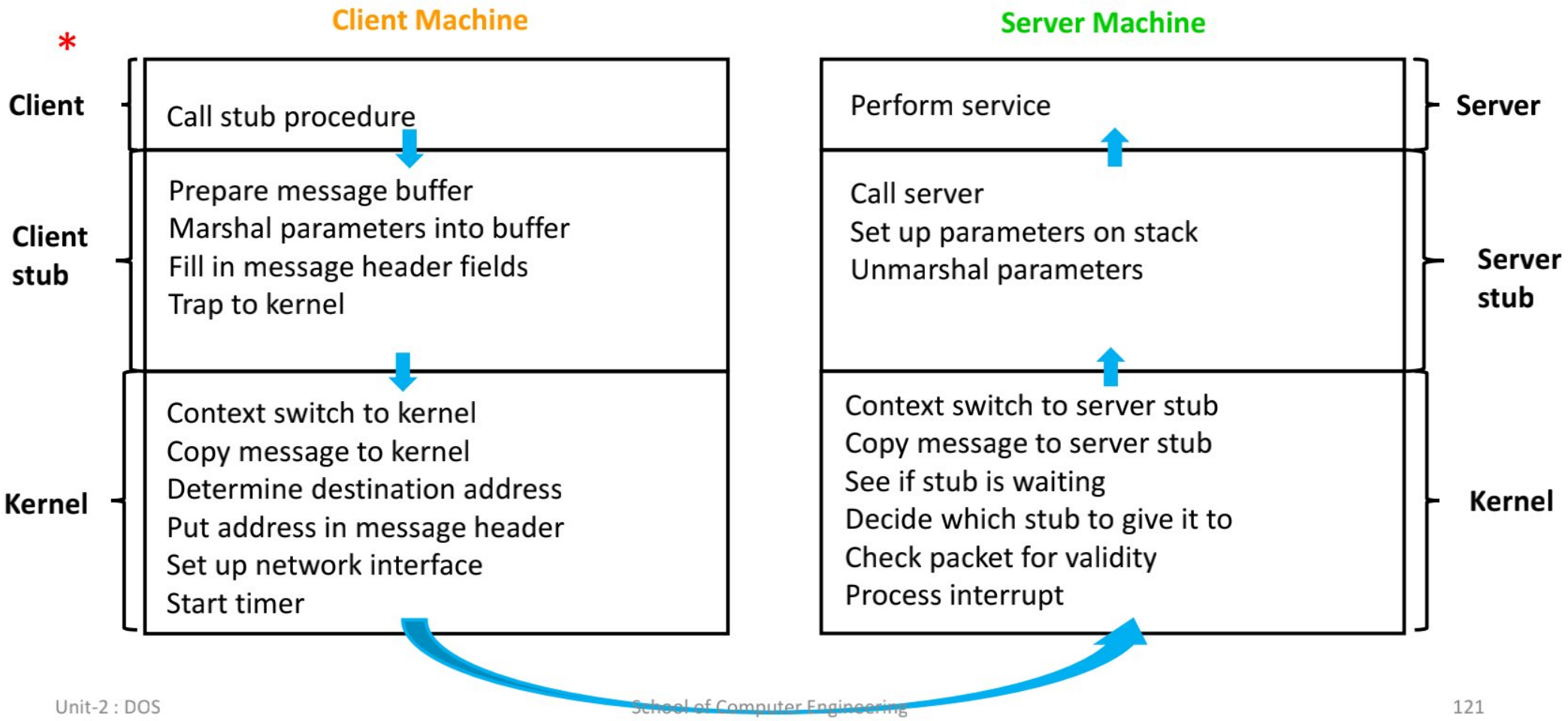
- What type of acknowledgements?
  - Selective acknowledgement
  - Cumulative acknowledgement
  - Negative acknowledgement
- How big should be the timeout value, Sliding Window Size (SWS), Receiving Window Size (RWS), sequence number field?
- Reliability mechanism used to implement other functions: flow control, congestion control
  - Function overloading introduces ambiguity and complexity

