

Distributed System

Slides prepared by: Er. Roshan Kandel

Syllabus

Course Title: Distributed Systems (3 Cr.)

Course Code: CACS352

Year/Semester: III/VI

Class Load: 4 Hrs. / Week (Theory: 3Hrs. Tutorial: 1 Hr.)

Course Description

The course introduces basic knowledge to give an understanding how modern distributed systems operate. The focus of the course is on distributed algorithms and on practical aspects that should be considered when designing and implementing real systems. Some topics covered during the course are causality and logical clocks, synchronization and coordination algorithms, transactions and replication, and end-to-end system design. In addition, the course explores recent trends exemplified by current highly available and reliable distributed systems.

Course objectives

The objective of the course is to make familiar with different aspect of the distributed system, middleware, system level support and different issues in designing distributed algorithms and finally systems.

Course Contents

Unit 1. Introduction	4 Hrs.
1.1 Characteristics	
1.2 Design Goals	
1.3 Types of Distributed Systems	
1.4 Case Study: The World Wide Web	

Syllabus contd...

Unit 2. Architecture	4 Hrs.
2.1 Architectural Styles	
2.2 Middleware organization	
2.3 System Architecture	
2.4 Example Architectures	
Unit 3. Processes	6 Hrs.
3.1 Threads	
3.2 Virtualization	
3.3 Clients	
3.4 Servers	
3.5 Code Migration	
Unit 4. Communication	5 Hrs.
4.1 Foundations	
4.2 Remote Procedure Call	
4.3 Message-Oriented Communication	
4.4 Multicast Communication	
4.5 Case Study: Java RMI and Message Passing Interface (MPI)	
Unit 5. Naming	5 Hrs.
5.1 Name Identifiers, and Addresses	
5.2 Structured Naming	
5.3 Attribute-based naming	
5.4 Case Study: The Global Name Service	

Syllabus contd...

Unit 5. Coordination	7 Hrs.
6.1 Clock Synchronization	
6.2 Logical Clocks	
6.3 Mutual Exclusion	
6.4 Election Algorithm	
6.5 Location System	
6.6 Distributed Event Matching	
6.7 Gossip-based coordination	
Unit 7. Consistency and Replication	5 Hrs.
7.1 Introduction	
7.2 Data-centric consistency models	
7.3 Client-centric consistency models	
7.4 Replica management	
7.5 Consistency protocols	
7.6 Caching and Replication in Web	
Unit 8. Fault Tolerance	5 Hrs.
8.1 Introduction to fault tolerance	
8.2 Process resilience	
8.3 Reliable client-server communication	
8.4 Reliable group communication	
8.5 Distributed commit	
8.6 Recovery	
Unit 9. Security	4 Hrs.
9.1 Introduction to security	
9.2 Secure channels	
9.3 Access control	
9.4 Secure naming	
9.5 Security Management	

Syllabus contd...

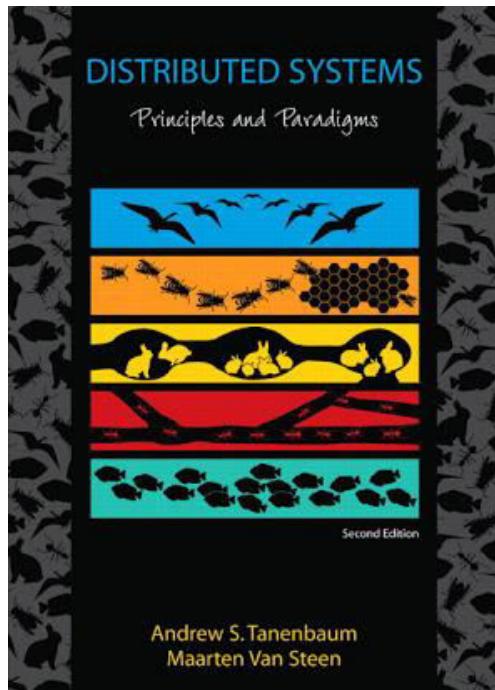
References:

1. A.S. Tanenbaum, M. VanSteen, "Distributed Systems", Pearson Education.
2. George Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems Concepts and Design", Third Edition, Pearson Education.
3. Mukesh Singhal, "Advanced Concepts in Operating Systems", McGraw-Hill Series in Computer Science.
4. Ajay D. Kshemkalyani, Mukesh Singhal, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press
5. Christian Cachin, Rachid Guerraoui, Luís, "Introduction to Reliable and Secure Distributed Programming", Springer

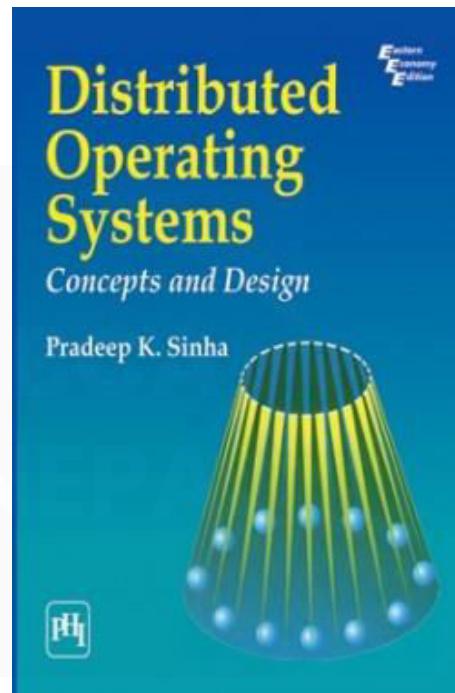
Unit-1

Introduction to Distributed System

Reference Books



Distributed systems: principles and paradigms | Andrew S.Tanenbaum, Maarten Van Steen



Distributed Operating Systems
Concepts and Design
By Pradeep K. Sinha, PHI

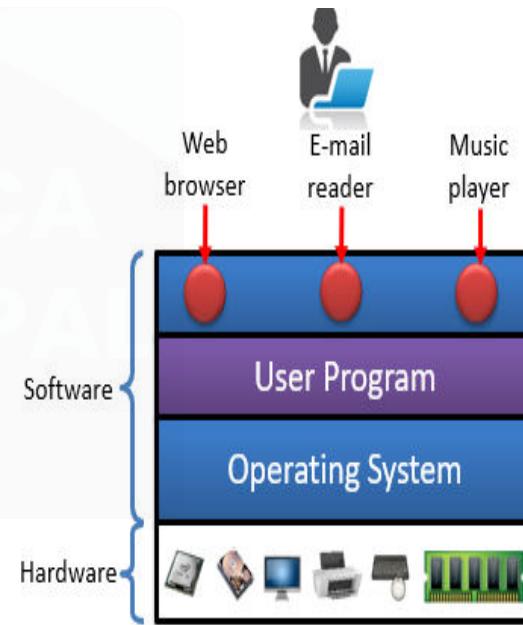
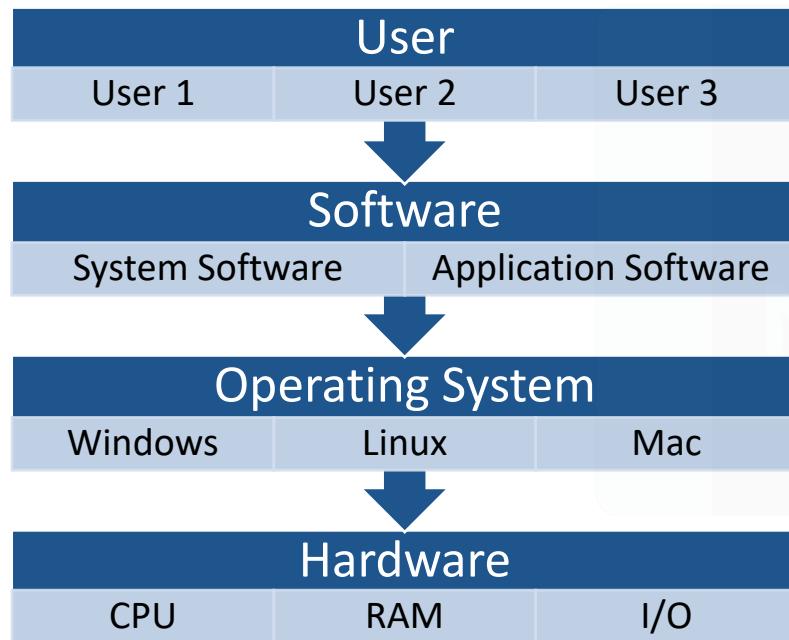
What is Operating System?

- ▶ An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs.
- ▶ Example:



What is Operating System?

- ▶ It is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



OS Example



← Regular OS

- ▶ An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs.
- ▶ Simple (No rules to follow).

Evolution of Modern OS

► First Generation OS

→ System:

- Centralized OS

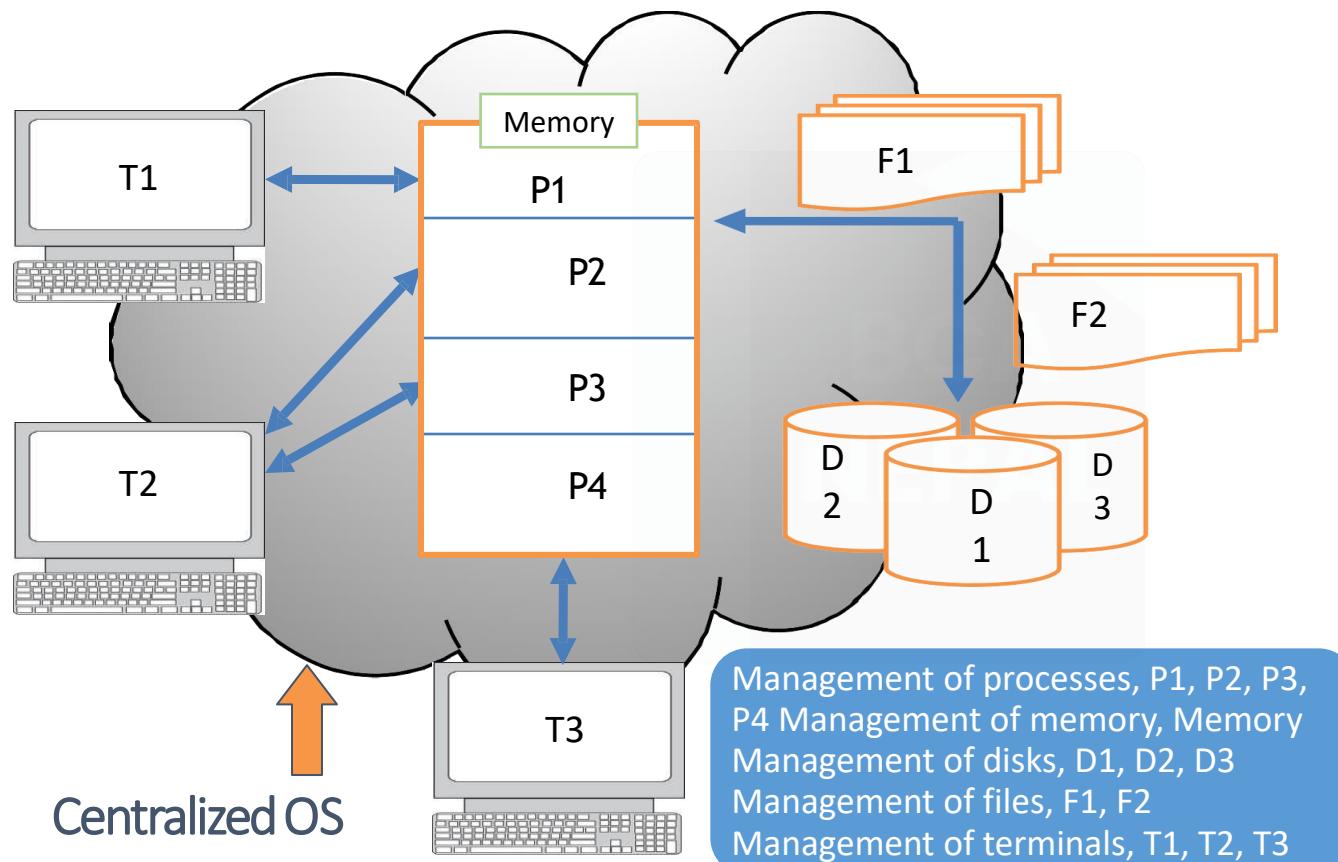
→ Characteristics:

- Process Management
- Memory Management
- I/O Management
- File Management

→ Goals:

- Resource Management

Centralized OS



Evolution of Modern OS

► Second Generation OS

→ System:

- Network OS(NOS)

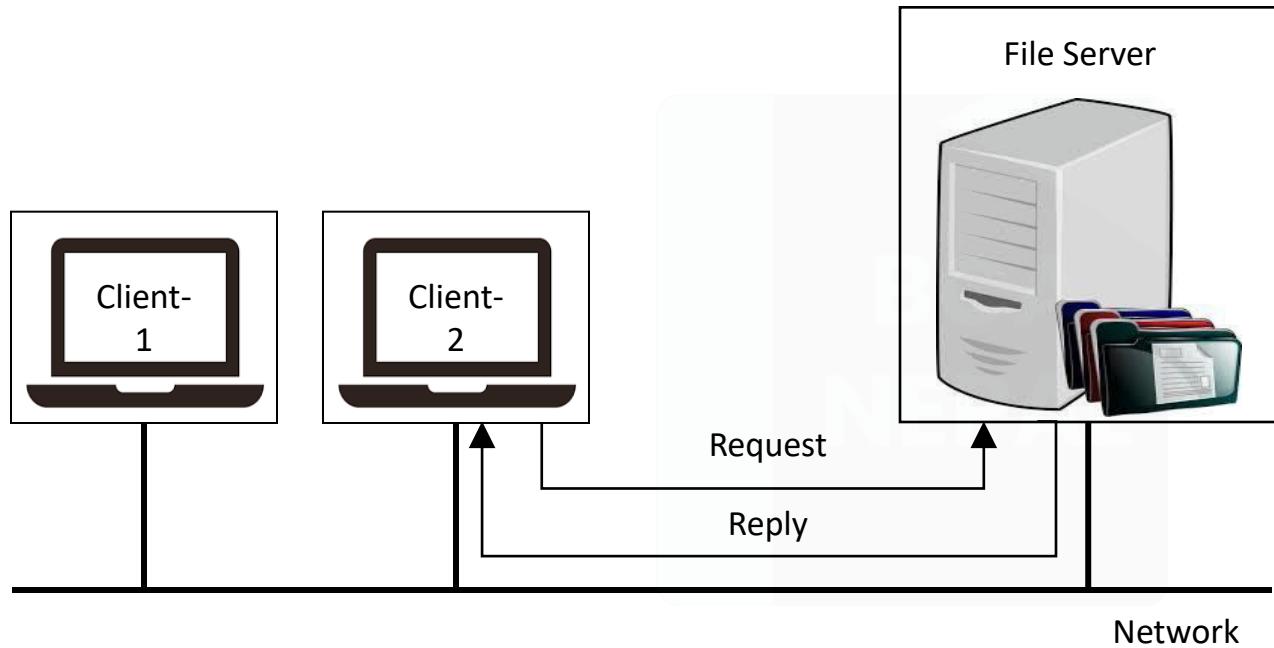
→ Characteristics:

- Remote access
- Information exchange
- Network browsing

→ Goals:

- Interoperability-Sharing of resources between the systems.

Network Operating System

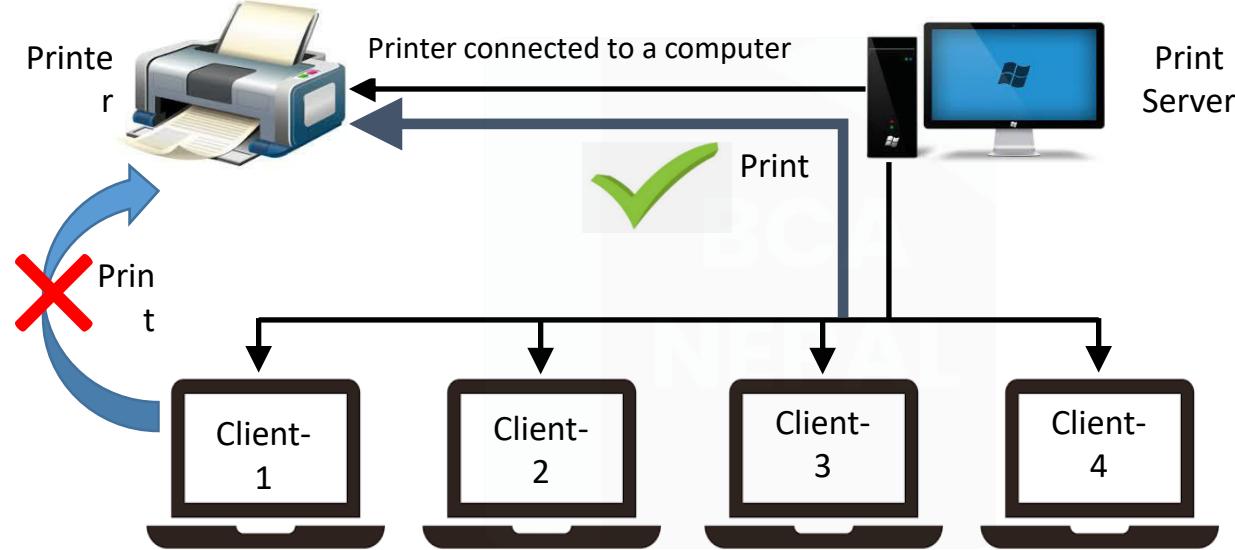


NOS Example



- ▶ When you want to interact with others.
- ▶ Introduces Network.
- ▶ Hard compared to regular OS (have to follow rules E.g., traffic rules).

NOS Example



Evolution of Modern OS

► Third Generation OS

→ System:

- **Distributed OS(DOS)**

→ Characteristics:

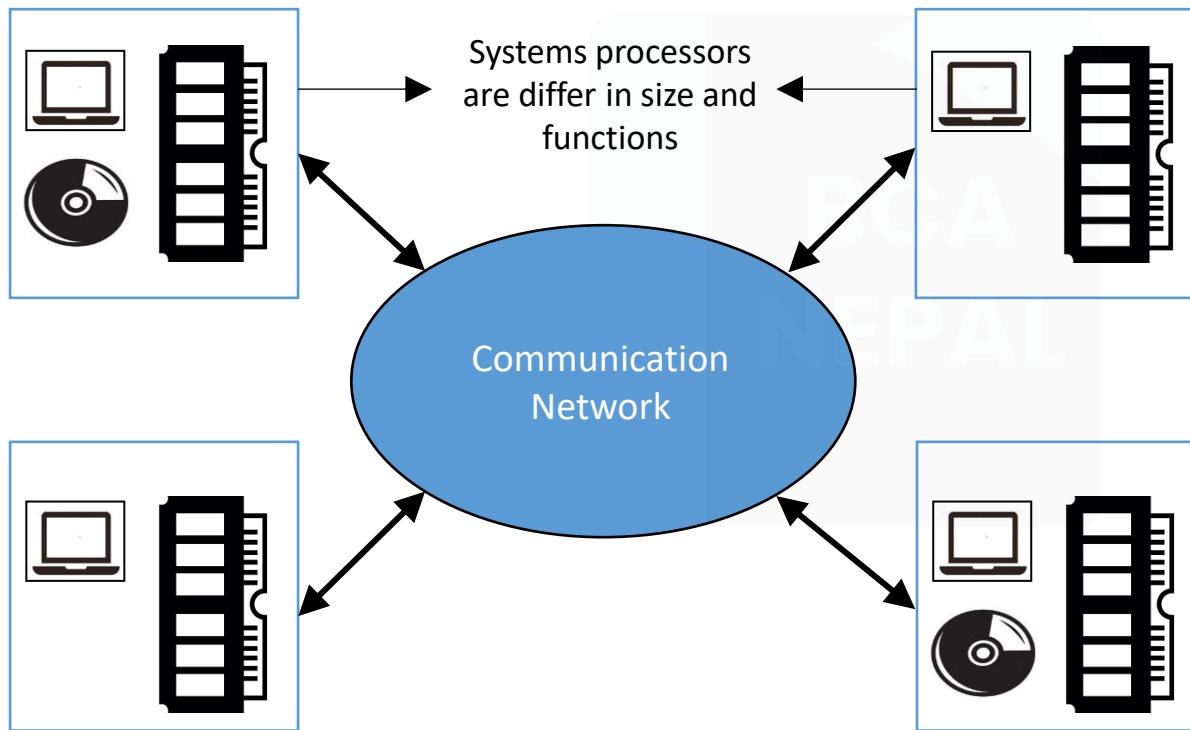
- Global View of Computational power, file system, name space, etc.

→ Goals:

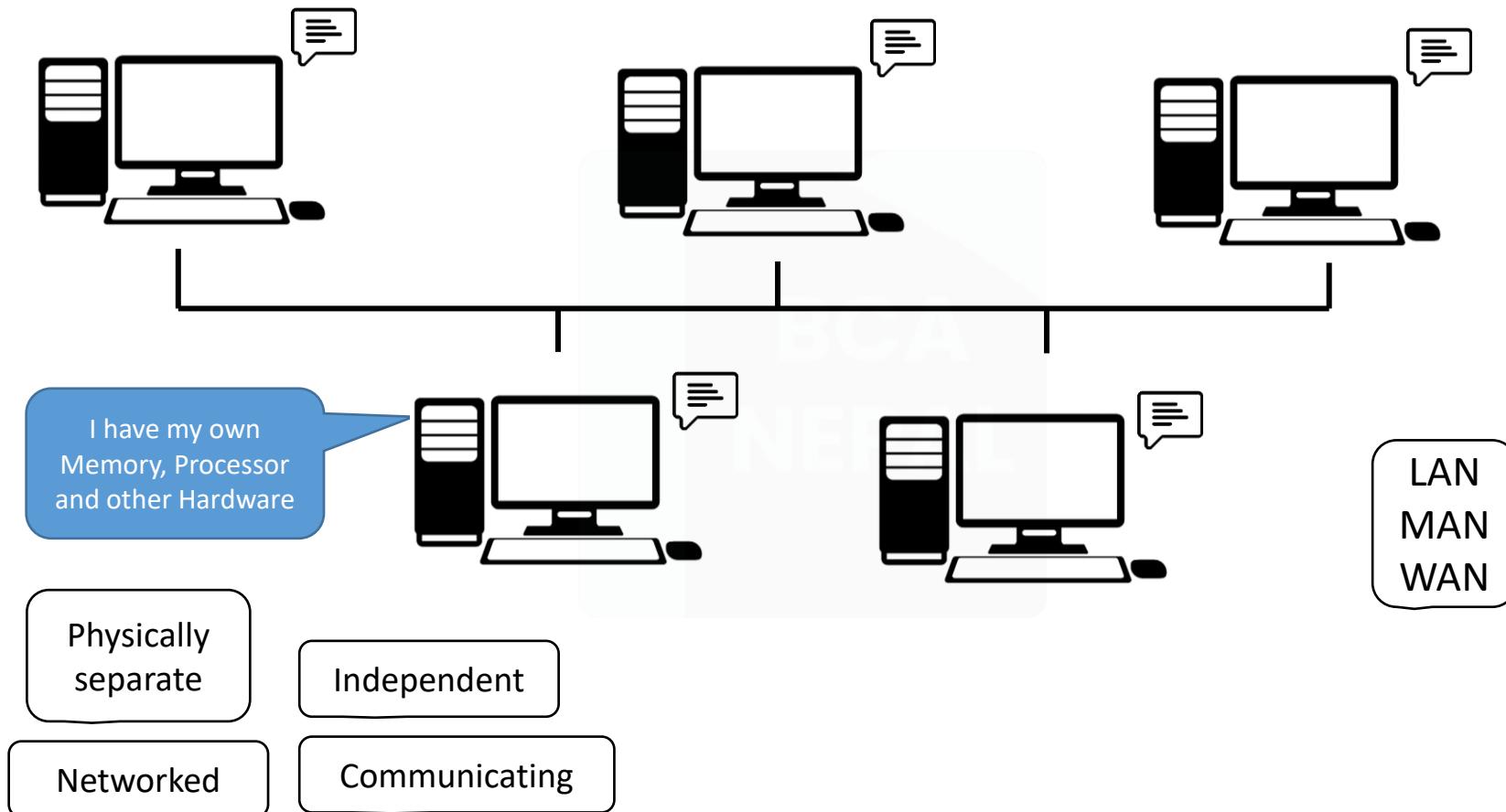
- Single computer view of multiple heterogeneous computer systems.

Distributed Operating System

- ▶ “A Distributed system is collection of independent computers which are connected through network.”