#9.3: Critical Path

- A critical path is defined as the sequence of instructions that is executed on every RPC (Eg. A client to a remote server)
- There are 14 steps in the RPC from Client-to-Server

1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to Kernel
7. Queue packet for transmission
8. Move packet to controller over the QBus
9. Ethernet transmission time
10. Get packet from controller
11. Interrupt service routine
12. Computer UDP Checksum
13. Context switch to user space
14. Server stub code

#9.3: Critical Path : Schematic View

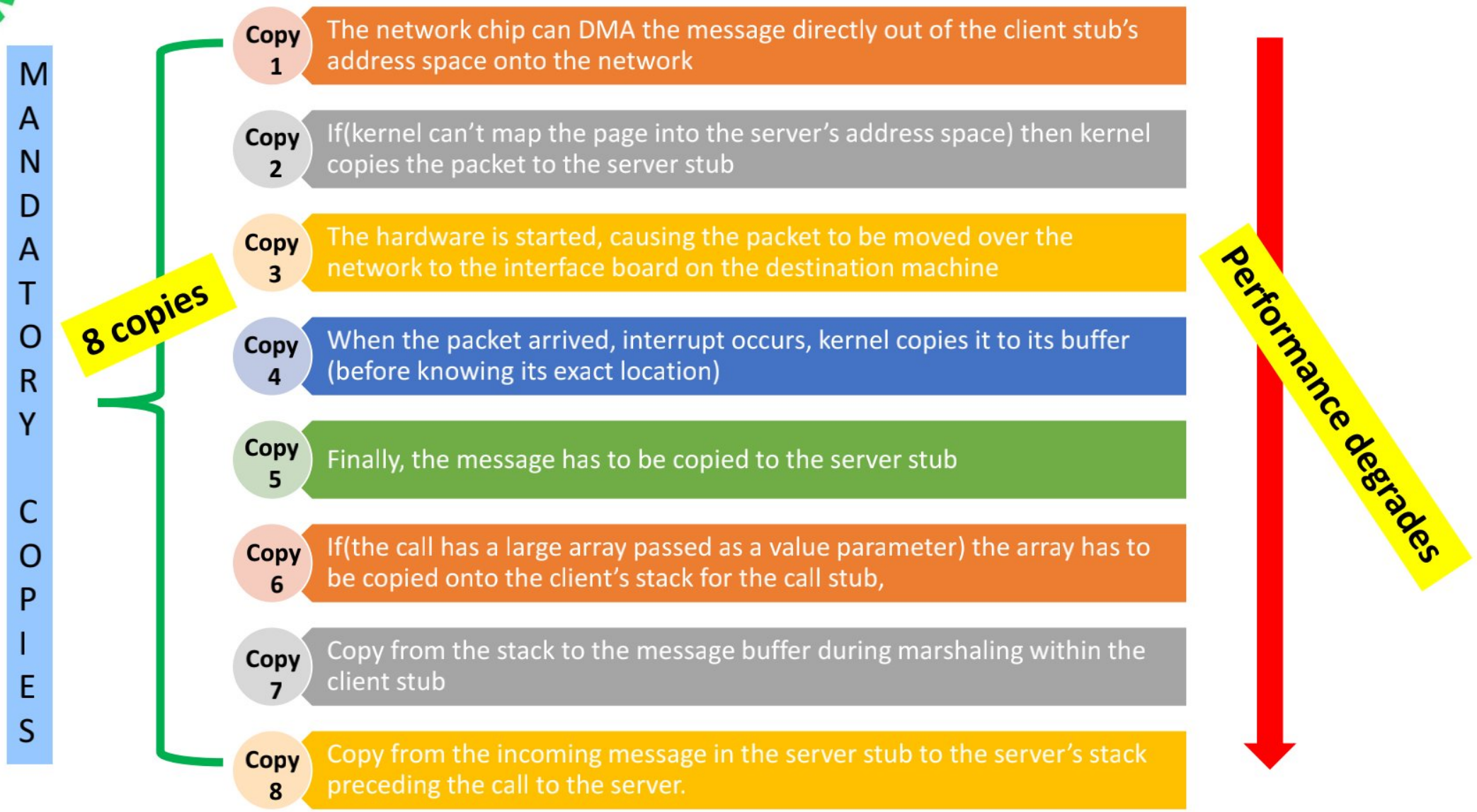


#9.3: Critical Path Cont.