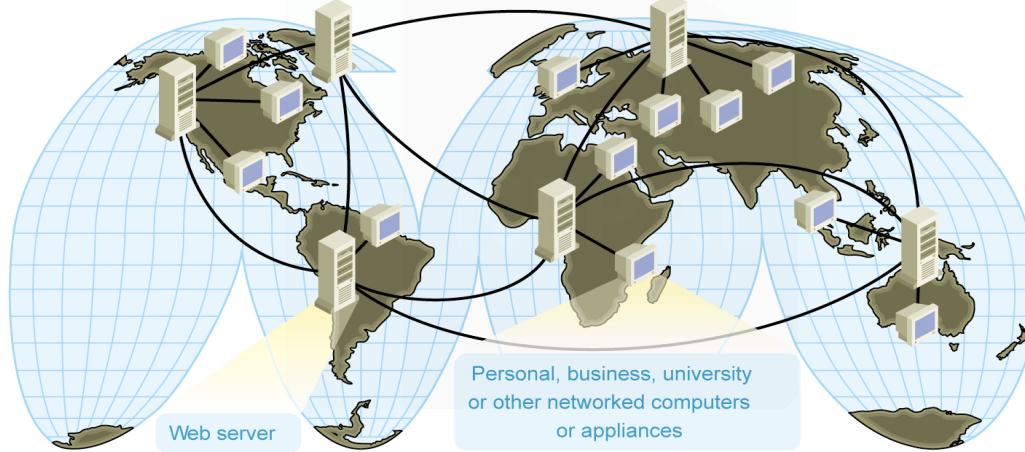
Distributed Operating System



Distributed Operating System

Definition by Coulouris, Dollimore, Kindberg and Blair

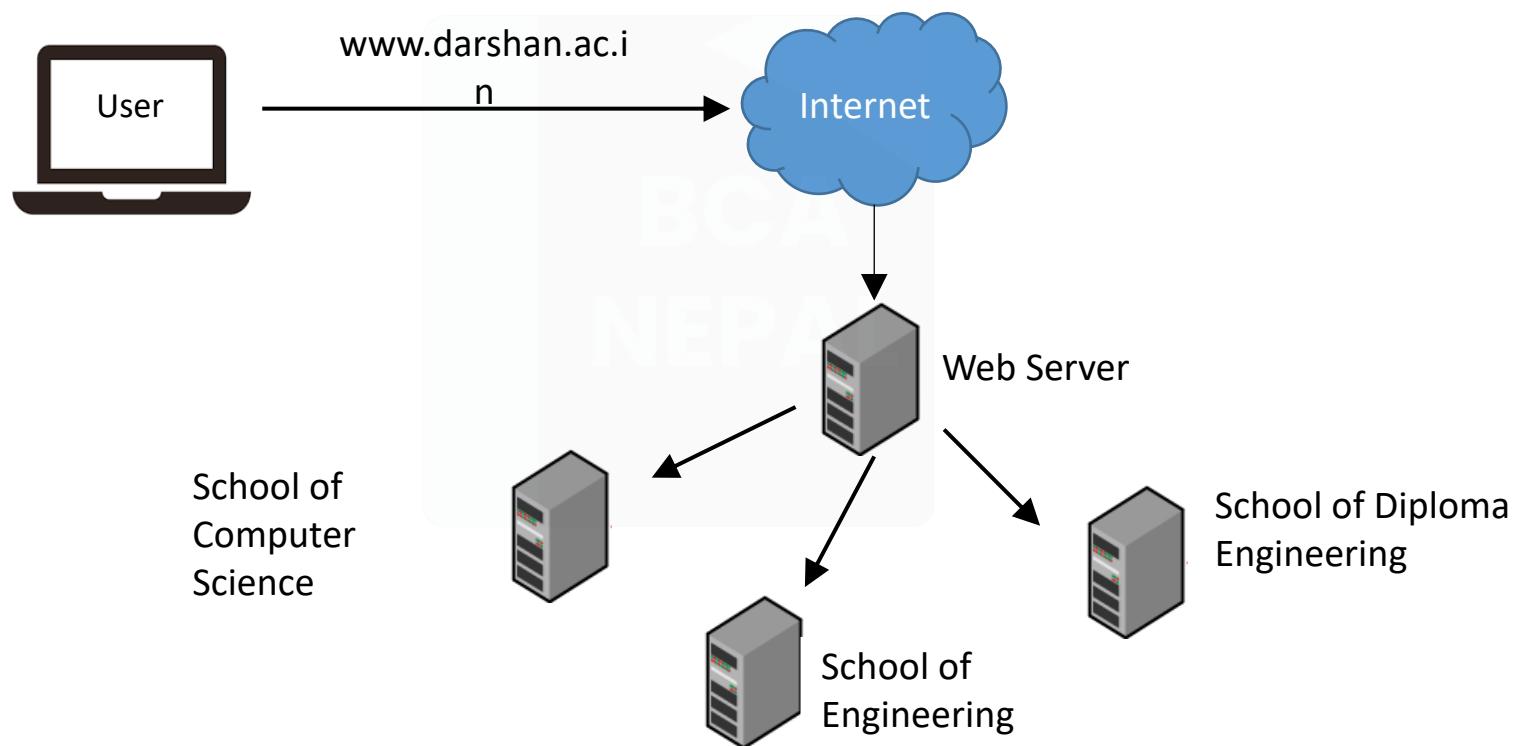
- ▶ “A distributed system is defined as one in which components at networked computers communicate and coordinate their actions only by passing messages.”
- ▶ “A Distributed system is collection of independent computers which are connected through network.”



- ▶ This system looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).

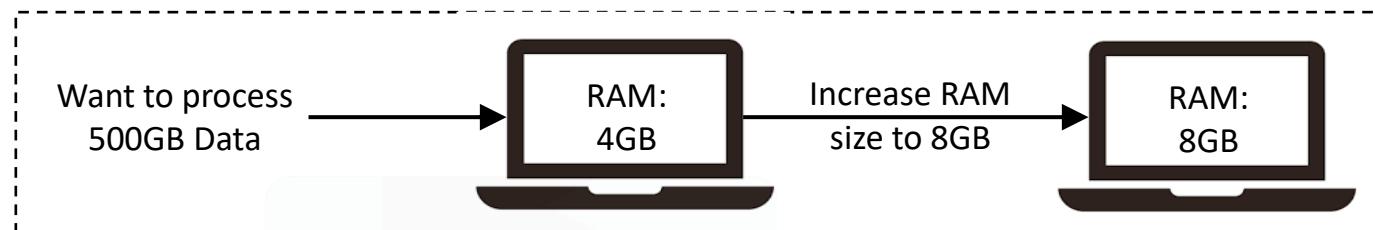
Distributed Operating System

- ▶ A great example of distributed system is the web page of Darshan University.

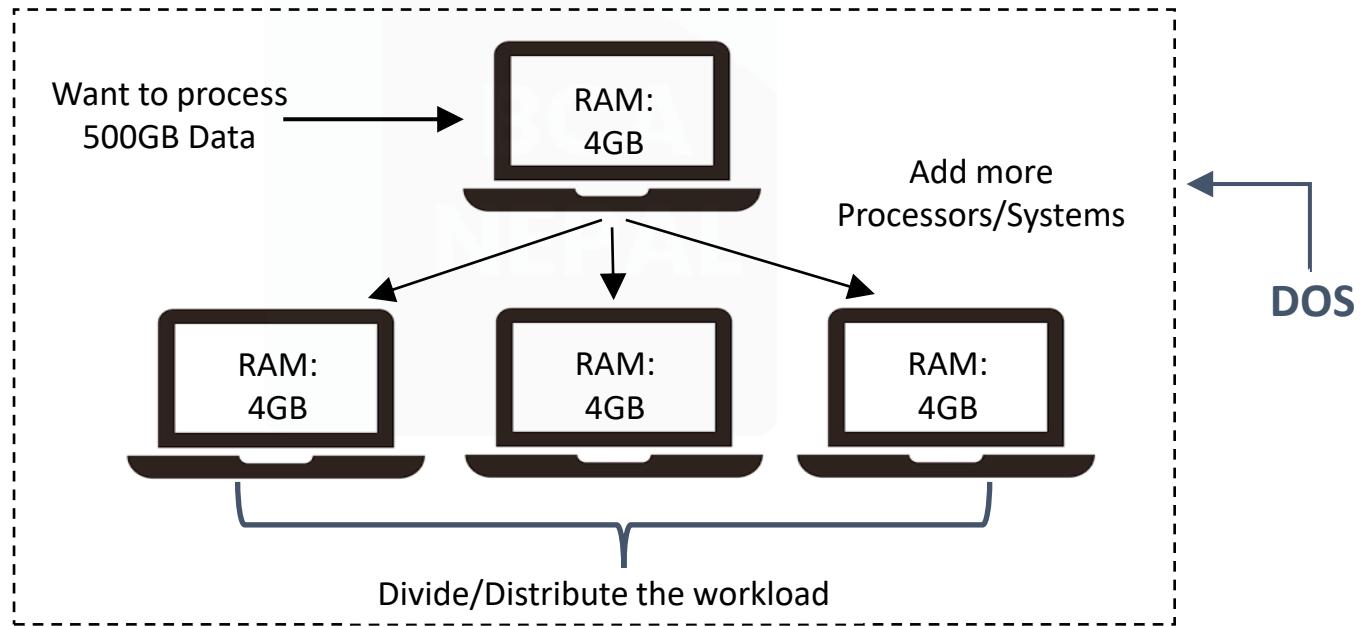


Distributed Operating System

Scenario-1: Vertical Scaling



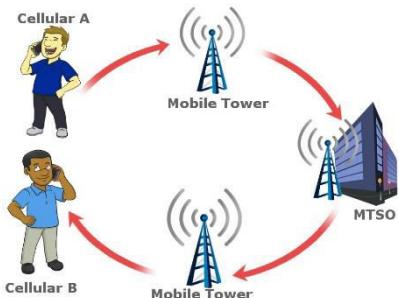
Scenario-2: Horizontal Scaling



Examples of Distributed Systems

- ▶ From the definition, Distributed Systems also looks the same as **single system**.
- ▶ Let us say about **Google Web Server**, from users perspective while they submit the searched query, they assume google web server as a single system.
- ▶ Just visit google.com, then search.
- ▶ However, under the hood Google builds a lot of servers even distributes in different geographical area to give you a search result within few seconds.
- ▶ So the Distributed Systems does not make any sense for normal users.

Examples of Distributed Systems



Telephone networks and cellular networks



ATM machines



Computer network such internet



Mobile Computing

Examples of Distributed Systems

▶ Web Search Engines:

- ▶ Major growth industry in the last decade.
- ▶ 10 billion per month for global number of searches.
- ▶ e.g. Google distributed infrastructure

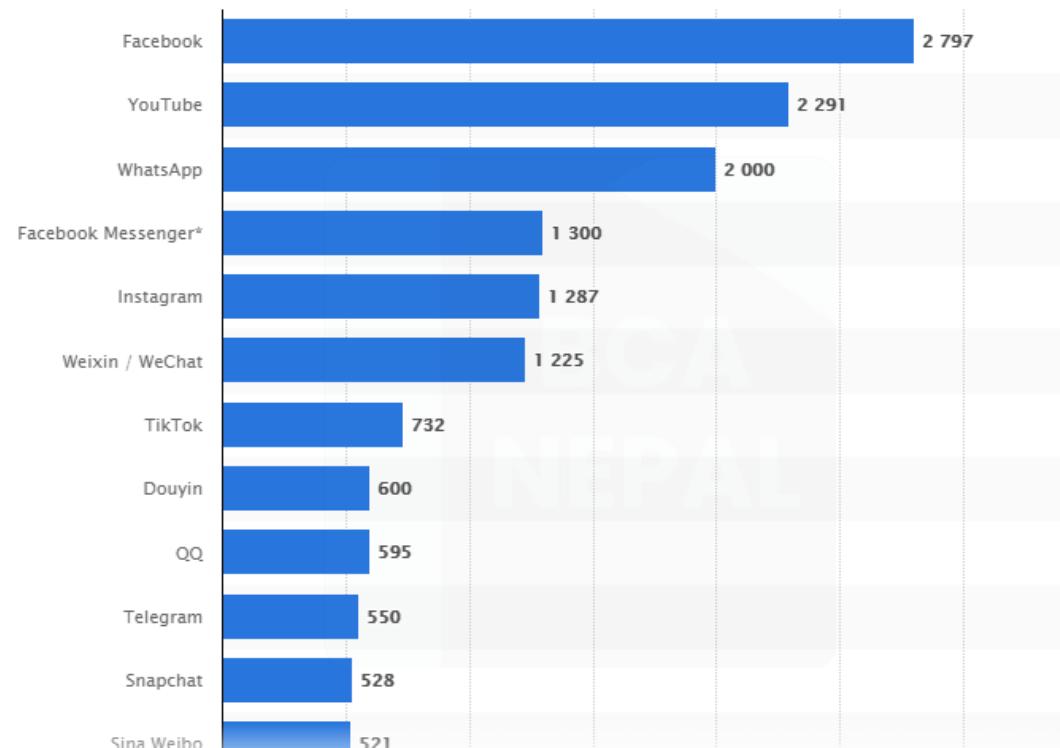


▶ Massively multiplayer online games:

- ▶ Large number of people interact through the Internet with a virtual world.
- ▶ Challenges include fast response time, real-time propagation of events.



Most popular social networks worldwide as of April 2021



Number of active users monthly (in millions)

Why Distributed Operating System?

- ▶ Facebook, currently, has 2.7 billion active monthly users.
- ▶ Google performs at least 2 trillion searches per year.
- ▶ About 500 hours of video is uploaded in Youtube every minute.
- ▶ A single system would be unable to handle the processing. Thus, comes the need for Distributed Systems.
- ▶ The main answer is to cope with the extremely higher demand of users in both processing power and data storage.
- ▶ With this extremely demand, single system could not achieve it.
- ▶ There are many reasons that make distributed systems is viable such as high availability, scalability, resistant to failure, etc.

Why Distributed Operating System?

- ▶ It is Challenging/Interesting.

- ▶ **Partial Failures**

- ▶ Network
 - ▶ Node failures

- ▶ **Concurrency**

- ▶ Nodes execute in parallel.

- ▶ Messages travel asynchronously.



Parallel Computing

Network OS vs Distributed OS

Network Operating System	Distributed Operating System
A network operating system is made up of software and associated protocols that allow a set of computer network to be used together.	A distributed operating system is an ordinary centralized operating system but runs on multiple independent CPUs.
Environment users are aware of multiplicity of machines.	Environment users are not aware of multiplicity of machines.
Control over file placement is done manually by the user.	It can be done automatically by the system itself.
No implicit sharing of loads.	Sharing of loads between nodes(load balancing).
Network OS is highly scalable. (A new machine can be added very easily.)	Distributed OS is less scalable. (The process to add new hardware is complex.)

Network OS vs Distributed OS

Network Operating System

Performance is badly affected if certain part of the hardware starts malfunctioning.

Remote resources are accessed by either logging into the desired remote machine or transferring data from the remote machine to user's own machines.

Easy to Implement

Low Transparency

Distributed Operating System

It is more reliable or fault tolerant i.e. distributed operating system performs even if certain part of the hardware starts malfunctioning.

Users access remote resources in the same manner as they access local resources.

Difficult to Implement.

High Transparency

Distributed System Goals

The following are the main goals of distributed systems:

- ▶ **The relative simplicity of the software** - Each processor has a dedicated function.
- ▶ **Incremental growth** - If we need 10 percent more computing power, we just add 10 percent more processors.



- ▶ **Reliability and availability** - A few parts of the system can be down without disturbing people using the other parts.

Distributed System Goals

- ▶ **Openness:** An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.
 - It should be easy to configure the system out of different components.



- ▶ **Making Resources Accessible** - Main goal of a distributed system
 - ▶ make it easy for the users (and applications) to access remote resources
 - ▶ to share them in a controlled and efficient way.
 - Resources - anything: printers, computers, storage facilities, data, files, Web pages, and networks, etc.

Advantages of Distributed Systems over Centralized Systems

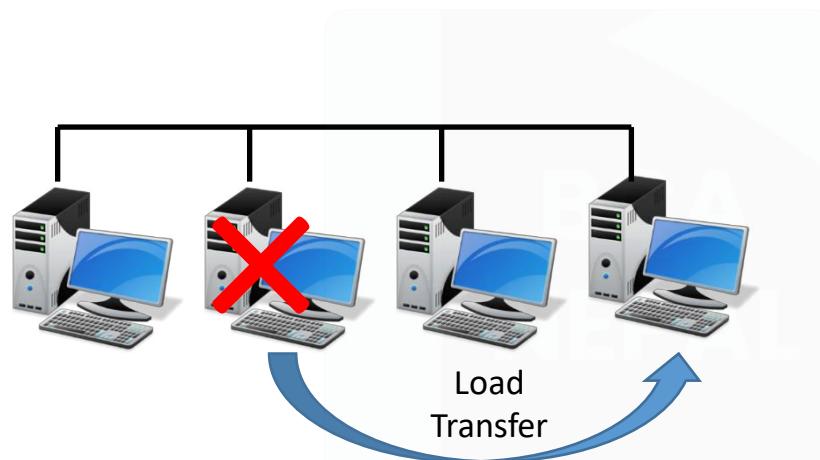
- ▶ **Economics:** A collection of microprocessors offer a better price/performance than mainframes. It is an cost effective way to increase computing power.



- ▶ **Speed:** A distributed system may have more total computing power than a mainframe.
- ▶ **Inherent distribution:** Some applications are inherently distributed. Ex. a supermarket chain, Banking, Airline reservation.

Advantages of Distributed Systems over Centralized Systems

- ▶ **Reliability:** If one machine crashes, the system as a whole can still survive. Higher availability and improved reliability.
 - Ex. control of nuclear reactors or aircraft.



- ▶ **Data sharing:** Allow many users to access to a common database.

Advantages of Distributed Systems over Centralized Systems

- ▶ **Resource Sharing:** Expensive peripherals such as color laser printers, photo-type setters and massive archival storage devices are also among the few things that should be sharable.



- ▶ **Communication:** Enhance human-to-human communication, e.g., email, chat.
- ▶ **Flexibility:** Spread the workload over the available machines

Disadvantages of Distributed Systems over Centralized System

► **Software:**

- ➔ Would be complex.

► **Network problem:**

- ➔ Network saturation.
- ➔ Malfunctioning of network.

► **Security:**

- ➔ Possibility of security violation since the private data are visible to others over the network.

Issues in Designing a Distributed System

Transparency

Reliability

Flexibility

Performance

Scalability

Heterogeneity

Security

Transparency

- ▶ Main goal of Distributed system is to make the existence of multiple computers invisible (transparent) and provide single system image to user.
- ▶ A transparency is some aspect of the distributed system that is hidden from the user (programmer, system developer, application).
- ▶ While users hit search in google.com, They never notice that their query goes through a complex process before google shows them a result.

Types of Transparency

Access Transparency

- Local and remote objects should be accessed in a uniform way.
- User should not find any difference in accessing local or remote objects.
- Hide differences in data representation & resource access (enables interoperability).
- Example : Navigation in the Web

Location Transparency

- Objects are referred by logical names which hide the physical location of the objects.
- Resource should be independent of the physical connectivity or topology of the system or the current location of the resources.
- Hide location of resource (can use resource without knowing its location).
- Example: Pages in the Web

Replication Transparency

- The provision of create replicas (additional copies) of files and other resources on different node of the distributed system.
- Hide the possibility that multiple copies of the resource exist (for reliability and/or availability).
- Replica of the files and data are transparent to the user.

Types of Transparency

Failure Transparency

- It deals with the **masking from the users partial failures** in the system, such as a communication link failure, a machine failure, or a storage device crash.
- Hide failure and recovery of the resource.
- Example: Database Management System.

Migration Transparency

- Resource object is to be **moved from one place to another automatically** by the system.
- Hide possibility that a system may change location of resource (no effect on access).
- **Load balancing** is one among many reason for migration of objects.

Concurrency Transparency

- Each user has the feeling that **he or she is the sole user of the system** and other user do not exists in the system.
- Hide the possibility that the resource may be shared concurrently.
- Example: Automatic teller machine network, DBMS

Types of Transparency

Performance Transparency

- It allows the system to be automatically reconfigured to improve performance, as load varies dynamically in the system.
- As far as practicable, a situation in which one processor of the system is overloaded with jobs while another processor is idle should not be allowed to occur.

Scaling Transparency

- It allows the system to expand in scale without disrupting the activities of the users.
- Example: World-Wide-Web

Reliability

- ▶ Distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources.
- ▶ System failure are of two types:
 - ➔ **Fail-stop:** The system stop functioning after detecting the failure.
 - ➔ **Byzantine failure:** The system continues to function but gives wrong results.
- ▶ The fault-handling mechanism must be designed properly to **avoid faults**, to **tolerate faults** and to **detect and recover from faults**.

Reliability

► Fault avoidance

- Fault avoidance deals with designing the components of the system in such a way that the occurrence of faults is minimized

► Fault tolerance:

- Redundancy technique: To avoid single point of failure.
- Distributed control: To avoid simultaneous functioning of the servers.

► Fault detection and recovery

- Atomic transaction.
- Stateless server.
- Acknowledgment and timeout-based retransmissions of messages.

Flexibility

- ▶ The design of Distributed operating system should be flexible due to following reasons:
- ▶ **Ease of Modification:** It should be easy to incorporate changes in the system in a user transparent manner or with minimum interruption caused to the users.
- ▶ **Ease of Enhancement:** New functionality should be added from time to time to make it more powerful and easy to use.
- ▶ A group of users should be able to add or change the services as per the comfortability of their use.

Performance

- ▶ A **performance** should be **better than or at least equal to** that of running the same application on a **single-processor system**.
- ▶ Some design principles considered useful for better performance are as below:
 - **Batch if possible:** Batching often helps in improving performance.
 - **Cache whenever possible:** Caching of data at clients side frequently improves over all system performance.
 - **Minimize copying of data:** Data copying overhead involves a substantial CPU cost of many operations.
 - **Minimize network traffic:** It can be improved by reducing internode communication costs.

Scalability

- ▶ Distributed systems must be **scalable** as the number of user increases.

A system is said to be scalable if it can handle the **addition of users** and **resources** without suffering a noticeable loss of performance or increase in administrative complexity.

- ▶ Scalability has 3 dimensions:

- ▶ **Size:** Number of users and resources to be processed. Problem associated is overloading.
- ▶ **Geography:** Distance between users and resources. Problem associated is communication reliability.
- ▶ **Administration:** As the size of distributed systems increases, many of the system needs to be controlled. Problem associated is administrative mess.

- ▶ Guiding principles for designing scalable distributed systems:

- ▶ Avoid centralized entities.
- ▶ Avoid centralized algorithms.
- ▶ Perform most operations on client workstations.

Heterogeneity

- ▶ This term means the **diversity of the distributed systems** in terms of hardware, software, platform, etc.
- ▶ Modern distributed systems will likely span different:
 - ▶ **Hardware devices:** computers, tablets, mobile phones, embedded devices, etc.
 - ▶ **Operating System:** Ms Windows, Linux, Mac, Unix, etc.
 - ▶ **Network:** Local network, the Internet, wireless network, satellite links, etc.
 - ▶ **Programming languages:** Java, C/C++, Python, PHP, etc.
 - ▶ Different roles of software developers, designers, system managers.

Security

- ▶ System must be protected against **destruction** and **unauthorized access**.
- ▶ Enforcement of security in a distributed system has the following additional requirements as compared to centralized system:
 - ➔ Sender of the message should know that message was received by the intended receiver.
 - ➔ Receiver of the message should know that the message was sent by genuine sender.
 - ➔ Both sender and receiver should be guaranteed that the content of message were not changed while it is in transfer.

Brief (Issues in Designing a Distributed System)

Transparency

Provide a single system image to its user.

Reliability

Degree of Fault tolerance should be low.

Flexibility

Ease of Modification and Enhancement.

Performance

Performance should be better than Centralized system.

Scalability

Capability of a system to adopt increased service load.

Heterogeneity

It consist of dissimilar hardware or software systems.

Security

Must be protected against destruction and unauthorized access.