- **Q:** Where is most of the time spent on the critical path?
- **Ans:**
  - Marshaling parameters and moving messages around
  - In case of null RPC, context switch to the server stub when packet arrives, the interrupt service routine, and moving the packets to the network interface for transmission
  - Managing a pool of buffers – which client stubs use to avoid having to fill in the entire UDP header every time.
  - All the machines don't share the same address space, so context switch and use of page table takes time
  - Entire RPC system has been carefully coded in assembly language and optimized. So, it is faster and saves time.
- *****

#9.4: Copying

- Copy is an issue in RPC [∵ it dominates execution time in RPC]
- **Q:** Why does this issue occur in RPC? [∵ In most of the systems the kernel and the user address spaces are disjoint. A message must be copied 1-to-8 times depending on the h/w, s/w and type of call].
- **Analysis:** In general a message is required to be copied many times during RPC communication. It hampers the performance of the RPC execution time.
- The 8 different copies degrade **the performance of RPC**.



#9.4: Copying Cont.

- How to eliminate unnecessary copying?
  - Using the hardware ***scatter-gather***
  - **At the Sender's side**: With cooperative hardware, a reusable packet header inside the kernel and a data buffer in user space can be put out onto the network with no internal copying on the sending side.
  - **At the Receiver's side**: Dump the message into a kernel buffer and let the kernel figure out what to do with it.
  - **In Operating Systems:** Using virtual memory
  - **Using mapping:** If(memory map can be updated in less time)
    1. Then, mapping is faster than copying
    2. Else, Not
- *****

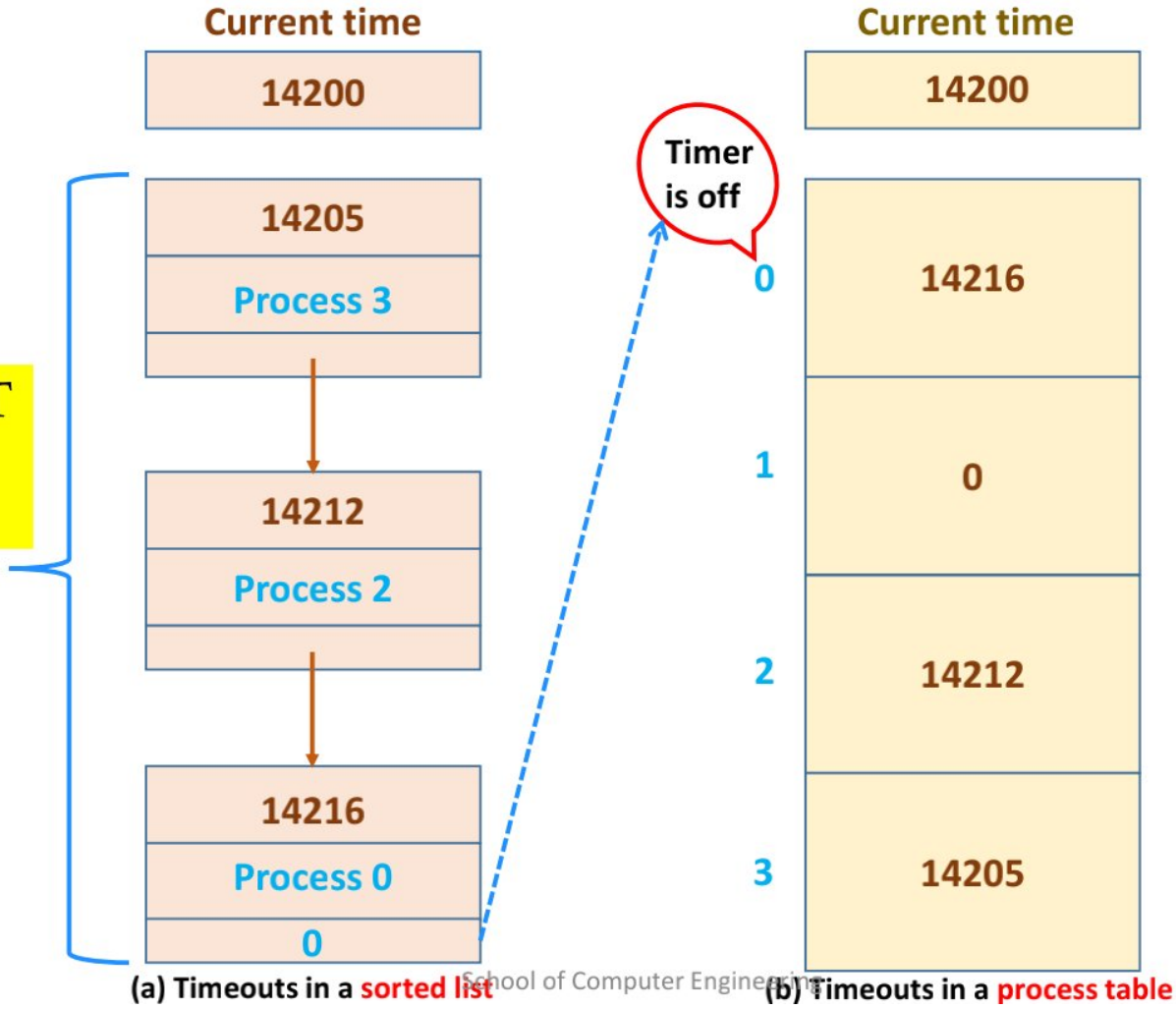
#9.5: Timer Management

- **Timer**: It is an automatic mechanism for activating an entity at a preset time.
- **Setting a timer** requires **building a data structure** specifying when the timer is to expire and what is to be done when that happens.
- The list of messages are in sorted order
- Timer starts just after message transmitted
- If(ACK or Reply arrives before the timer expires)
  - Then the timeout entry must be located and removed from the list
- Timer value should be neither too high or too low
- Most systems maintains a **Process Table** to implement **Timer**

#9.5: Timer Management via Sorted List and Process Table

- *

SORTED LIST W.R.T. TIME OUT



(a) Timeouts in a sorted list (b) Timeouts in a process table

Explanation: In process table in stead of storing timeouts in a sorted linked list, each process table entry has a field for holding its timeouts. It is shown the left of the process table in blue color.

Working Principle: The kernel scans the entire process table, checks each timer value against the current time. If (T_{read}<=$T_{current}$) then it is processed and reset.

Note: *Sweep Algorithms* operates by periodically making a sequential pass through a process table.

Q&A

Homework Questions

1) Draw the block diagram of the OSI Reference model and the TCP/IP protocol suite. Explain the mechanism of synchronisation between the two models.
2) Explain the functionalities of Hub, Bridge, Unmanaged switch, Managed switch, Router, Brouter and Gateway network devices.
3) An ATM system is transmitting cells at the OC-3 rate. Each packet is 1024 bytes long and thus, fits into a cell. An interrupt takes 1 µsec. What fraction of the CPU is devoted to interrupt handling?
4) The SPARC chip uses a 32-bit word in big endian format. If a SPARC sends the integer 2 to a 486, which is little endian, what numerical value does the 486 see?

Homework Questions

1. Suppose that the time to do a null RPC (i.e., 0 data bytes) is 1.0 msec, with an additional 1.5 msec for every 1K of data. How long does it take to read 32K from the file server in a single 32K RPC? How about a 32 1K RPC?
2. Imagine that in a particular distributed system, all the machines are redundant multiprocessors, so that the possibility of a machine crashing is so low that it can be ignored. Devise a simple method for implementing global time-ordered atomic broadcast using only unicasting.

Thank You!