Classification of Distributed System



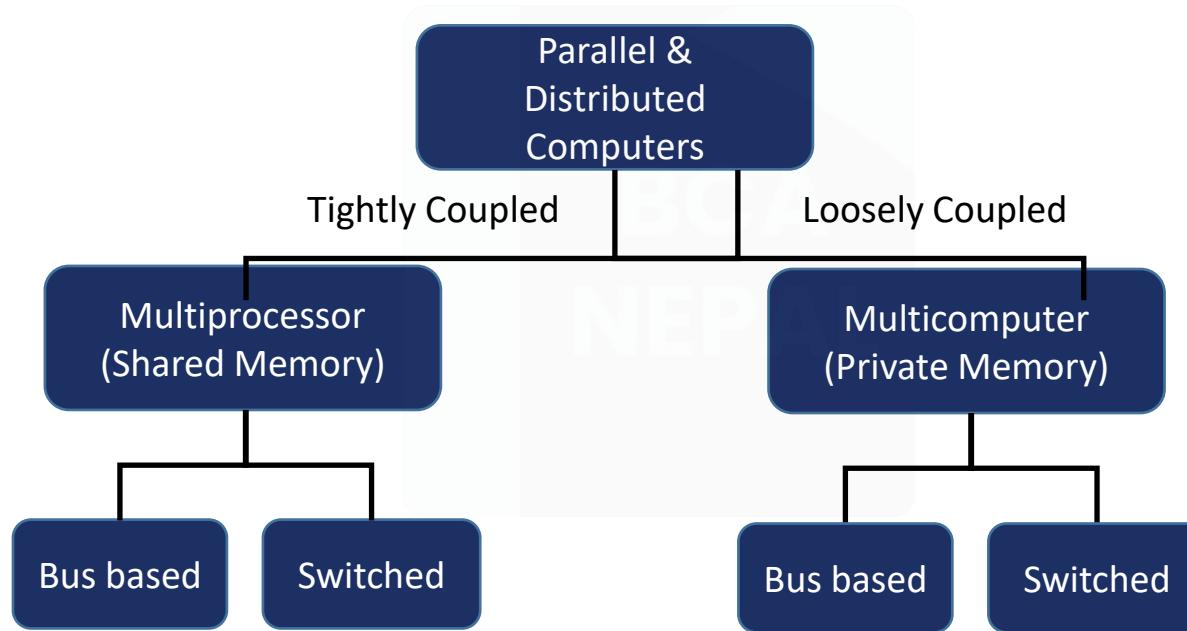
Based on Hardware



Based on number
of instructions and
DataStream

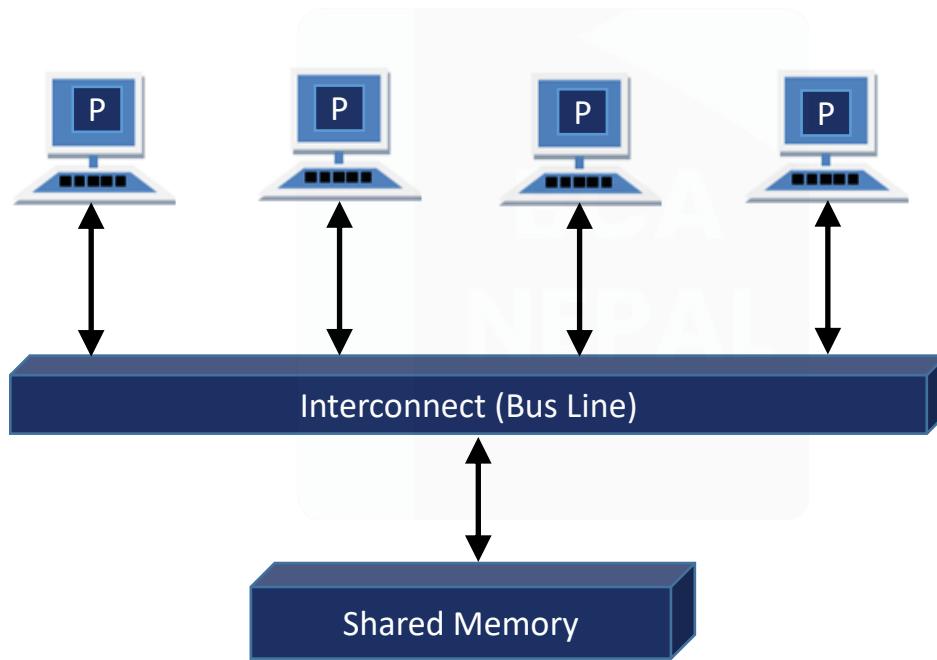
Classification based on Hardware

- Even though all distributed system consist of multiple CPUs, there are several different ways the hardware can be organized, specially in terms of how they are interconnected and communicate.



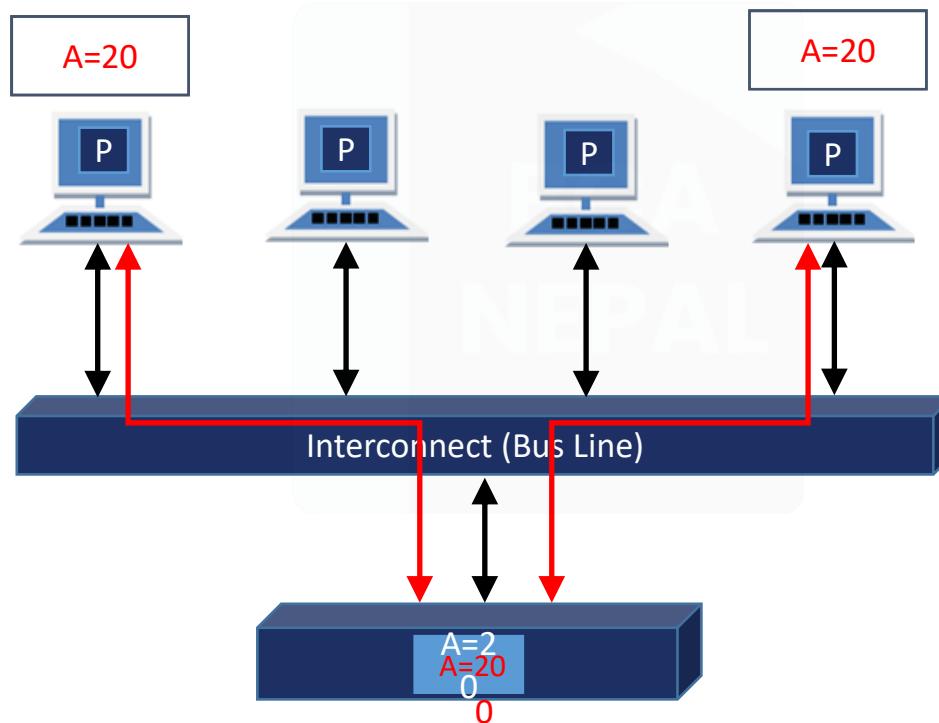
Tightly-Coupled OS(Shared Memory)

- ▶ **Shared Memory Machine:** The n processors shares physical address space. Communication can be done through **shared memory**.



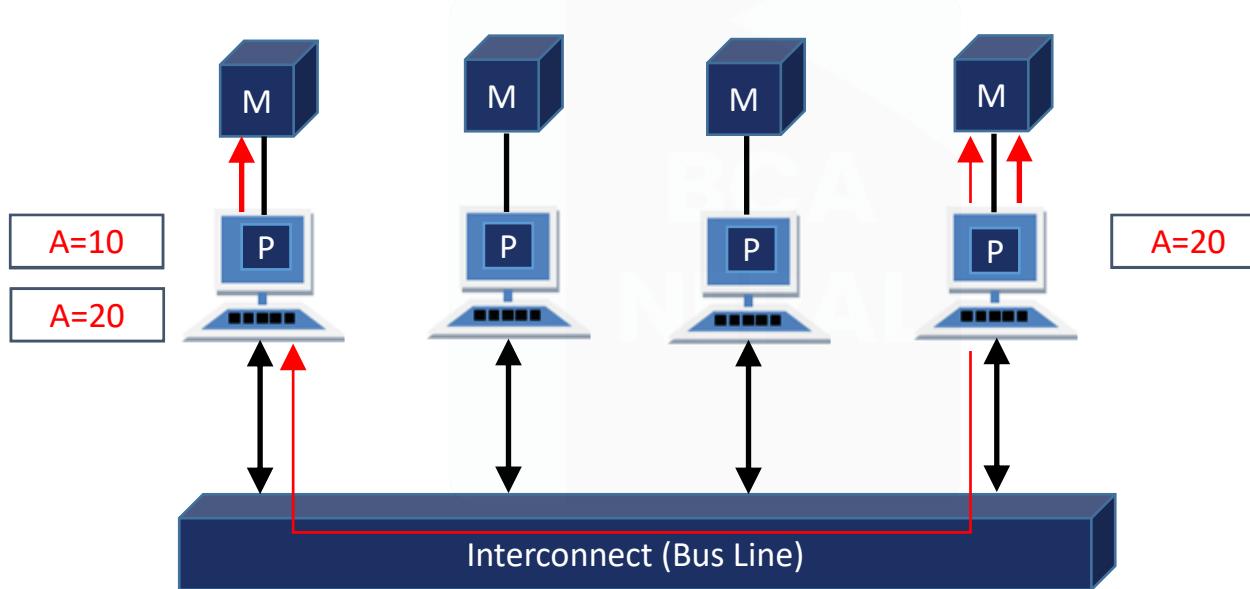
Tightly-Coupled OS(Shared Memory)

- ▶ **Shared Memory Machine:** The n processors shares physical address space. Communication can be done through **shared memory**.



Loosely-Coupled OS(Private Memory)

- ▶ **Private Memory Machine:** Each processor has its own local memory. **Communication** can be done through **Message passing**.

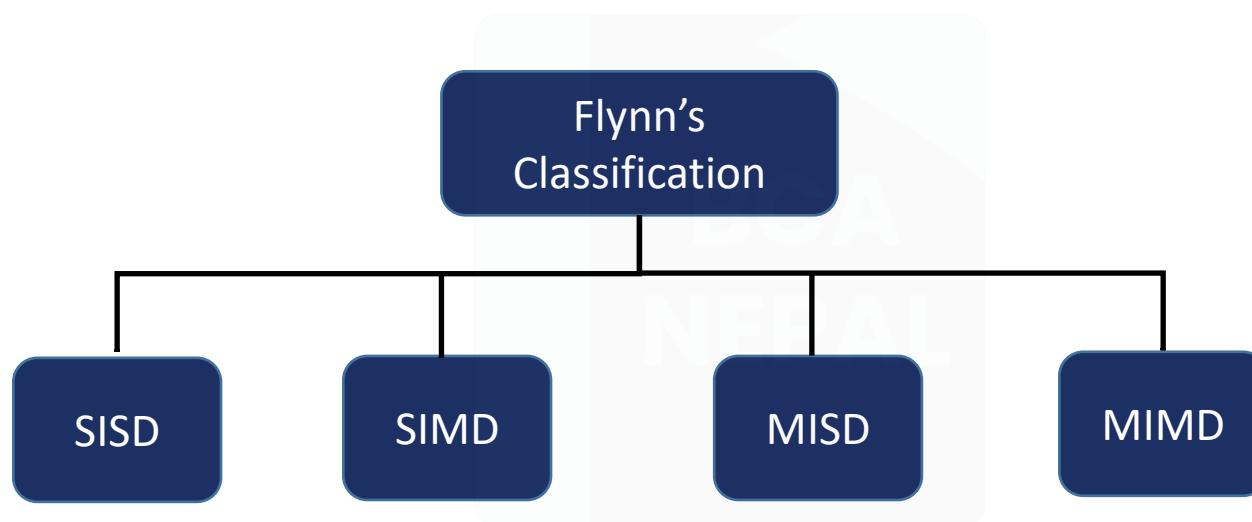


Classification based on Hardware

Loosely-Coupled OS	Tightly-Coupled OS
Each processor has its own local memory.	The n processors shares physical address space.
Communication can be done through Message passing.	Communication can be done through shared memory.
Manages heterogeneous multicompiler Distributed Systems.	Manages multiprocessors & homogeneous multicompiler.
Similar to “local access feel” as a non-distributed, standalone OS.	Provides local services to remote clients via remote logging
Data migration or computation migration modes (entire process or threads)	Data transfer from remote OS to local OS via FTP (File Transfer Protocols)
Distributed Operating System (DOS)	Network Operating System (NOS)

Classification based on Instruction & DataStream

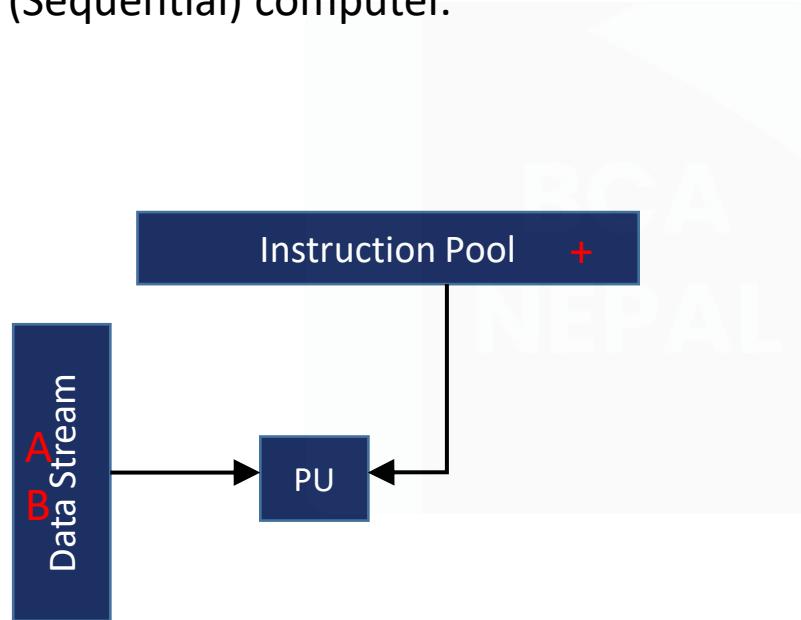
- ▶ According to Flynn's classification can be done based on the number of instruction streams and number of data streams.



Classification based on Instruction & DataStream

► Single instruction stream single data stream (SISD)

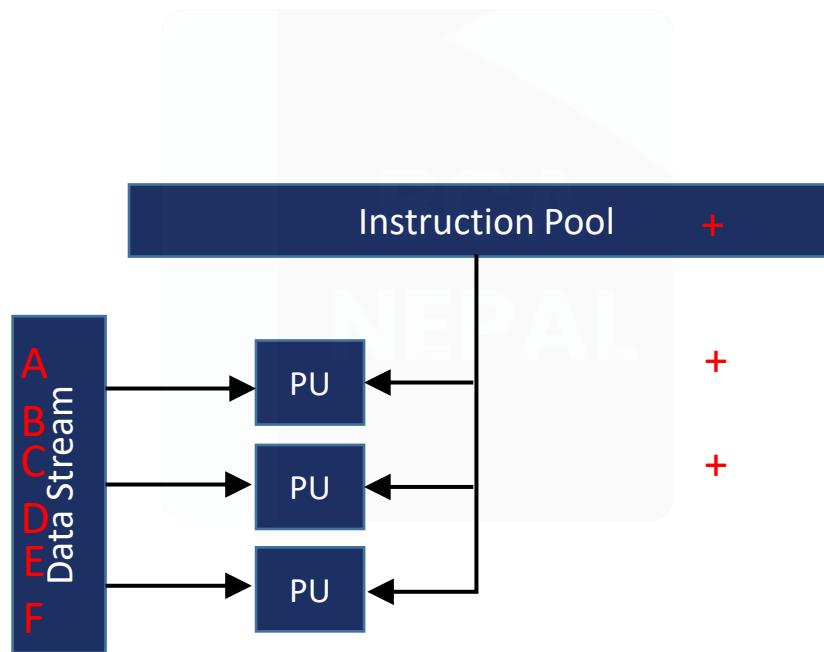
- One Program counter and one path to data memory.
- A computer is capable of executing one instruction at a time operating on one piece of data.
- An ordinary (Sequential) computer.



Classification based on Instruction & DataStream

▶ Single instruction stream, multiple data streams (SIMD)

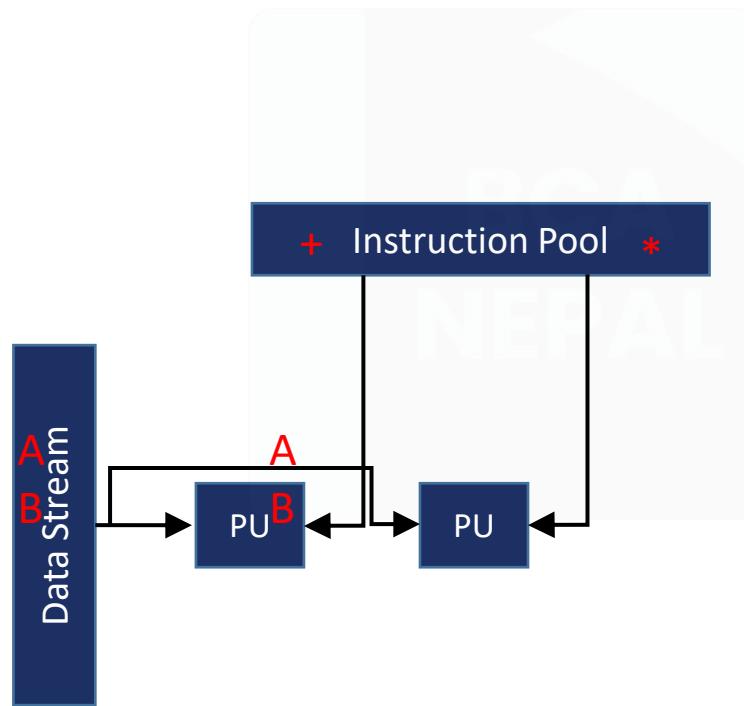
- One Program counter and multiple paths to data memory.
- A computer is capable of executing one instruction at a time, but operating on different pieces of data.



Classification based on Instruction & DataStream

► Multiple instruction streams, single data stream (MISD)

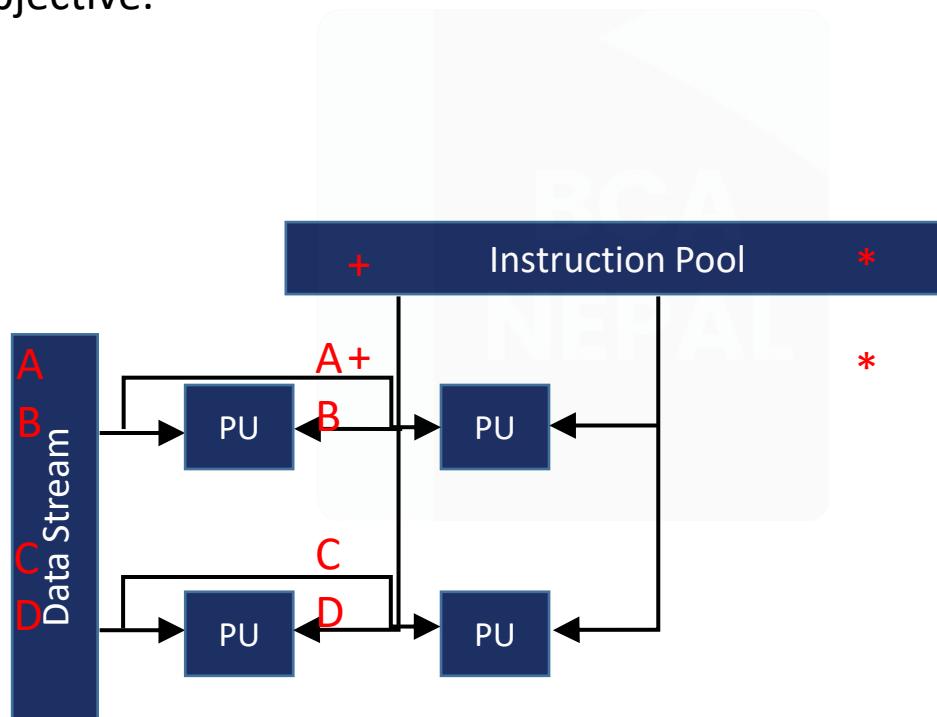
- No more computers fit this model.
- Uncommon architecture which is generally used for fault tolerance.



Classification based on Instruction & DataStream

► Multiple instruction streams, Multiple data stream (MIMD)

- A group of independent computers, each with its own program counter, program, and data.
- A computer that can run multiple processes or threads that are cooperating towards a common objective.

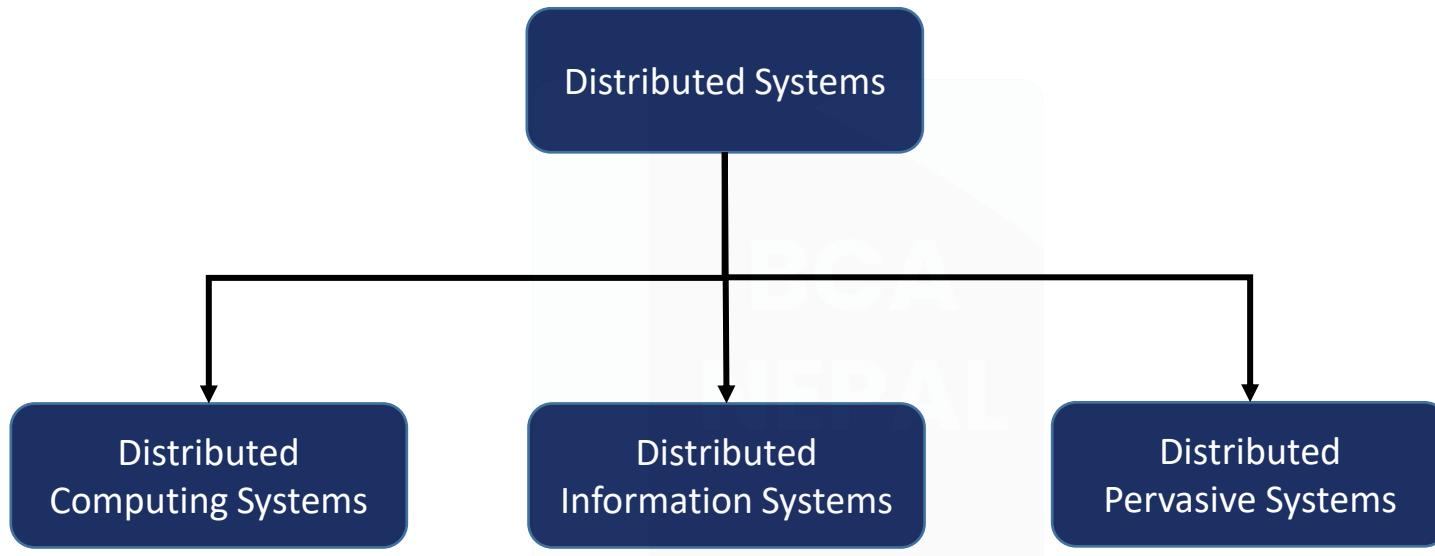


Classification based on Instruction & DataStream

- ▶ All distributed systems are MIMD, We divide all MIMD computers into two groups:
 - Have shared memory, usually called multiprocessors.
 - Do not have shared memory, called multicompiler.



Types of Distributed Systems



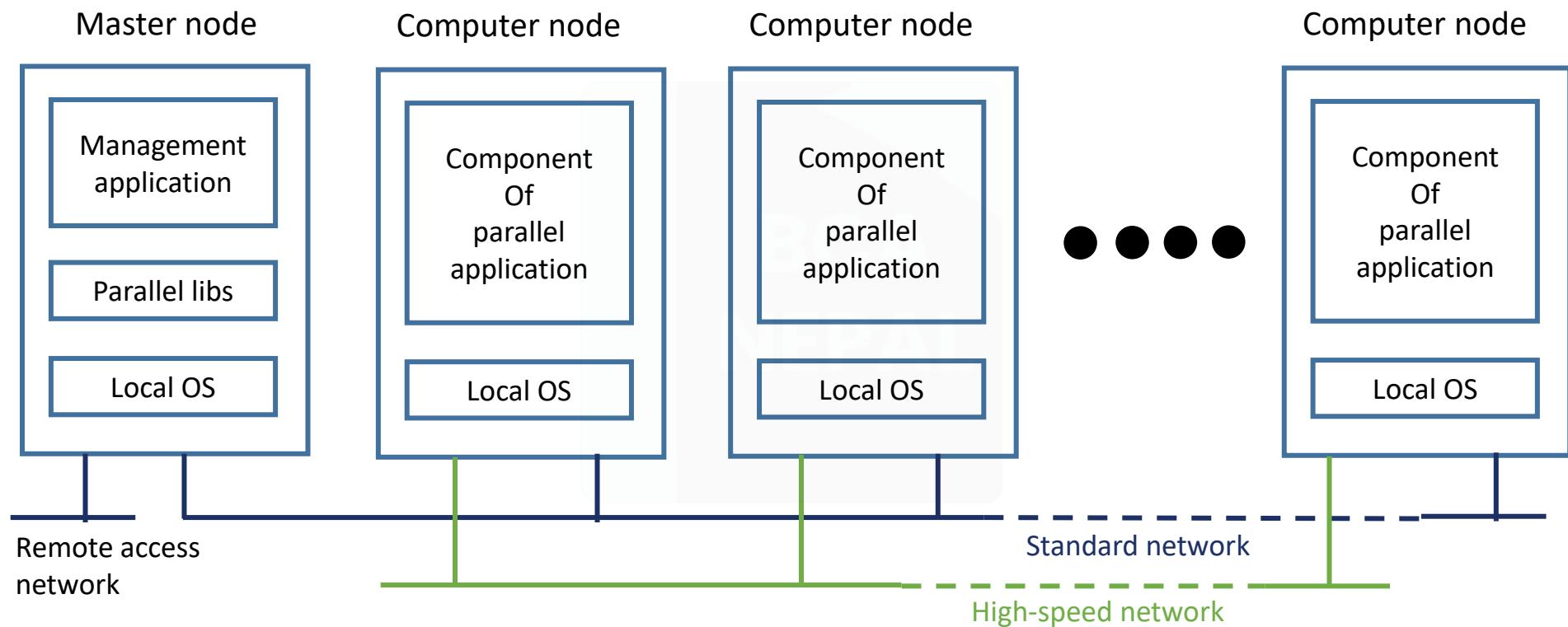
Distributed Computing Systems

- ▶ An important class of distributed systems is the one used for higher performance computing tasks.
- ▶ Here, Computers in a network communicate via message passing.
- ▶ Two Types of Distributed Computing Systems:
 - ▶ Cluster Computing Systems
 - ▶ Grid Computing Systems

Cluster Computing Systems

- ▶ A collection of similar workstations or PCs connected by a high- speed local-area network (LAN)
- ▶ It is homogenous given that each node runs the same OS.
- ▶ The ever increasing price / performance ration of computers makes it cheaper to build a supercomputer by putting together many simple computers, rather than buying a high-performance one.
- ▶ Also, robustness is higher, maintenance and incremental addition of computing power is easier
- ▶ Usage
 - ➔ Parallel programming
 - ➔ Typically, a single computationally-intensive program is run in parallel on multiple machines

An Example of Cluster Computing Systems



Grid Computing Systems

- ▶ A characteristic feature of cluster computing is its homogeneity.
- ▶ In most cases, the computers in a cluster are largely the same, they all have the same operating system, and are all connected through the same network.
- ▶ In contrast, grid computing systems have a **high degree of heterogeneity**: no assumptions are made concerning hardware, operating systems, networks, administrative domains, security policies, etc.
- ▶ A key issue in a grid computing system is that resources from different organizations are brought together to allow the collaboration of a group of people or institutions.



Distributed Information Systems

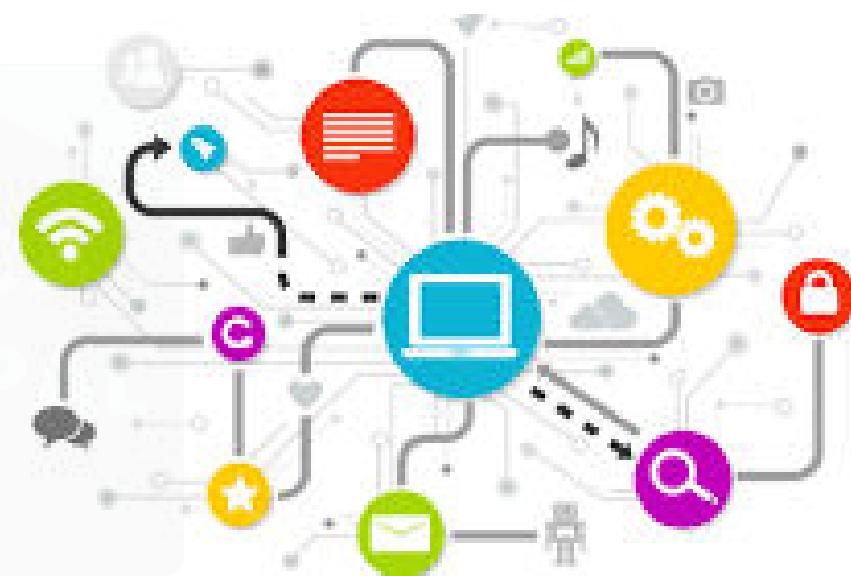
- ▶ Evolved in organizations that were confronted with a wealth of **networked applications**, but for which **interoperability** turned out to be problematic.
- ▶ Many of the existing middleware solutions are the result of working with an infrastructure in which it was easier to integrate applications into an **enterprise-wide information system**.
- ▶ Several levels at which integration took place:
 - **Several non-interoperating servers shared by a number of clients**: distributed queries, distributed transactions. Example : Transaction Processing Systems
 - **Several sophisticated applications** – not only databases, but also processing components – requiring to directly communicate with each other. Example. Enterprise Application Integration (EAI)

Transaction Processing Systems

- ▶ A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID):
 - **Atomicity**: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.
 - **Consistency**: A transaction establishes a valid state transition.
 - **Isolation**: Concurrent transactions do not interfere with each other. It appears to each transaction T that other transactions occur either before T, or after T, but never both.
 - **Durability**: After the execution of a transaction, its effects are made permanent

Enterprise Application Integration

- ▶ The more applications became decoupled from the databases they were built upon, the more evident it became that facilities were needed to integrate applications independent from their databases.
- ▶ Application components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems.
- ▶ Result: **Middleware as a communication facilitator in enterprise application integration**

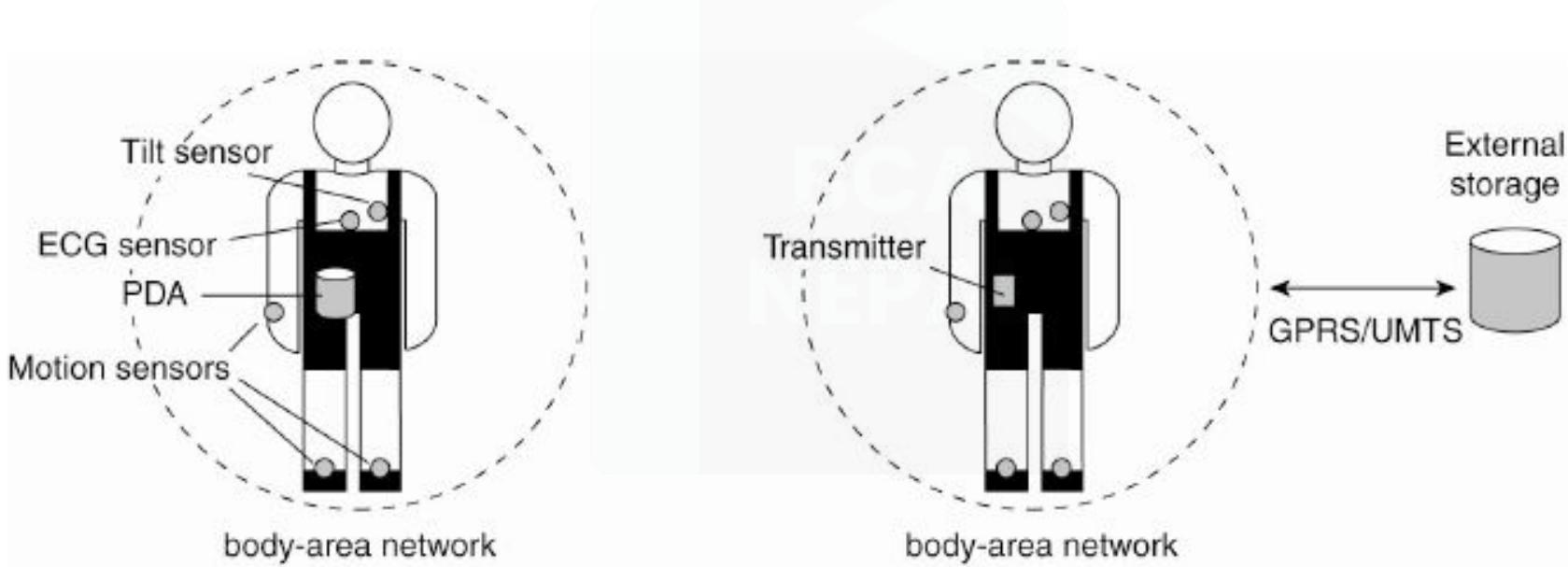


Distributed Pervasive Systems

- ▶ Above distributed systems characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network.
- ▶ Mobile and embedded computing devices: instability is the default behavior.
- ▶ We are now confronted with distributed systems in which instability is the default behavior. The devices in these, what we refer to as distributed pervasive systems.
- ▶ They are often characterized by being small, battery-powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices.

Health Care Systems

- ▶ Personal systems built around a Body Area Network
- ▶ Possibly, minimizing impact on the person like, preventing free motion



Features and Requirements of Distributed Pervasive Systems

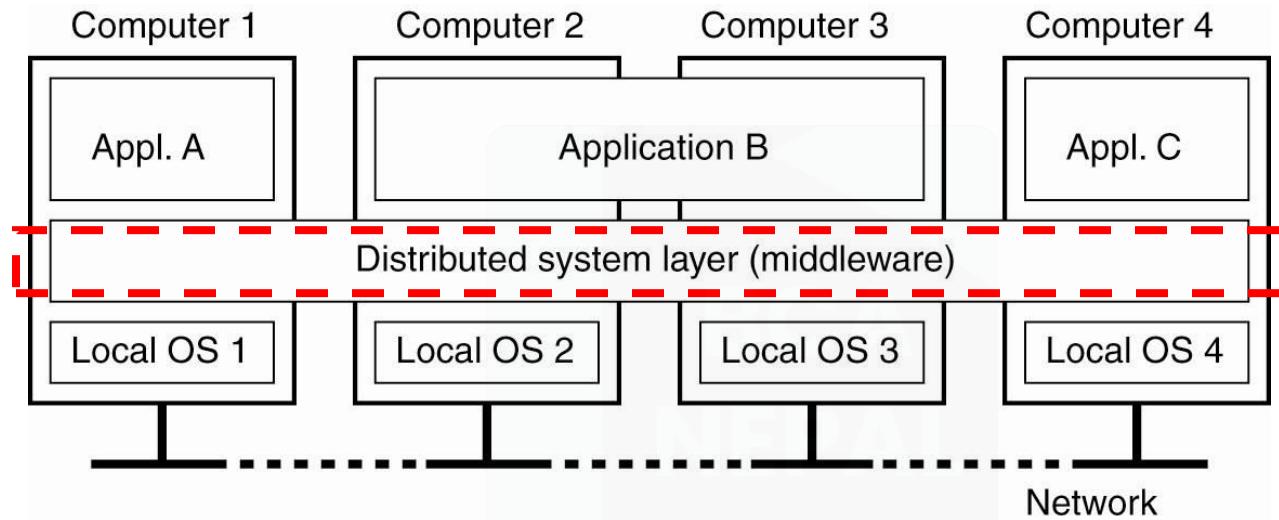
► Features:

- General lack of human administrative control ·
- Devices can be configured by their owners
- They need to automatically discover their environment and fit in as best as possible

► Requirements for pervasive applications

- Embrace contextual changes
- Encourage ad hoc composition
- Recognize sharing as the default

Distributed Operating System Architecture



- ▶ A distributed system organized as **Middleware**.
- ▶ The middleware layer runs on all machines, and offers a **uniform interface to the system**.
- ▶ Middleware is software which lies between an operating system and the applications running on it.

Middleware (MW)

- ▶ Software that manages and supports the different components of a distributed system. In essence, it sits in the middle of the system.
- ▶ Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications.
- ▶ It enables multiple systems to communicate with each other across different platforms.
- ▶ Examples:
 - ▶ Transaction processing monitors
 - ▶ Data converters
 - ▶ Communication controllers

Role of Middleware (MW)

► In some early systems:

- ➔ Middleware tried to provide the illusion that a collection of separate machines was a single computer.

► Today:

- ➔ Clustering software allows independent computers to work together closely.
- ➔ Middleware also supports seamless access to remote services, doesn't try to look like a general-purpose OS.

■ Other Middleware Examples

- ➔ CORBA (Common Object Request Broker Architecture)
- ➔ DCOM (Distributed Component Object Management) – being replaced by .NET
- ➔ Sun's ONC RPC (Remote Procedure Call)
- ➔ RMI (Remote Method Invocation)
- ➔ SOAP (Simple Object Access Protocol)

CASE STUDY: The World Wide Web

- ▶ WWW stands for World Wide Web. A technical definition of the World Wide Web is : all the resources and users on the Internet that are using the Hypertext Transfer Protocol (HTTP).
- ▶ A broader definition comes from the organization that Web inventor Tim Berners-Lee helped found, the World Wide Web Consortium (W3C).
- ▶ The World Wide Web is the universe of network-accessible information, an embodiment of human knowledge.
- ▶ In simple terms, The World Wide Web is a way of exchanging information between computers on the Internet, tying them together into a vast collection of interactive multimedia resources.
- ▶ Note: Internet and Web is not the same thing: Web uses internet to pass over the information.



CASE STUDY: The World Wide Web

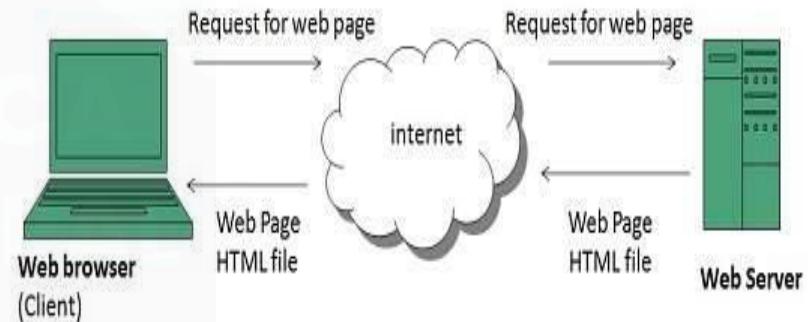
- ▶ The World Wide Web [www.w3.org I, Berners-Lee 1991] is an evolving system for publishing and accessing resources and services across the Internet. Through commonly available web browsers, users retrieve and view documents of many types, listen to audio streams and view video streams, and interact with an unlimited set of services.
- ▶ The Web began life at the European center for nuclear research (CERN), Switzerland, in 1989 as a vehicle for exchanging documents between a community of physicists connected by the Internet [Berners-Lee 1999]. A key feature of the Web is that it provides a hypertext structure among the documents that it stores, reflecting the users' requirement to organize their knowledge. This means that documents contain links (or hyperlinks) – references to other documents and resources that are also stored in the Web.
- ▶ The Web is an open system: it can be extended and implemented in new ways without disturbing its existing functionality.
 - ➔ First, its operation is based on communication standards and document or content standards that are freely published and widely implemented.
 - ➔ Second, the Web is open with respect to the types of resource that can be published and shared on it.

WWW Operation

► **WWW** works on client-server approach. Following steps explains how the web works:

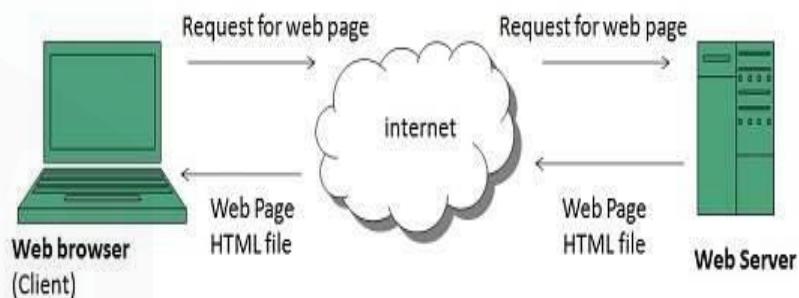
► 1. User enters the URL (say, **http://www.ambition.com**) of the web page in the address bar of web browser.

► 2. Then browser requests the Domain Name Server for the IP address corresponding to **www.ambition.com**.

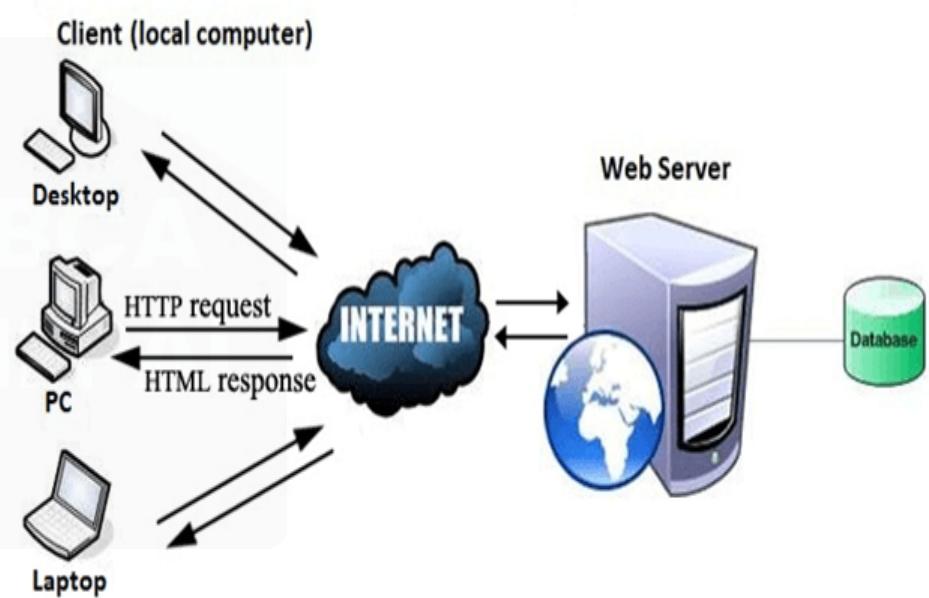
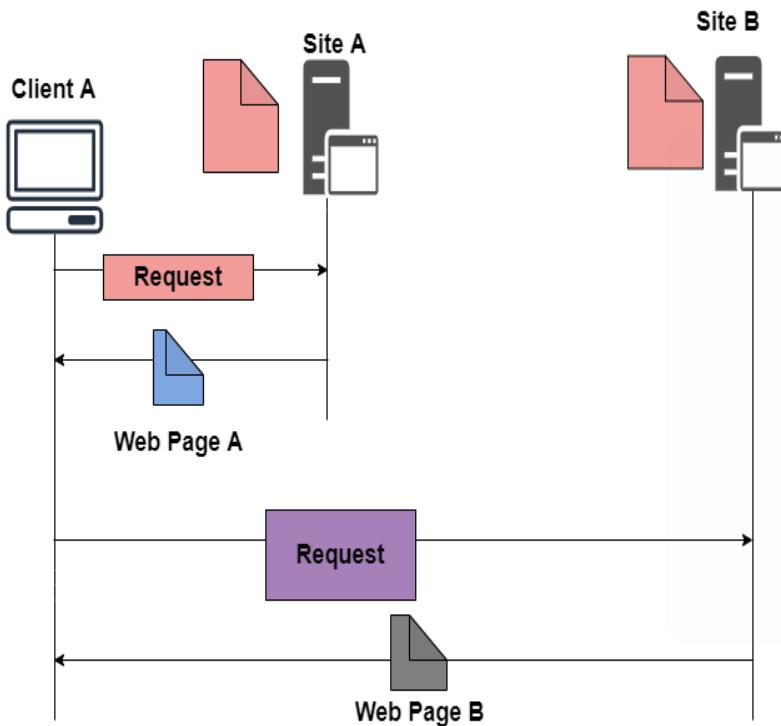


WWW Operation

- ▶ 3. After receiving IP address, browser sends the request for web page to the web server using HTTP protocol which specifies the way the browser and web server communicates.
- ▶ 4. Then web server receives request using HTTP protocol and checks its search for the requested web page. If found it returns it back to the web browser and close the HTTP connection.
- ▶ 5. Now the web browser receives the web page, It interprets it and display the contents of web page in web browser's window.



WWW Operation



Components of WWW

- ▶ 1.Client/Browser
- ▶ 2.Server
- ▶ 3. URL
- ▶ 4. HTML
- ▶ 5. XML



1.Client/Browser

- The Client/Web browser is basically a program that is used to communicate with the webserver on the Internet.
- Each browser mainly comprises of three components and these are:
 - Controller
 - Interpreter
 - Client Protocols
- The Controller mainly receives the input from the input device, after that it uses the client programs in order to access the documents.
- After accessing the document, the controller makes use of an interpreter in order to display the document on the screen.
- An interpreter can be Java, HTML, javascript mainly depending upon the type of the document.
- The Client protocol can be FTP, HTTP, TELNET.

2.Server

- The Computer that is mainly available for the network resources and in order to provide services to the other computer upon request is generally known as the **server**.
- The Web pages are mainly stored on the server.
- Whenever the request of the client arrives then the corresponding document is sent to the client.
- The connection between the client and the server is TCP.
- It can become more efficient through multithreading or multiprocessing. Because in this case, the server can answer more than one request at a time.

3.URL

- ▶ URL is an abbreviation of **the Uniform resource locator**.
- ▶ It is basically a standard used for specifying any kind of information on the Internet.
- ▶ In order to access any page the client generally needs an address.
- ▶ To facilitate the access of the documents throughout the world HTTP generally makes use of Locators.
- ▶ URL mainly defines the four things:
 - ▶ **Protocol** It is a client/server program that is mainly used to retrieve the document. A commonly used protocol is HTTP.
 - ▶ **Host Computer** It is the computer on which the information is located. It is not mandatory because it is the name given to any computer that hosts the web page.
 - ▶ **Port** The URL can optionally contain the port number of the server. If the port number is included then it is generally inserted in between the host and path and is generally separated from the host by the colon.
 - ▶ **Path** It indicates the pathname of the file where the information is located.

4.HTML

- ▶ HTML is an abbreviation of Hypertext Markup Language.
 - ➔ It is generally used for creating web pages.
 - ➔ It is mainly used to define the contents, structure, and organization of the web page.



5.XML

- ▶ XML is an abbreviation of Extensible Markup Language. It mainly helps in order to define the common syntax in the semantic web.



Features of WWW

► Given below are some of the features provided by the World Wide Web:

- ➔ Provides a system for Hypertext information
- ➔ Open standards and Open source
- ➔ Distributed.
- ➔ Mainly makes the use of Web Browser in order to provide a single interface for many services.
- ➔ Dynamic
- ➔ Interactive
- ➔ Cross-Platform

Advantages of WWW

- ▶ Given below are the benefits offered by WWW:
 - It mainly provides all the information for Free.
 - Provides rapid Interactive way of Communication.
 - It is accessible from anywhere.
 - It has become the Global source of media.
 - It mainly facilitates the exchange of a huge volume of data.

Disadvantages of WWW

- ▶ There are some drawbacks of the WWW and these are as follows;
 - ➔ It is difficult to prioritize and filter some information.
 - ➔ There is no guarantee of finding what one person is looking for.
 - ➔ There occurs some danger in case of overload of Information.
 - ➔ There is no quality control over the available data.
 - ➔ There is no regulation.

CASE STUDY: The World Wide Web

- ▶ WWW is the huge collection of pages of information linked to each other around one globe.
- ▶ Web page is the collection of text, images, video, audio, animation and hyperlink.

- ▶ There are two types of web:
 - ➔ 1. Static web: Information site. Eg. Personal website
 - ➔ 2. Dynamic web: Interactive site. Eg. Commercial website, Government website

CASE STUDY: The World Wide Web

► Publishing a resource:

- While the Web has a clearly defined model for accessing a resource from its URL, the exact methods for publishing resources on the Web are dependent upon the web server implementation.
- In terms of low-level mechanisms, the simplest method of publishing a resource on the Web is to place the corresponding file in a directory that the web server can access.

CASE STUDY: The World Wide Web

- **Web application:** Web applications are run into browser through URL.
- Types:
 - ▶ **1. Service Oriented:** it is used to implement web services. It is coded into CGI, JSP, ASP, etc.
 - **JSP stands for Java Server Pages**, which helps developers to create dynamically web pages based on HTML, XML, or other types.
 - **ASP stands for Active Server Pages**, which is used in web development to implement dynamic web pages.
 - **CGI stands for Common Gateway Interface**, It is a technology that enables a web browser to submit forms and connect to programs over a webserver. It is the best way for a webserver to send forms and connect to programs on the server.

CASE STUDY: The World Wide Web

► 2. Presentation oriented:

- ➔ It provides client side services. They are coded using HTML, XML, JavaScript, etc.
- ➔ Web architecture:
 - WWW follows the 2 tier architecture.
 - It is combination of webserver and web client.
 - Webserver produce and deliver the information.
 - Web client retrieve and display information.

Distributed System



Unit:2

Unit 2. Architecture

- 2.1 Architectural Styles
- 2.2 Middleware organization
- 2.3 System Architecture
- 2.4 Example Architectures

4 Hrs.

Distribution System Architecture

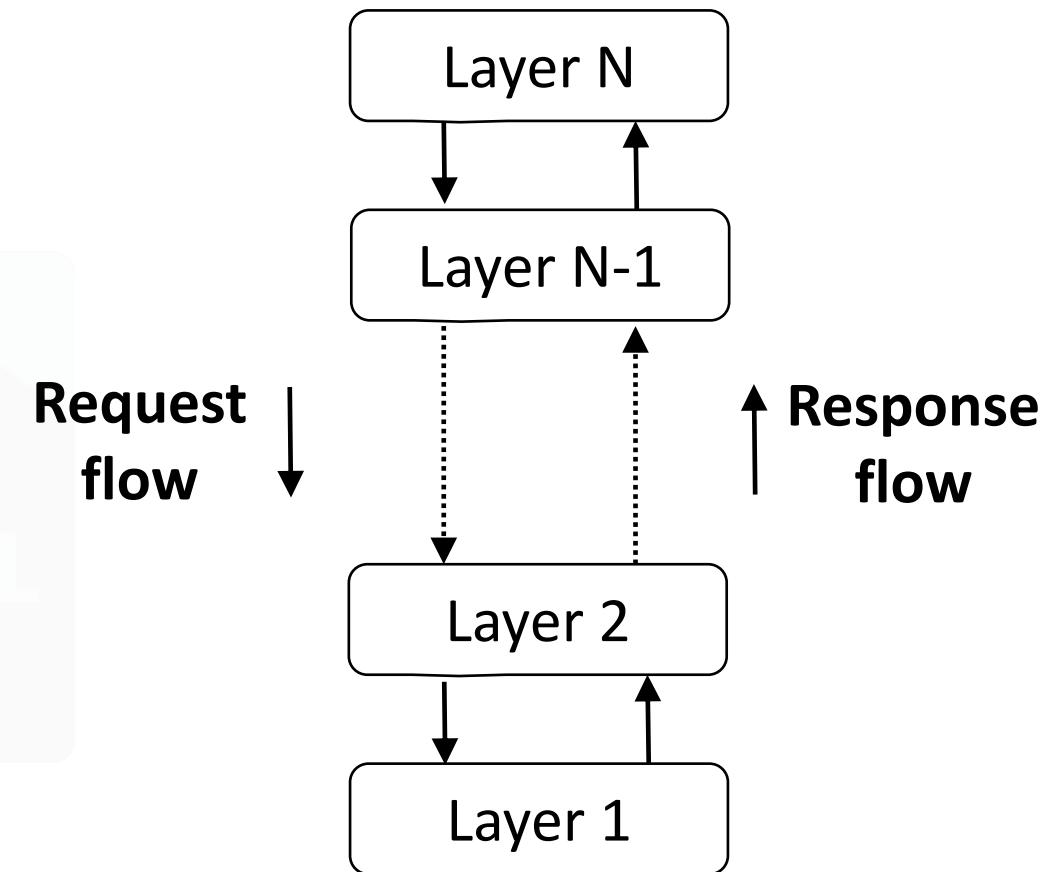
- ▶ Distribution system architectures are bundled up with **components** and **connectors**.
 - ▶ Components can be individual nodes or important components in the architecture whereas connectors are the ones that connect each of these components.
-
- ▶ **Component:** A modular unit with well-defined interfaces; replaceable, reusable.
 - ▶ **Connector:** A communication link between modules which mediates coordination or cooperation among component.

Architectural Styles

- ▶ Distributed systems are often complex pieces of software of which the components are by definition dispersed across multiple machines.
- ▶ To master their complexity, it is crucial that these systems are properly organized.
- ▶ There are different ways on how to view the organization of a distributed system, but an obvious one is to make a distinction between the **logical organization of the collection of software components** and on the other hand the **actual physical realization**.
- ▶ Important styles of architecture for distributed systems:
 - Layered architectures
 - Object-based architectures
 - Data-centered architectures
 - Event-based architectures

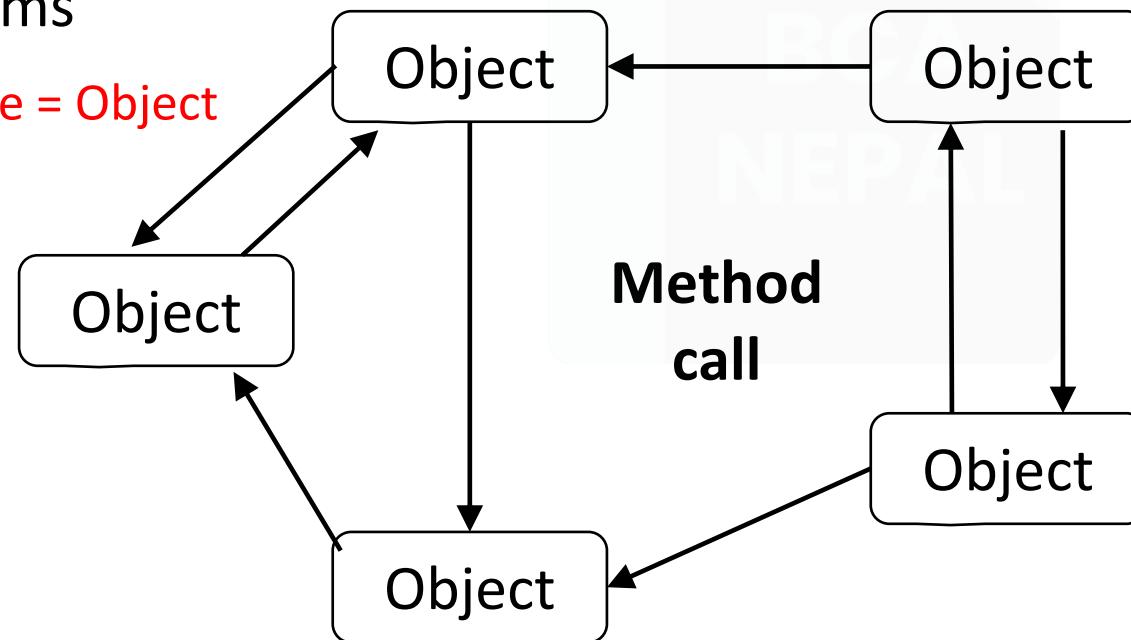
Layered architectures

- ▶ The Components are organized in a layered fashion where a component at layer L_i is allowed to call components at the underlying layer L_{i-1} , but not the other way around,
- ▶ This model has been widely adopted by the networking community
- ▶ A key observation is that control generally flows from layer to layer; requests go down the hierarchy whereas the results flow upward.



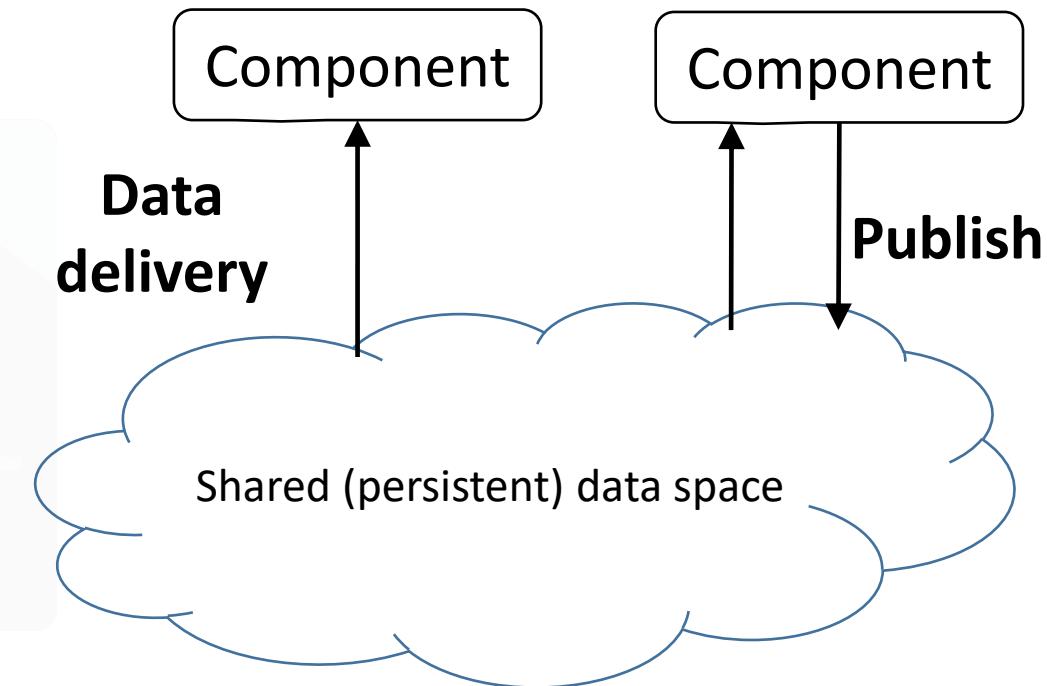
Object-based architectures

- ▶ Loosely coupled (*Own Private Memory*) arrangements of objects.
 - ▶ Each object corresponds a component.
 - ▶ Components are connected through a (remote) procedure call (RPC) mechanism.
 - ▶ The layered and object-based architectures still form the most important styles for large software systems
- ▶ Component, Node = Object
▶ Connector = RPC



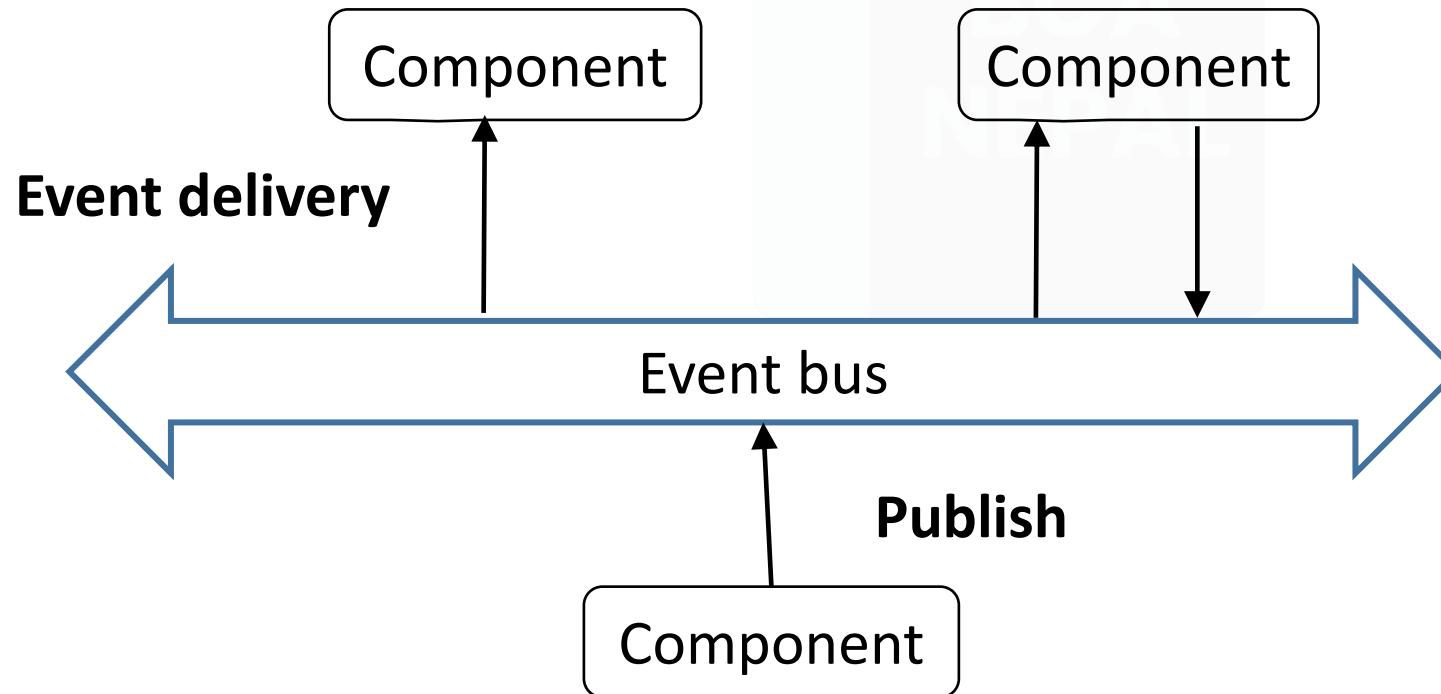
Data-centered architectures

- ▶ It evolve around the idea that processes communicate through a common (passive or active) repository.
- ▶ Service Provider and Service User
- ▶ It can be argued that for distributed systems these architectures are as important as the layered and object-based architectures
- ▶ For example,
 - ▶ A wealth of networked applications have been developed that rely on a shared distributed file system in which virtually all communication takes place through files.
 - ▶ Web-based distributed systems are largely data-centric: processes communicate through the use of shared Web-based data services.

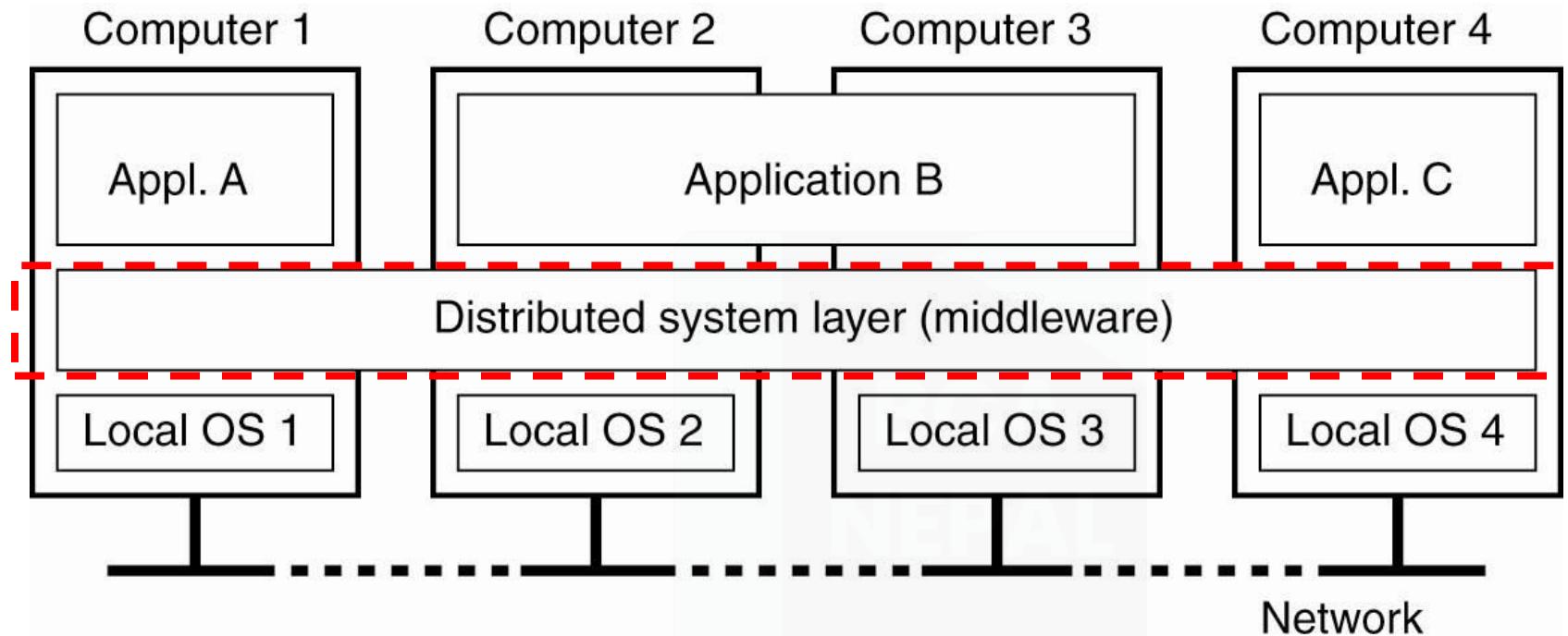


Event-based architectures

- ▶ Processes communicate through the propagation of events.
- ▶ **event producers and event consumers**
- ▶ For distributed systems, event propagation has generally been associated with what are known as publish/subscribe systems.
- ▶ **Advantage** - processes are loosely coupled. In principle, they need not explicitly refer to each other. This is also referred to as being decoupled in space, or referentially decoupled.



Middleware (MW)



- ▶ A distributed system organized as **Middleware**.
- ▶ The middleware layer runs on all machines, and offers a **uniform interface to the system**.
- ▶ Middleware is software which lies between an operating system and the applications running on it.

Middleware (MW)

- ▶ Software that manages and supports the different components of a distributed system. In essence, it sits in the middle of the system.
- ▶ Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications.
- ▶ It enables multiple systems to communicate with each other across different platforms.
- ▶ Examples:
 - ➔ Transaction processing monitors
 - ➔ Data converters
 - ➔ Communication controllers

Role of Middleware (MW)

► In some early systems:

- ➔ Middleware tried to provide the illusion that a collection of separate machines was a single computer.

► Today:

- ➔ Clustering software allows independent computers to work together closely.
- ➔ Middleware also supports seamless access to remote services, doesn't try to look like a general-purpose OS.

■ Other Middleware Examples

- ➔ CORBA (Common Object Request Broker Architecture)
- ➔ DCOM (Distributed Component Object Management) – being replaced by .NET
- ➔ Sun's ONC RPC (Remote Procedure Call)
- ➔ RMI (Remote Method Invocation)
- ➔ SOAP (Simple Object Access Protocol)

System Architecture

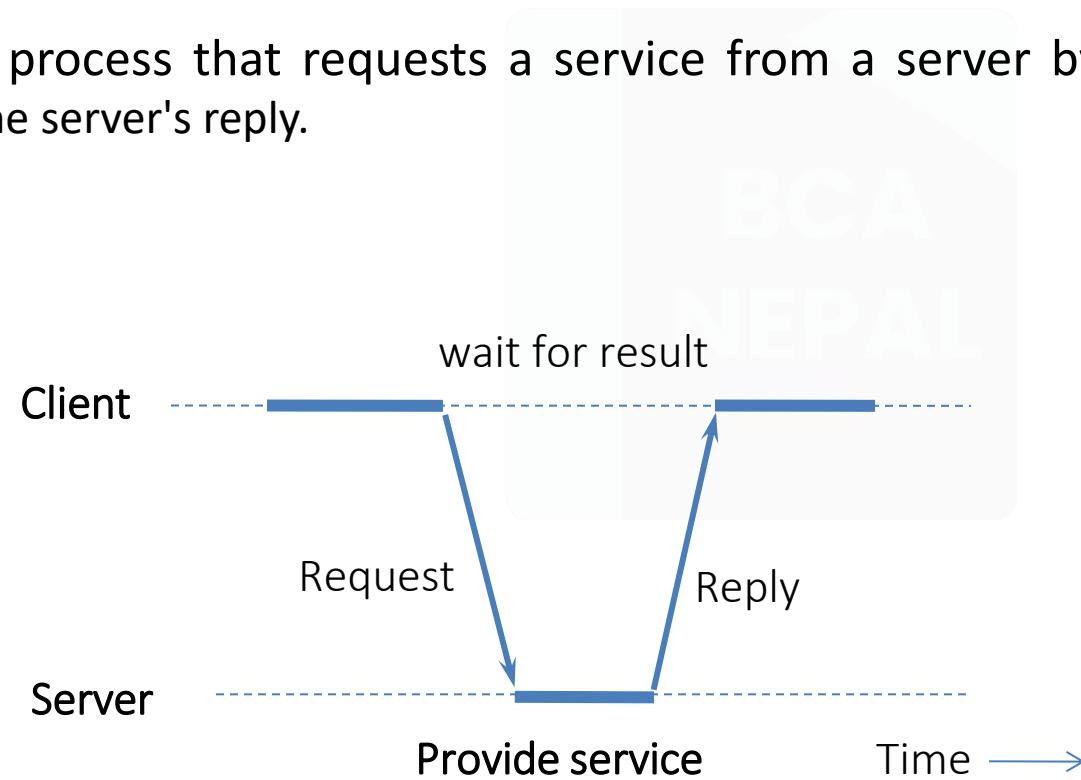
There are three views toward system architectures:

- ▶ Centralized Architectures
- ▶ Decentralized architectures
- ▶ Hybrid Architectures



Centralized Architectures

- ▶ Manage distributed system complexity – think in terms of clients that request services from servers.
- ▶ Processes are divided into two groups:
 1. A server is a process implementing a specific service, for example, a file system service or a database service.
 2. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply.



Centralized Architectures

► **Communication** - implemented using a connectionless protocol when the network is reliable e.g. local-area networks.

1. Client requests a service – packages and sends a message for the server, identifying the service it wants, along with the necessary input data.
2. The Server will always wait for an incoming request, process it, and package the results in a reply message that is then sent to the client.

Connectionless protocol

- ▶ Describes communication between two network end points in which a message can be sent from one end point to another without prior arrangement.
- ▶ Device at one end of the communication transmits data to the other, without first ensuring that the recipient is available and ready to receive the data.
- ▶ The device sending a message sends it addressed to the intended recipient.

Application Layering

Traditional three-layered view:

1. The user-interface level

- It contains units for an application's user interface
- Clients typically implement the user-interface level
 - Simplest user-interface program - character-based screen
 - Simple GUI

2. The processing level

- It contains the functions of an application, i.e. without specific data
- Middle part of hierarchy -> logically placed at the processing level

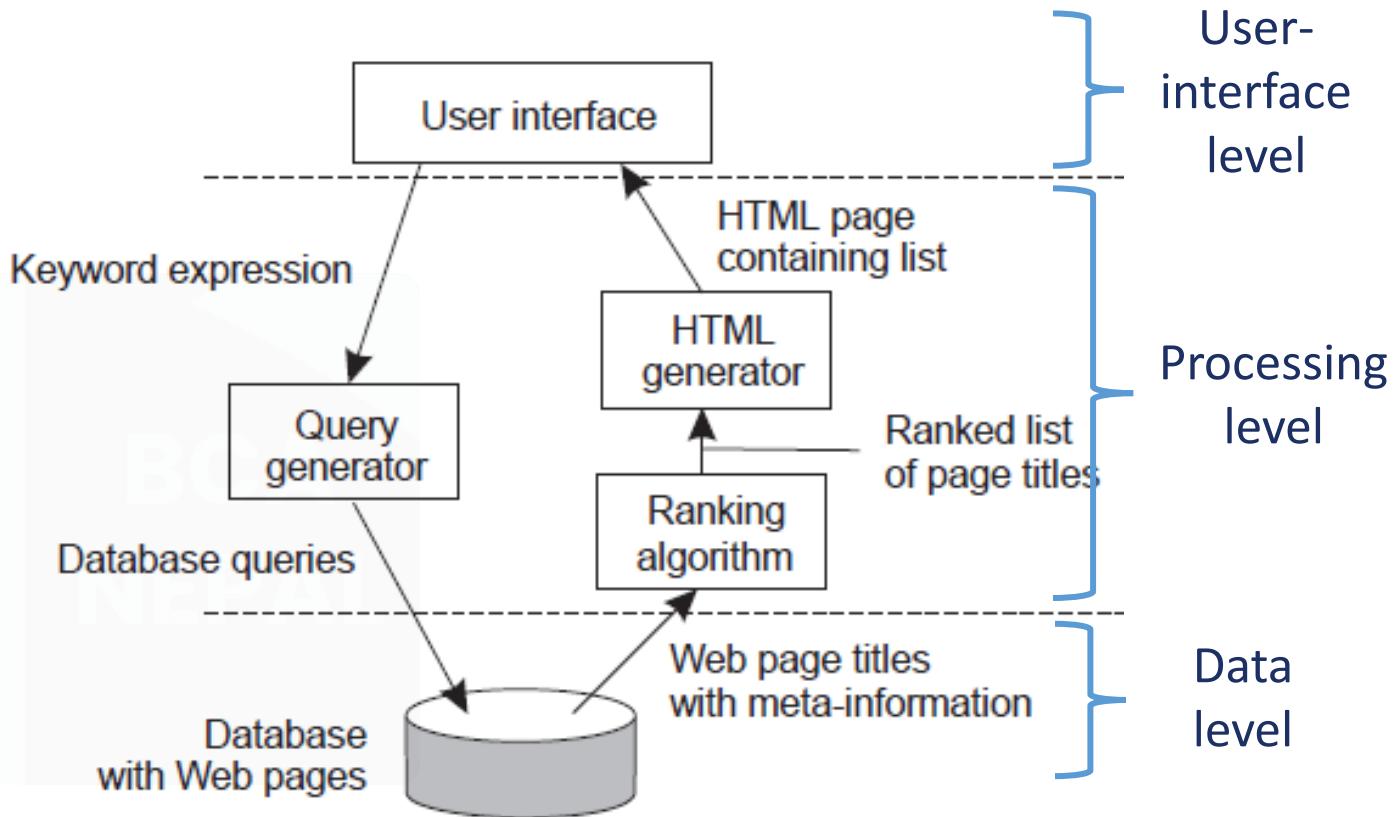
3. The data level

- It contains the data that a client wants to manipulate through the application components
- manages the actual data that is being acted on

Internet search engine- An example of Application Layering

Traditional three-layered view:

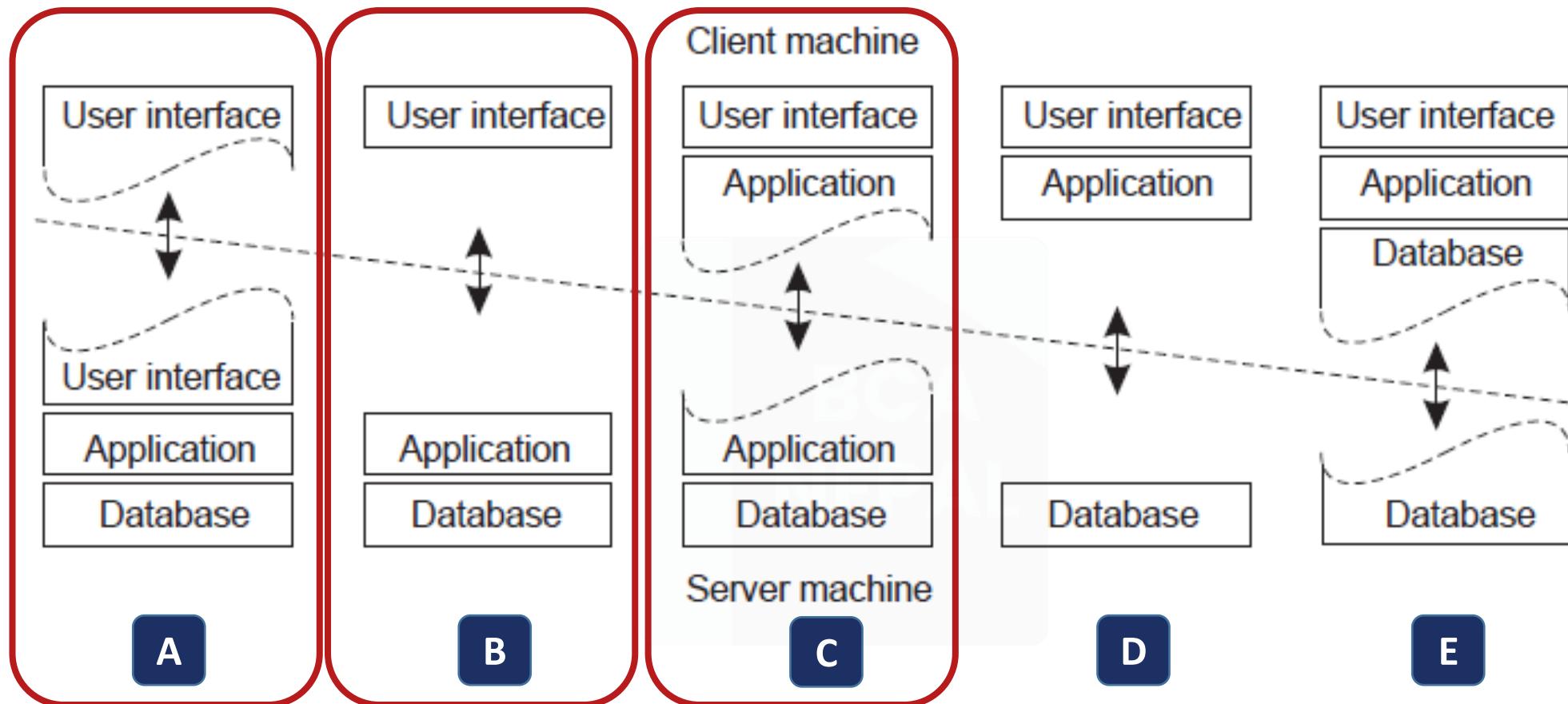
- ▶ **User-interface level:** a user types in a string of keywords and is subsequently presented with a list of titles of Web pages.
- ▶ **Data Level:** huge database of Web pages that have been prefetched and indexed.
- ▶ **Processing level:** search engine that transforms the user's string of keywords into one or more database queries.
 - Ranks the results into a list
 - Transforms that list into a series of HTML pages



Multi-Tiered Architectures

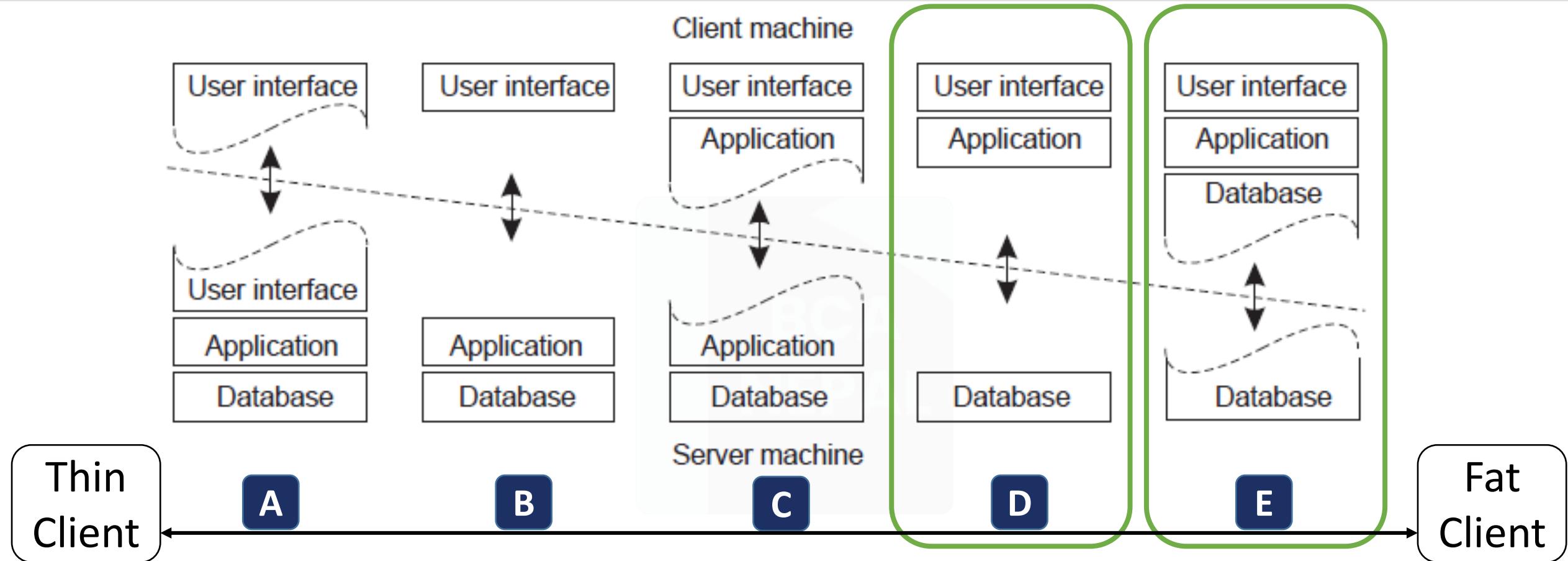
- ▶ Simplest organization - two types of machines:
 1. A client machine containing only the programs implementing (part of) the user-interface level
 2. A server machine containing the rest, that is the programs implementing the processing and data level
- ▶ Single-tiered: dumb terminal/mainframe configuration
- ▶ Two-tiered: client/single server configuration
- ▶ Three-tiered: each layer on separate machine

Two-tiered Architectures - Thin-client model and fat-client model



- ▶ Case-A: Only the terminal-dependent part of the user interface
- ▶ Case- B: Place the entire user-interface software on the client side
- ▶ Case- C: Move part of the application to the front end

Two-tiered Architectures - Thin-client model and fat-client model



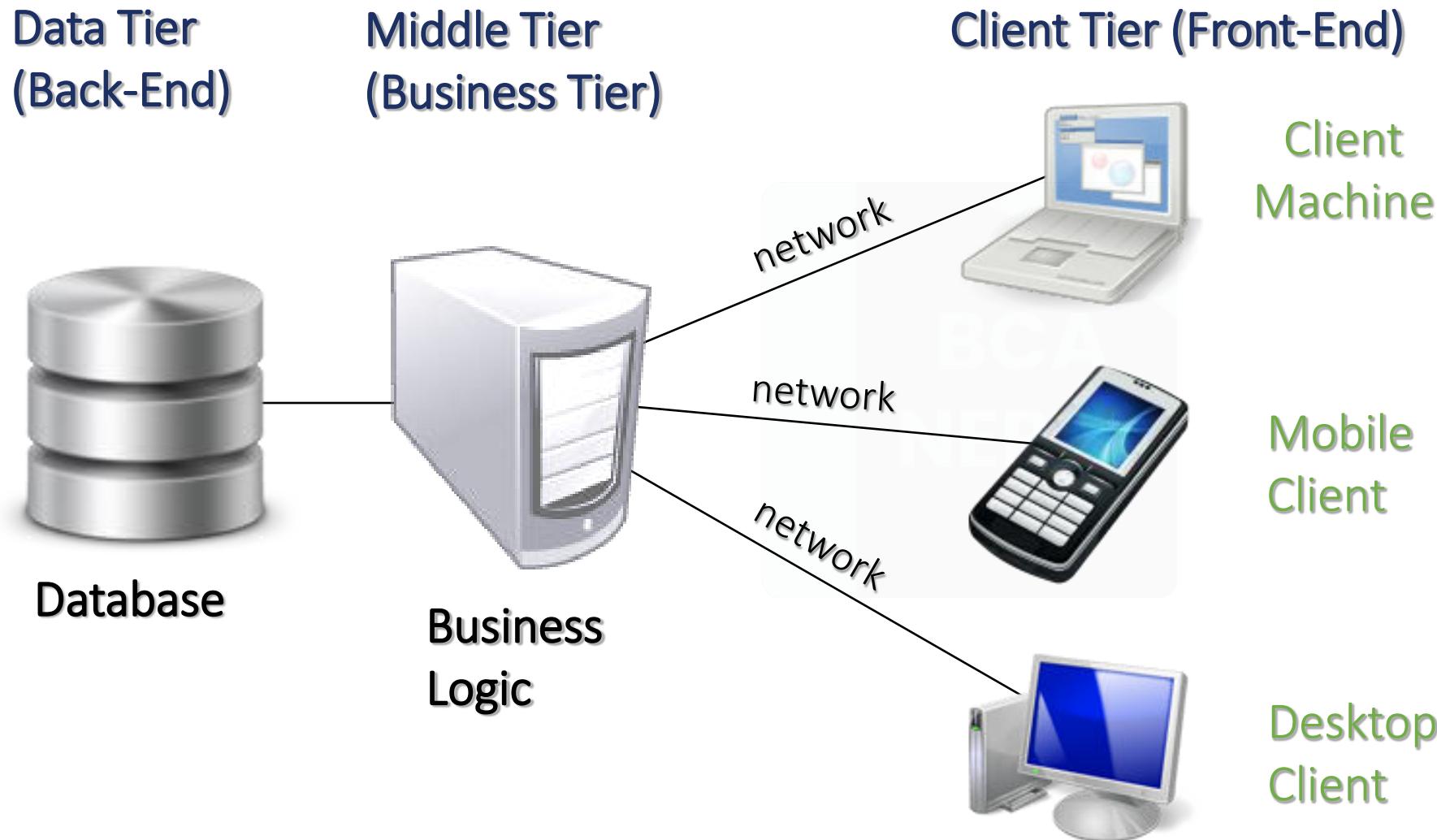
- ▶ Case-D: Used where the client machine is a PC or workstation, connected through a network to a distributed file system or database
- ▶ Case- E: Used where the client machine is a PC or workstation, connected through a network to a distributed file system or database

- ▶ **Case A:** One possible organization is to have only the terminal-dependent part of the user interface on the client machine and give the applications remote control over the presentation of their data.
- ▶ **Case B:** An alternative is to place the entire user-interface software on the client side. In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol. In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.
- ▶ **Case C:** Continuing along this line of reasoning, we may also move part of the application to the front end. An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and where necessary interact with the user.

- ▶ Another example of the organization is that of a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data. but where the advanced support tools such as checking the spelling and grammar execute on the server side.
- ▶ **Case D:** In many client-server environments, the organizations shown in Case D & E are particularly popular. These organizations are used where the client machine is a PC or workstation, connected through a network to a distributed file system or database. Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server. For example, many banking applications run on an end-user's machine where the user prepares transactions and such. Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing.
- ▶ **Case E:** This represents the situation where the client's local disk contains part of the data. For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.

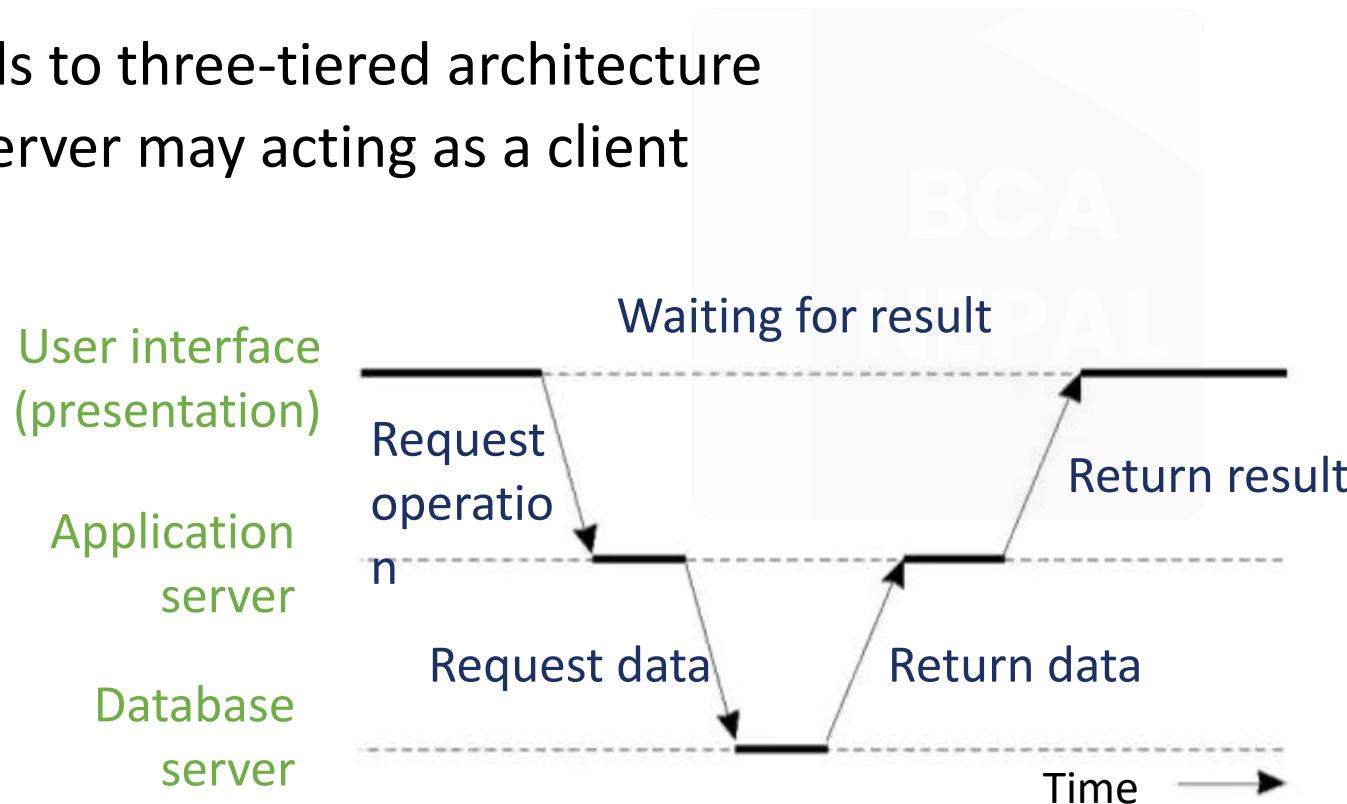
- ▶ We note that for a few years there has been a strong trend to move away from the configurations shown in Case D and Case E in those case that client software is placed at end-user machines. In these cases, most of the processing and data storage is handled at the server side. The reason for this is simple: although client machines do a lot, they are also more problematic to manage. Having more functionality on the client machine makes client-side software more prone to errors and more dependent on the client's underlying platform (i.e., operating system and resources).

Multitiered Architectures (3-Tier Architecture)



Multitiered Architectures (3-Tier Architecture)

- ▶ The server tier in two-tiered architecture becomes more and more distributed
- ▶ Distributed transaction processing
 - A single server is no longer adequate for modern information systems
- ▶ This leads to three-tiered architecture
 - Server may act as a client



An example of
a
server acting as
client

An example of a server acting as client

- ▶ Programs that form part of the processing level reside on a separate server, but may additionally be partly distributed across the client and server machines.
- ▶ Example: Three-tiered architecture - organization of Web sites.
 - Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place.
 - Application server interacts with a database server.



Decentralized Architectures

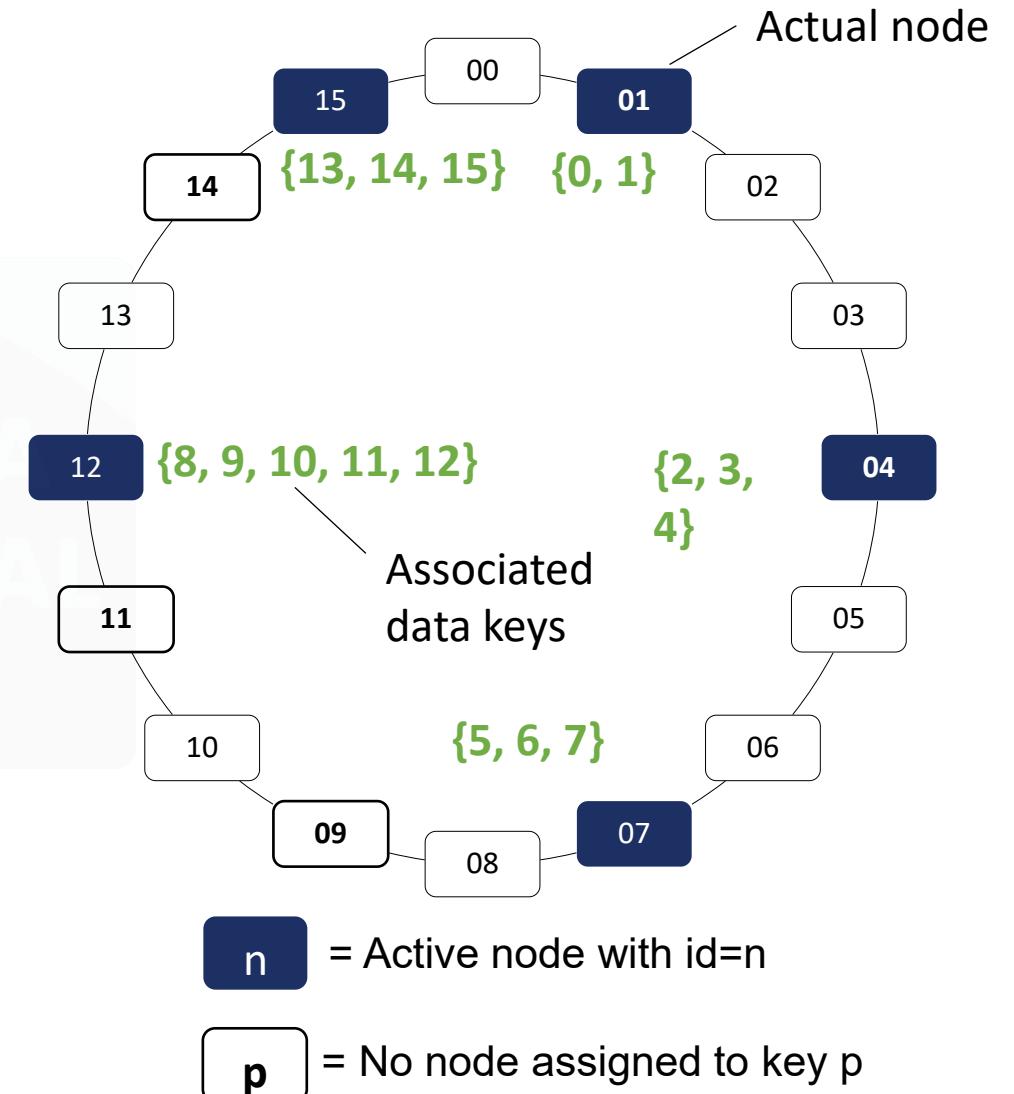
- ▶ Multi-tiered architectures can be considered as vertical distribution
 - Placing logically different components on different machines
- ▶ An alternative is horizontal distribution (peer-to-peer systems)
 - A collection of logically equivalent parts
 - Each part operates on its own share of the complete data set, balancing the load
- ▶ **Peer-to-peer architectures** - how to organize the processes in an overlay network in which the nodes are formed by the processes and the links represent the possible communication channels (which are usually realized as TCP connections).
- ▶ Decentralized Architectures Types
 1. Structured P2P
 2. Unstructured P2P
 3. Hybrid P2P

Structured P2P Architectures

- ▶ There are links between any two nodes that know each other
- ▶ **Structured**: the overlay network is constructed in a deterministic procedure
 - Most popular: distributed hash table (DHT)
- ▶ DHT-based system
 - Data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier.
 - Nodes are assigned a random number from the same identifier space.
- ▶ Some well known structured P2P networks are Chord, Pastry, Tapestry, CAN, and Tulip.

Chord

- ▶ Each node in the network knows the location of some fraction of other nodes.
 - If the desired key is stored at one of these nodes, ask for it directly
 - Otherwise, ask one of the nodes you know to look in *its* set of known nodes.
 - The request will propagate through the overlay network until the desired key is located
 - Lookup time is $O(\log(N))$



Chord

Join

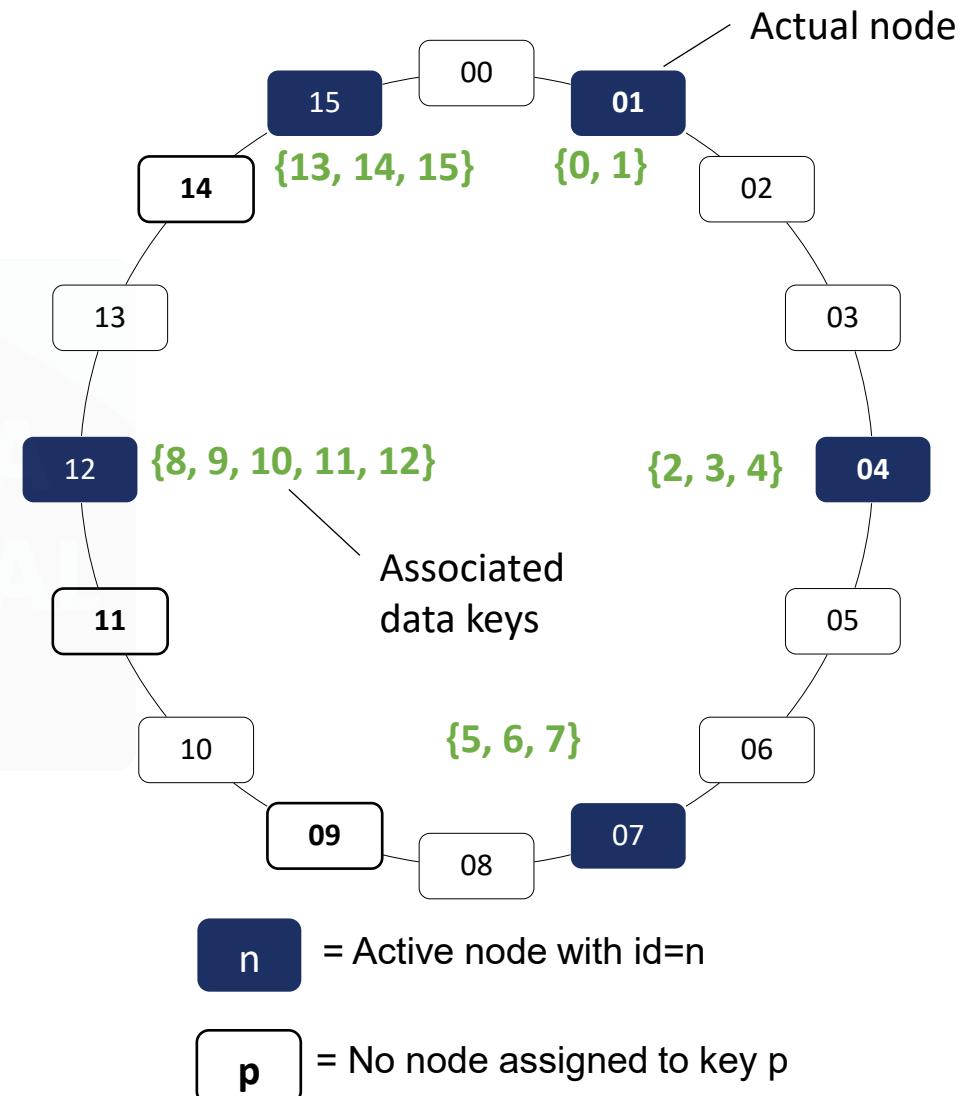
- Generate the node's random identifier, id , using the distributed hash function
- Use the lookup function to locate $succ(id)$
- Contact $succ(id)$ and its predecessor to insert self into ring.
- Assume data items from $succ(id)$

Leave (normally)

- Notify predecessor & successor;
- Shift data to $succ(id)$

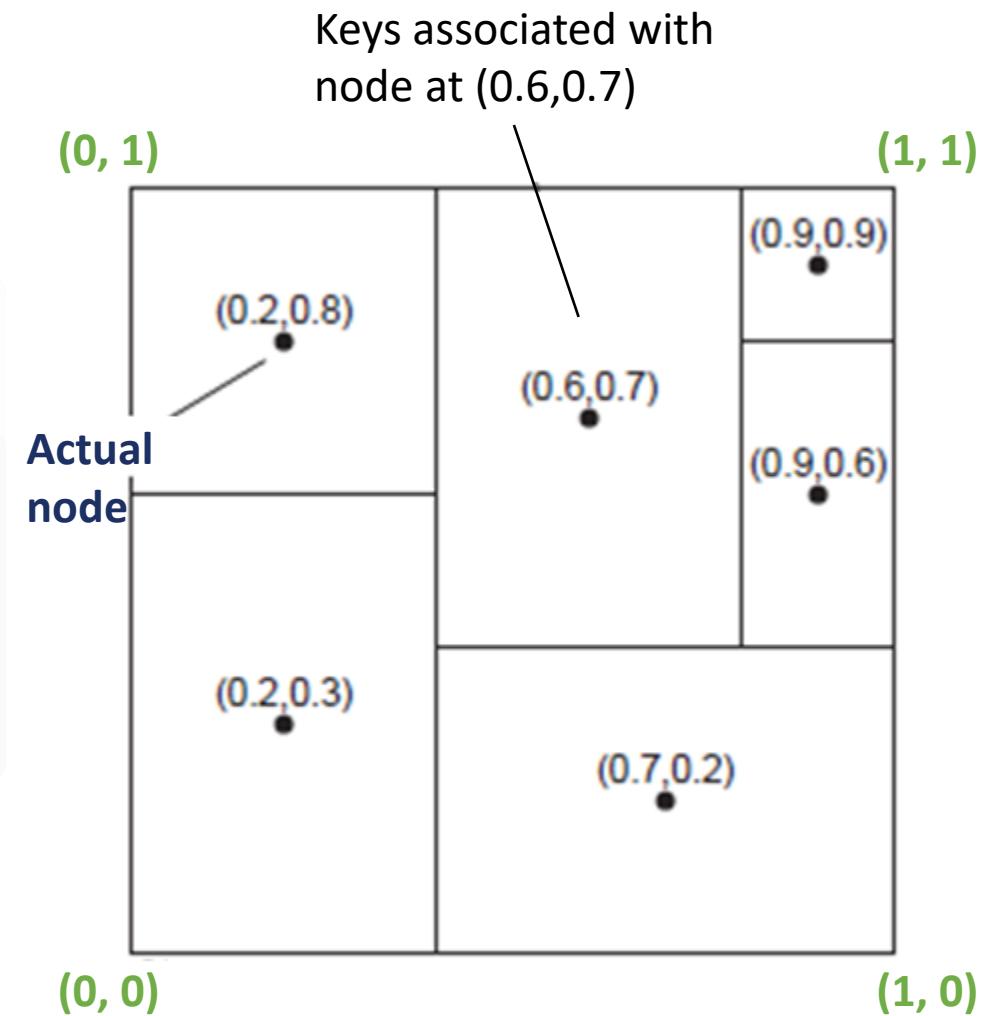
Leave (due to failure)

- Periodically, nodes can run “self-healing” algorithms



Content Addressable Network (CAN)

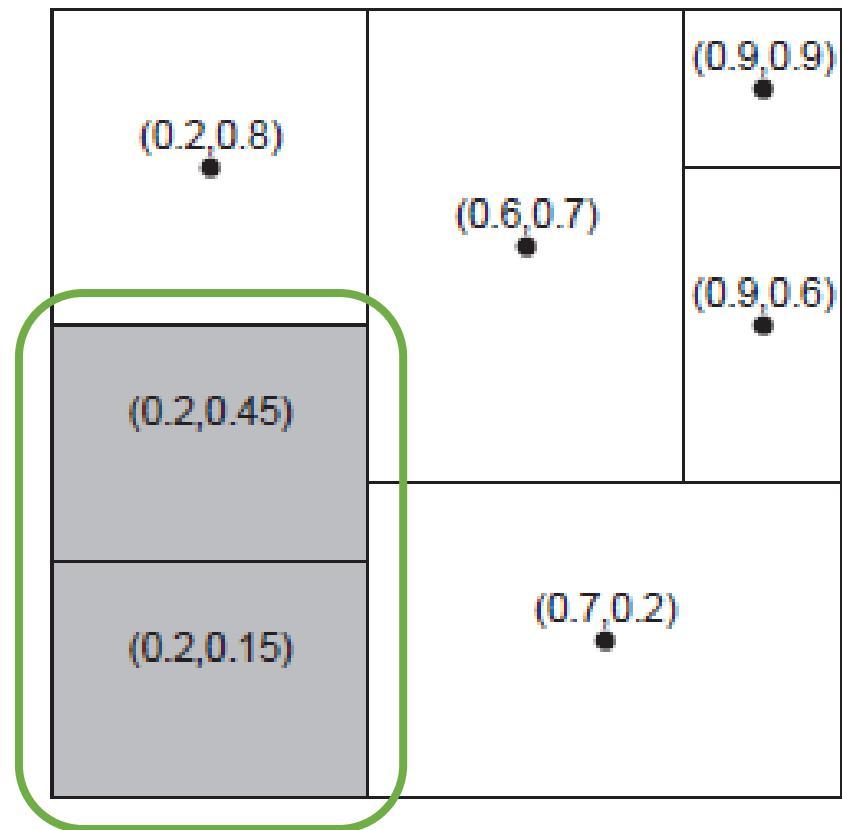
- ▶ CAN deploys a d-dimensional Cartesian coordinate space, which is completely partitioned among all the nodes that participate in the system.
- ▶ Example: **2-dimensional case**
 - Two-dimensional space $[0,1] \times [0,1]$ is divided among six nodes
 - Each node has an associated region
 - Every data item in CAN will be assigned a unique point in this space, after which it is also clear which node is responsible for that data



Content Addressable Network (CAN)

Joining CAN

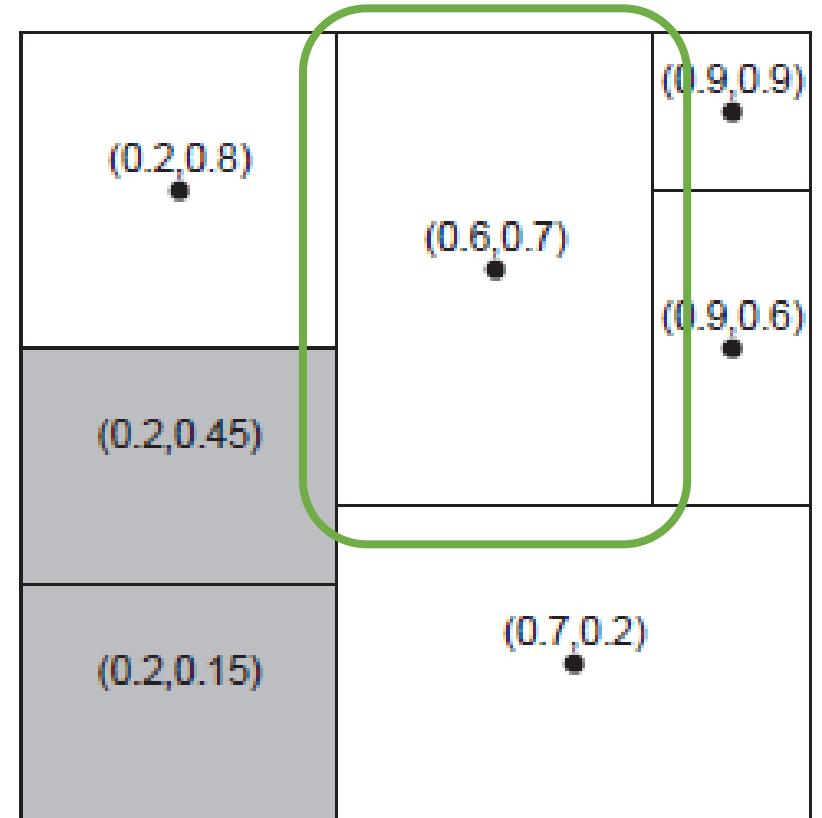
- When a node P wants to join a CAN system, it picks an arbitrary point from the coordinate space and subsequently looks up the node Q in whose region that point falls.
- Node Q then splits its region into two halves and one half is assigned to the node P.
- Nodes keep track of their neighbors, that is, nodes responsible for adjacent region.
- When splitting a region, the joining node P can easily come to know who its new neighbors are by asking node Q.
- As in Chord, the data items for which node P is now responsible are transferred from node Q.



Content Addressable Network (CAN)

Leaving CAN

- ▶ Assume that the node with coordinate $(0.6, 0.7)$ leaves.
- ▶ Its region will be assigned to one of its neighbors, say the node at $(0.9, 0.9)$, but it is clear that simply merging it and obtaining a rectangle cannot be done.
- ▶ In this case, the node at $(0.9, 0.9)$ will simply take care of that region and inform the old neighbors of this fact



Overlay Networks

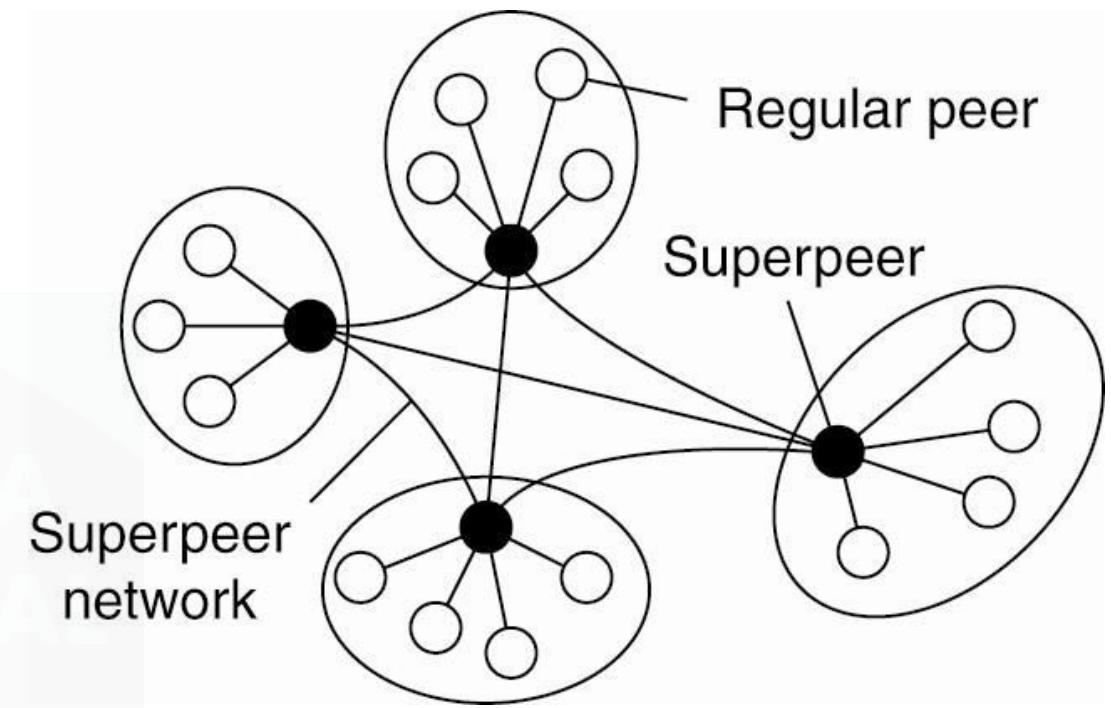
- ▶ Nodes act as both client and server; interaction is symmetric
- ▶ Each node acts as a server for part of the total system data
- ▶ Overlay networks **connect nodes in the P2P system.**
- ▶ An overlay network can be thought of as a computer network on top of another network. All nodes in an overlay network are connected with one another by means of logical or virtual links and each of these links correspond to a path in the underlying network.
- ▶ A link between two nodes in the overlay may consist of several physical links.
- ▶ Messages in the overlay are sent to logical addresses, not physical (IP) addresses
- ▶ Various approaches used to resolve logical addresses to physical.

Unstructured P2P Architectures

- ▶ Largely relying on randomized algorithm to construct the overlay network
 - Each node has a list of neighbors, which is more or less
- ▶ Many systems try to construct an overlay network that resembles a random graph
 - Each node maintains a partial view, i.e., a set of live nodes randomly chosen from the current set of nodes constructed in a random way
- ▶ An unstructured P2P network is formed when the overlay links are established arbitrarily.
- ▶ Data items are randomly mapped to some node in the system & lookup is random, unlike the structured lookup in Chord.
- ▶ Such networks can be easily constructed as a new peer that wants to join the network can copy existing links of another node and then form its own links over time.
- ▶ In an unstructured P2P network, if a peer wants to find a desired piece of data in the network, the query has to be flooded through the network in order to find as many peers as possible that share the data..

Superpeers

- ▶ Used to address the following question
 - How to find data items in unstructured P2P systems
 - Flood the network with a search query?
- ▶ An alternative is using **superpeers**
 - Nodes such as those maintaining an index or acting as a broker are generally referred to as superpeers
 - They hold index of info. from its associated peers (i.e. selected representative of some of the peers)



A hierarchical organization of nodes into a superpeer network

Finding Data Items

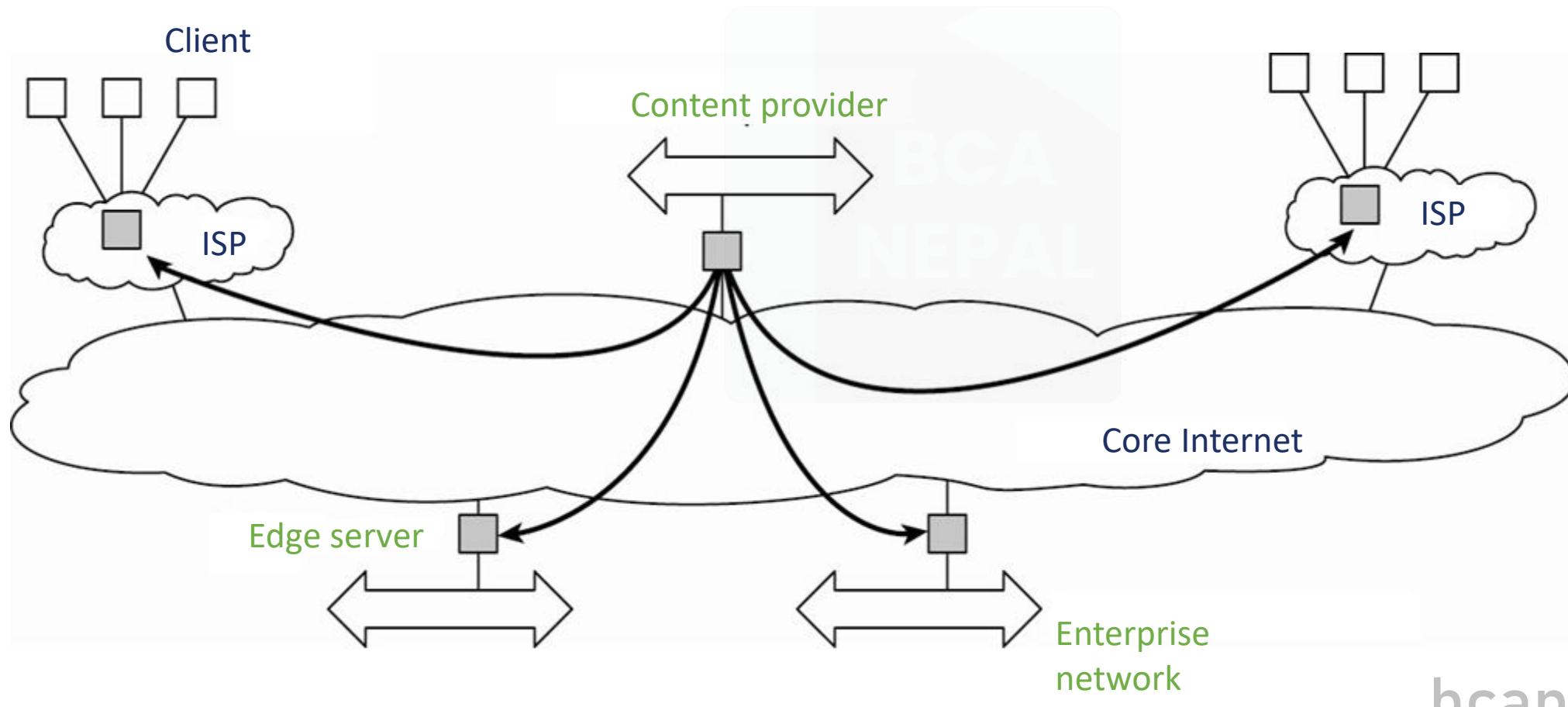
- ▶ This is quite challenging in unstructured P2P systems
 - Assume a data item is randomly placed
- ▶ Solution 1: Flood the network with a search query
- ▶ Solution 2: A randomized algorithm
 - Let us first assume that
 - Each node knows the IDs of k other randomly selected nodes
 - The ID of the hosting node is kept at m randomly picked nodes
 - The search is done as follows
 - Contact k direct neighbors for data items
 - Ask your neighbors to help if none of them knows
 - What is the probability of finding the answer directly?

Hybrid Architectures

- ▶ Many real distributed systems combine architectural features
 - E.g., the superpeer networks – combine client-server architecture (centralized) with peer-to-peer architecture (decentralized)
- ▶ Two examples of hybrid architectures
 - Edge-server systems
 - Collaborative distributed systems
 - Superpeer networks

Edge-Server Systems

- ▶ Deployed on the Internet where servers are “at the edge” of the network (i.e. first entry to network)
- ▶ Each client connects to the Internet by means of an edge server.



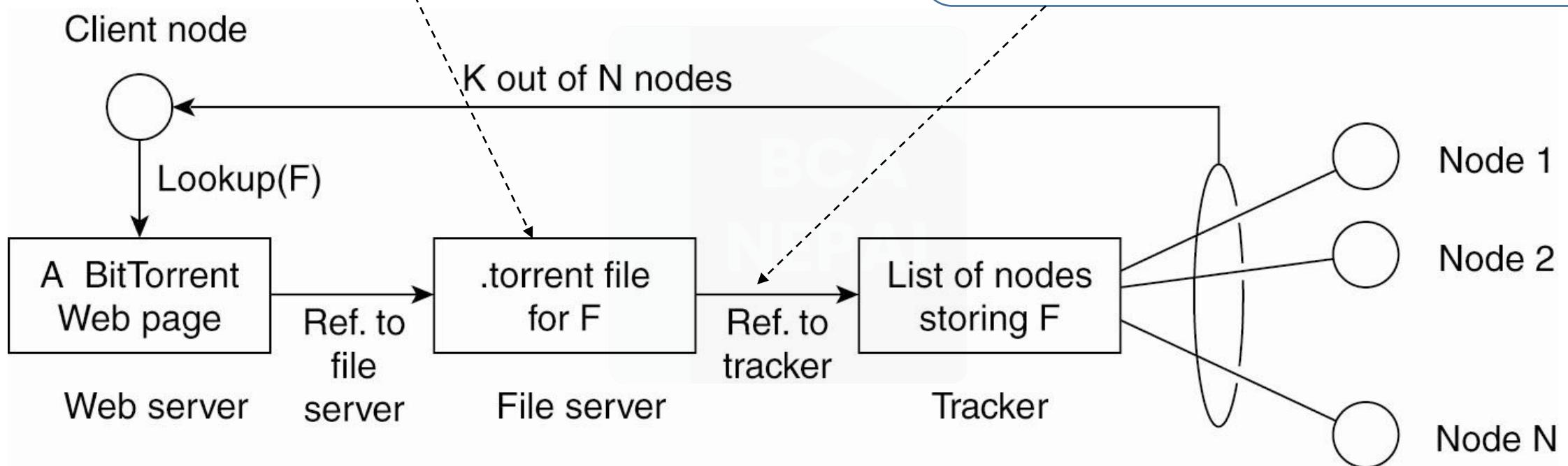
Collaborative Distributed Systems

- ▶ A hybrid distributed model that is based on mutual collaboration of various systems
 - Client-server scheme is deployed at the beginning
 - Fully decentralized scheme is used for collaboration after joining the system
- ▶ Examples of Collaborative Distributed System:
 - **BitTorrent**: is a P2P File downloading system. It allows download of various chunks of a file from other users until the entire file is downloaded
 - **Globule**: A Collaborative content distribution network. It allows replication of web pages by various web servers

BitTorrent

Information needed to download a specific file

Many trackers, one per file, tracker holds which node holds which chunk of the file

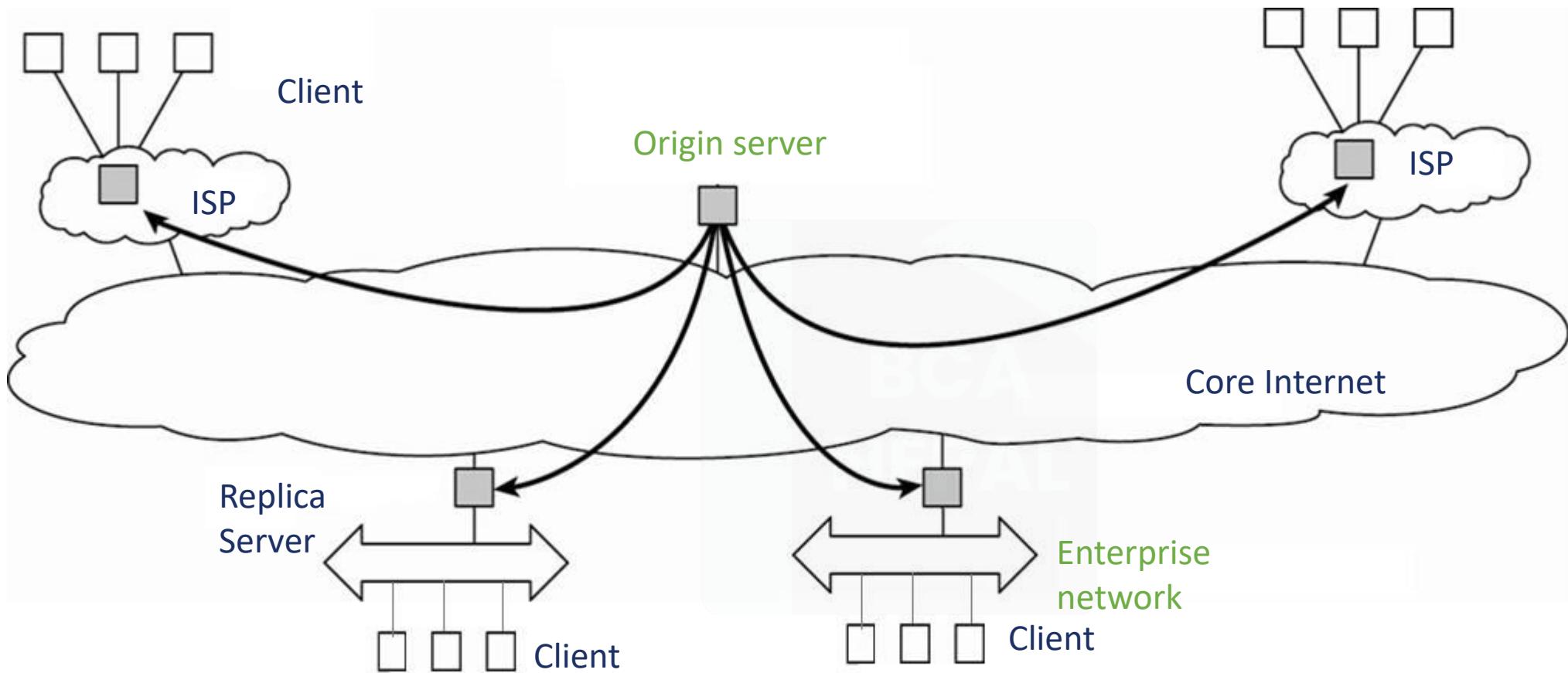


The principal working of BitTorrent (Pouwelse et al. 2004).

Globule

- ▶ Collaborative content distribution network:
 - Similar to edge-server systems
 - Enhanced web servers from various users that replicates web pages
- ▶ Components
 - A component that can redirect client requests to other servers.
 - A component for analyzing access patterns.
 - A component for managing the replication of Web pages.
- ▶ It Has a centralized component for registering the servers and make these servers known to others

Globule



- ▶ To find more information on p2p architecture. Read this paper
- ▶ *<https://snap.stanford.edu/class/cs224w-readings/lua04p2p.pdf>*



Distributed System



Unit:3

Unit 3. Processes

- 3.1 Threads**
- 3.2 Virtualization**
- 3.3 Clients**
- 3.4 Servers**
- 3.5 Code Migration**

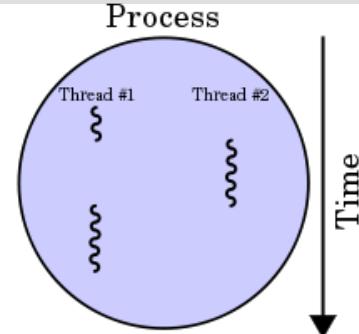


Introduction to Threads

- ▶ **Processor**: Provides a set of instructions along with the capability of automatically executing a series of those instructions.
- ▶ **Thread**: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.
- ▶ **Process**: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

What is Threads?

- ▶ Thread is a **light weight process** created by a process.
- ▶ Thread is a single sequence of execution within a process.
- ▶ Thread has its own.
 - Program counter that keeps track of which instruction to execute next.
 - System registers which hold its current working variables.
 - Stack which contains the execution history.
- ▶ Processes are generally used to execute large, '**heavyweight**' jobs such as working in word, while threads are used to carry out smaller or '**lightweight**' jobs such as auto saving a word document.
- ▶ A thread shares few information with its peer threads (having same input) like code segment, data segment and open files.



Process & Thread

► Similarities between Process & Thread

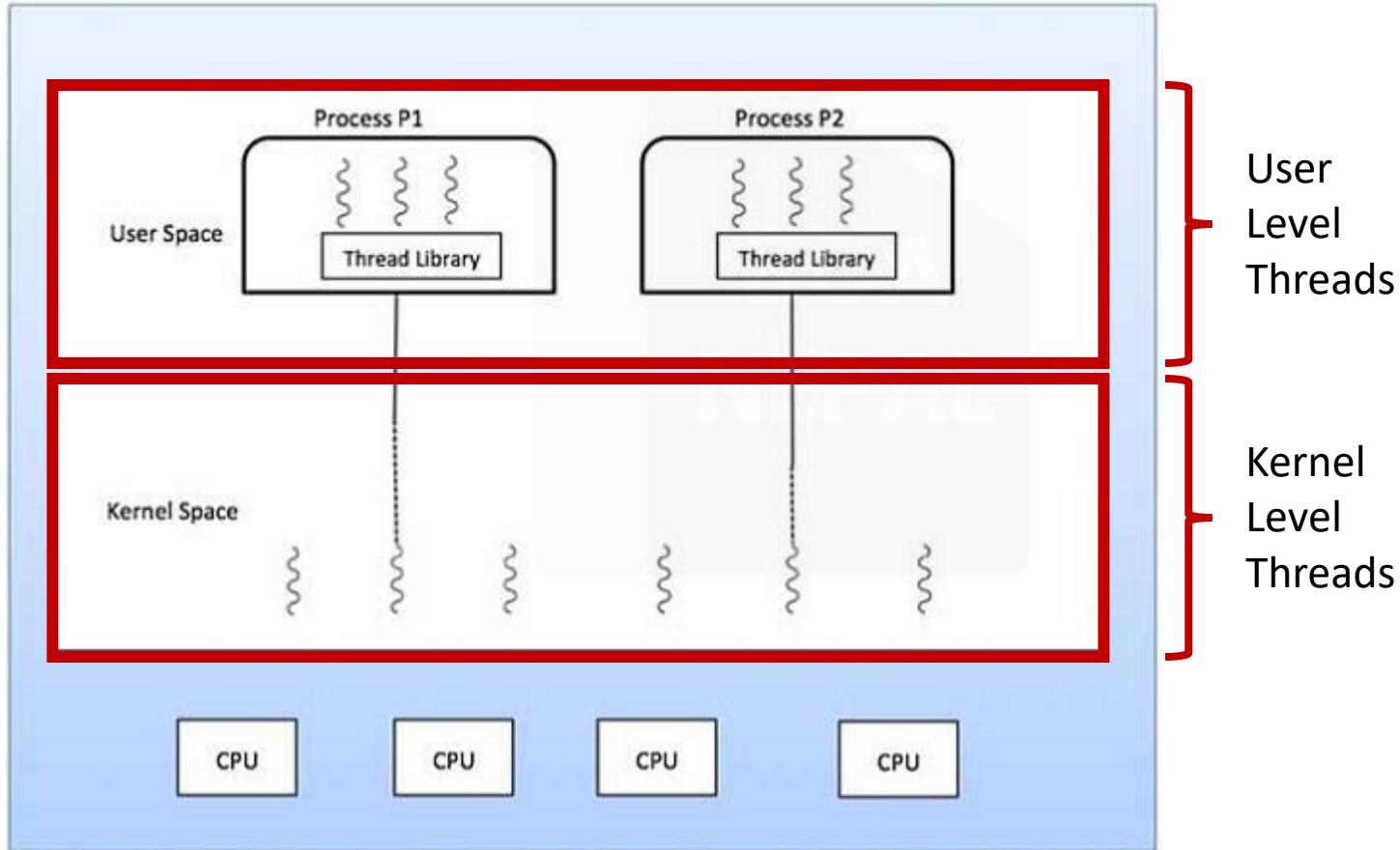
- Like processes, threads share CPU and only one thread is running at a time.
- Like processes, threads within a process execute sequentially.
- Like processes, thread can create children.
- Like a traditional process, a thread can be in any one of several states: running, blocked, ready or terminated.
- Like process, threads have Program Counter, Stack, Registers and State.

Process Vs. Thread

Process	Thread
Process means a program is in execution.	Thread means a segment of a process.
The process is not Lightweight.	Threads are Lightweight.
The process takes more time to terminate.	The thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes

Types of Threads

1. Kernel Level Thread
2. User Level Thread



User Level Thread Vs. Kernel Level Thread

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	Kernel threads are implemented by OS.
OS doesn't recognize user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complex.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Context switch requires hardware support.
If one user level thread performs blocking operation then entire process will be blocked.	If one kernel thread performs blocking operation then another thread within same process can continue execution.
Example : Java thread	Example : Windows Solaris

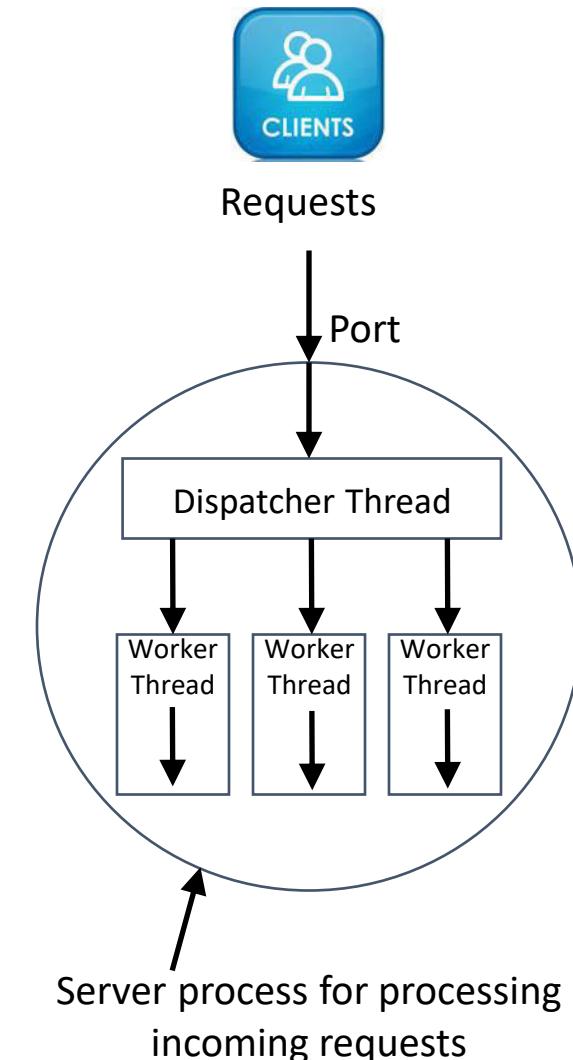
Models for Organizing Threads

- ▶ Depending on the application's needs, the threads of a process of the application can be organized in different ways.
- ▶ Threads can be organized by three different models.
 1. Dispatcher/Worker model
 2. Team model
 3. Pipeline model



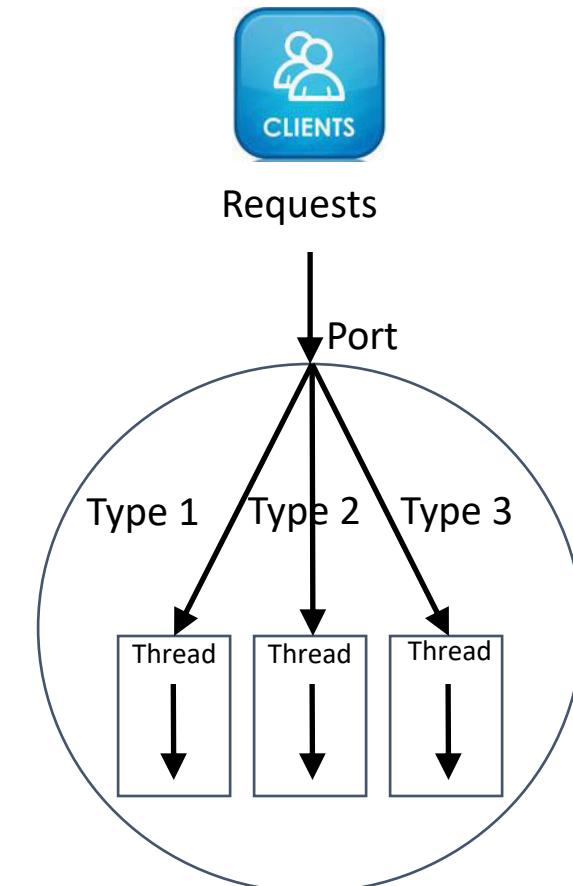
Dispatcher/Worker Model

- In this model, the process consists of a single **dispatcher thread** and multiple **worker threads**.
- The dispatcher thread:
 - Accepts requests from clients.
 - Examine the request.
 - Dispatches the request to one of the free worker threads for further processing of the request.
- Each worker thread works on a different client request.
- Therefore, multiple client requests can be processed in parallel.



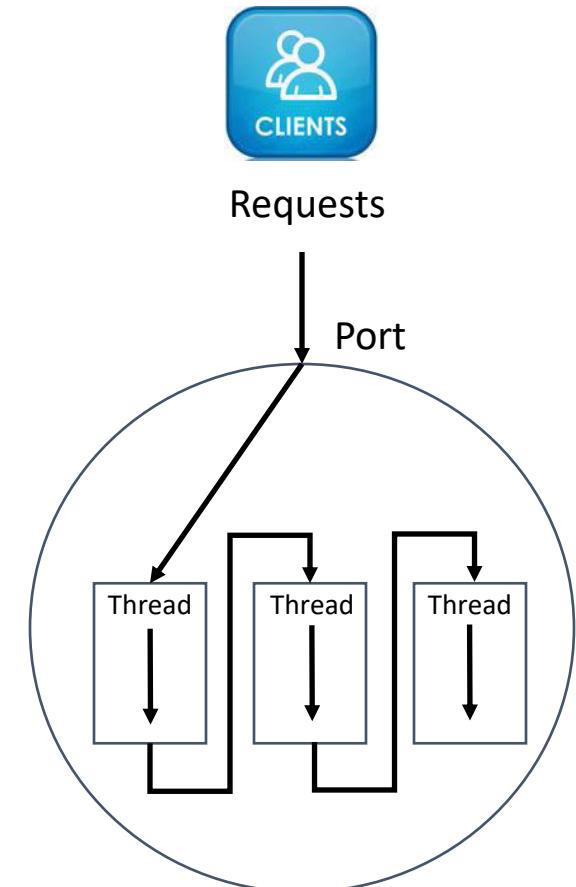
Team Model

- ▶ In this model, all **threads behave as equal**.
- ▶ Each thread gets and processes clients requests on its own.
- ▶ This model is often used for implementing specialized threads within a process.
- ▶ Each thread of the process is specialized in servicing a specific type of requests like copy, save, autocorrect.



Pipeline Model

- ▶ This model is useful for applications based on the **producer-consumer model**.
- ▶ The output data generated by one part of the application is used as input for another part of the application.
- ▶ The threads of a process are organized as a **pipeline**.
- ▶ The output data generated by the first thread is used for processing by the second thread, the output of the second thread is used for third thread, and so on.
- ▶ The output of the last thread in the pipeline is the final output of the process to which the threads belong.



Threads in Distributed system

- ▶ An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running.
- ▶ This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.
- ▶ We illustrate this point by taking a closer look at multithreaded clients and servers, respectively.
- ▶ **Multithreaded Clients**
- ▶ To establish a high degree of distribution transparency, distributed systems that operate in wide-area networks may need to conceal long interprocess message propagation times. The round-trip delay in a wide-area network can easily be in the order of hundreds of milliseconds, or sometimes even seconds.

- ▶ The usual way to hide communication latencies is to initiate communication and immediately proceed with something else. A typical example where this happens is in Web browsers. In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc. To fetch each element of a Web document, the browser has to set up a TCP/IP connection, read the incoming data, and pass it to a display component. Setting up a connection as well as reading incoming data are inherently blocking operations. When dealing with long-haul communication, we also have the disadvantage that the time for each operation to complete may be relatively long.
- ▶ A Web browser often starts with fetching the HTML page and subsequently displays it. To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in. While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page, such as the images. The latter are displayed as they are brought in. The user need thus not wait until all the components of the entire page are fetched before the page is made available.

- ▶ In effect, it is seen that the Web browser is doing a number of tasks simultaneously. As it turns out, developing the browser as a multithreaded client simplifies matters considerably. As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process. As is also illustrated in Stevens (1998), the code for each thread is the same and, above all, simple. Meanwhile, the user notices only delays in the display of images and such, but can otherwise browse through the document.
- ▶ There is another important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously. In the previous example, several connections were set up to the same server. If that server is heavily loaded, or just plain slow, no real performance improvements will be noticed compared to pulling in the files that make up the page strictly one after the other.

▶ However, in many cases, Web servers have been replicated across multiple machines, where each server provides exactly the same set of Web documents. The replicated servers are located at the same site, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load-balancing technique (Katz et al., 1994). When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a nonreplicated server. This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.

▶ Multithreaded Servers

▶ Although there are important benefits to multithreaded clients, as we have seen, the main use of multithreading in distributed systems is found at the server side. Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uniprocessor systems. However, now that multiprocessor computers are widely available as general-purpose workstations, multithreading for parallelism is even more useful.

- ▶ To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk. The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply. Here one thread, the dispatcher, reads incoming requests for a file operation. The requests are sent by clients to a well-known end point for this server. After examining the request, the server chooses an idle (i.e., blocked) worker thread and hands it the request.
- ▶ The worker proceeds by performing a blocking read on the local file system, which may cause the thread to be suspended until the data are fetched from disk. If the thread is suspended, another thread is selected to be executed. For example, the dispatcher may be selected to acquire more work. Alternatively, another worker thread can be selected that is now ready to run.

- ▶ Now consider how the file server might have been written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled. In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.
- ▶ So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.

- ▶ However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message. The next message may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed and subsequently sent to the client. In this scheme, the server will have to make use of nonblocking calls to send and receive.
- ▶ In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table for every message sent and received. In effect, we are simulating threads and their stacks the hard way. The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

- ▶ It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., an RPC to talk to the disk) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease and simplicity of blocking system calls, but gives up some amount of performance. The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls, thus is hard to program.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Figure 3-4. Three ways to construct a server.

Introduction to Virtualization

- ▶ Threads and processes can be seen as a way to do more things at the same time. In effect, they allow us build (pieces of) programs that appear to be executed simultaneously. On a single-processor computer, this simultaneous execution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time. By rapidly switching between threads and processes, the illusion of parallelism is created.
- ▶ This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as resource virtualization. This virtualization has been applied for many decades, but has received renewed interest as (distributed) computer systems have become more commonplace and complex, leading to the situation that application software is mostly always outliving its underlying systems software and hardware. In this section, we pay some attention to the role of virtualization and discuss how it can be realized.

Virtualization

- ▶ Multiprogrammed operating systems provide the illusion of simultaneous execution through ***resource virtualization***
 - Use software to make it look like concurrent processes are executing simultaneously
- ▶ Virtual machine technology creates separate virtual machines, capable of supporting multiple instances of different operating systems.
- ▶ Virtualization is a broad term that refers to the abstraction of computer resources.
- ▶ Virtualization creates an external interface that hides an underlying implementation
- ▶ Benefits:
 - Hardware changes faster than software
 - Compromised systems (internal failure or external attack) are isolated.
 - Run multiple different operating systems at the same time

Virtualization

Common uses of the term, divided into two main categories:

- Platform virtualization
- Resource virtualization

► Platform virtualization:

- It involves the simulation of virtual machines.
- Platform virtualization is performed on a given hardware platform by "host" software (a control program), which creates a simulated computer environment (a virtual machine) for its "guest" software.
- The "guest" software, which is often itself a complete operating system, runs just as if it were installed on a stand-alone hardware platform.

► Resource virtualization:

- It involves the simulation of combined, fragmented, or simplified resources.
- Virtualization of specific system resources, such as storage volumes, name spaces, and network resources.

Architectures of Virtual Machines

- ▶ To understand the differences in virtualization, it is important to realize that computer systems generally offer four different types of interfaces, at four different levels:
- ▶ 1. An interface between the hardware and software, consisting of machine instructions that can be invoked by any program.
- ▶ 2. An interface between the hardware and software, consisting of machine instructions that can be invoked only by privileged programs, such as an operating system.
- ▶ 3. An interface consisting of system calls as offered by an operating system.
- ▶ 4. An interface consisting of library calls, generally forming what is known as an application programming interface (API). In many cases, the aforementioned system calls are hidden by an API.

- ▶ These different types are shown in Fig. 3-6. The essence of virtualization is to mimic the behavior of these interfaces.

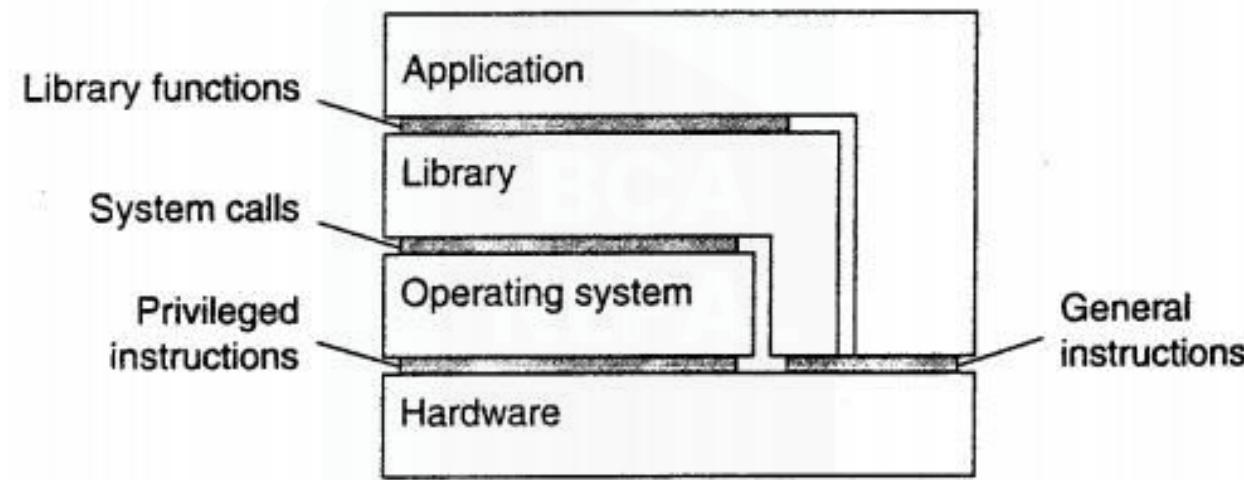
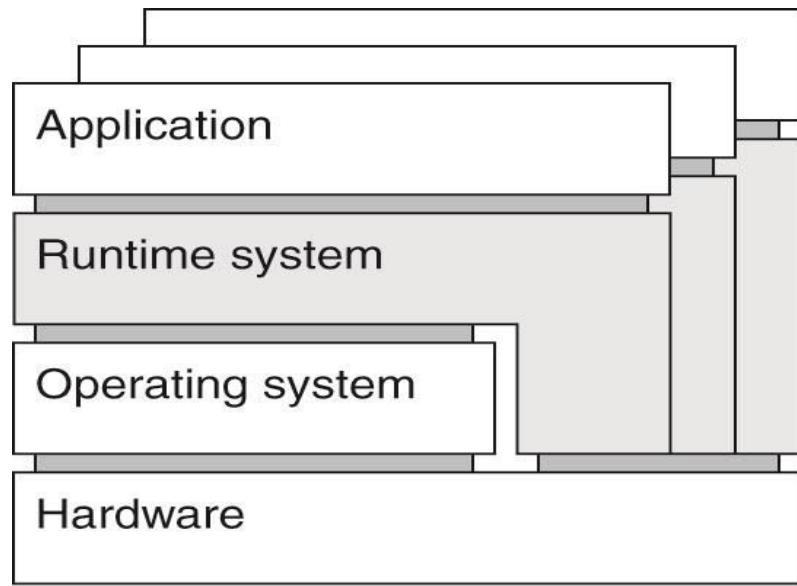


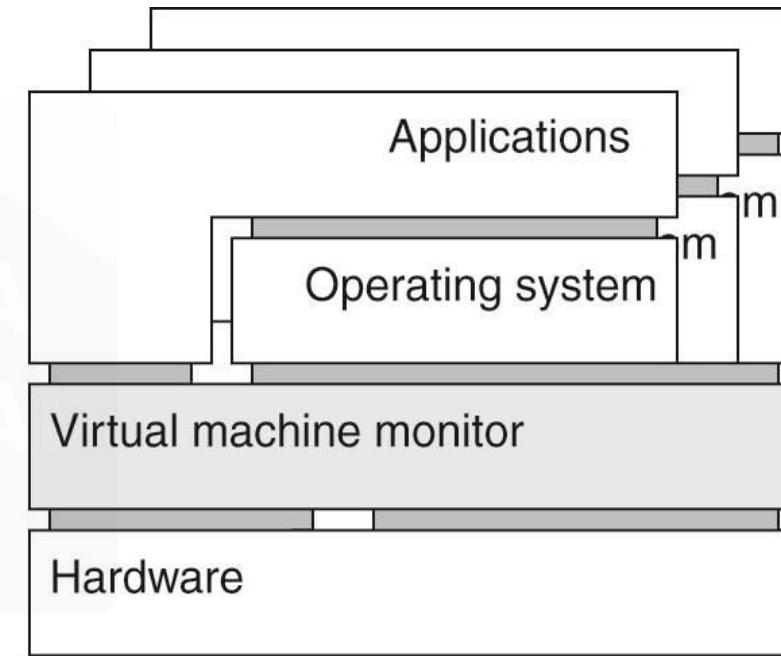
Figure 3-6. Various interfaces offered by computer systems.

Two Ways to Virtualize

Process Virtual Machine



Virtual Machine Monitor



- A process virtual machine, with multiple instances of (application, runtime) combinations.

- A **virtual machine monitor**, with multiple instances of (applications, operating system) combinations.

Clients

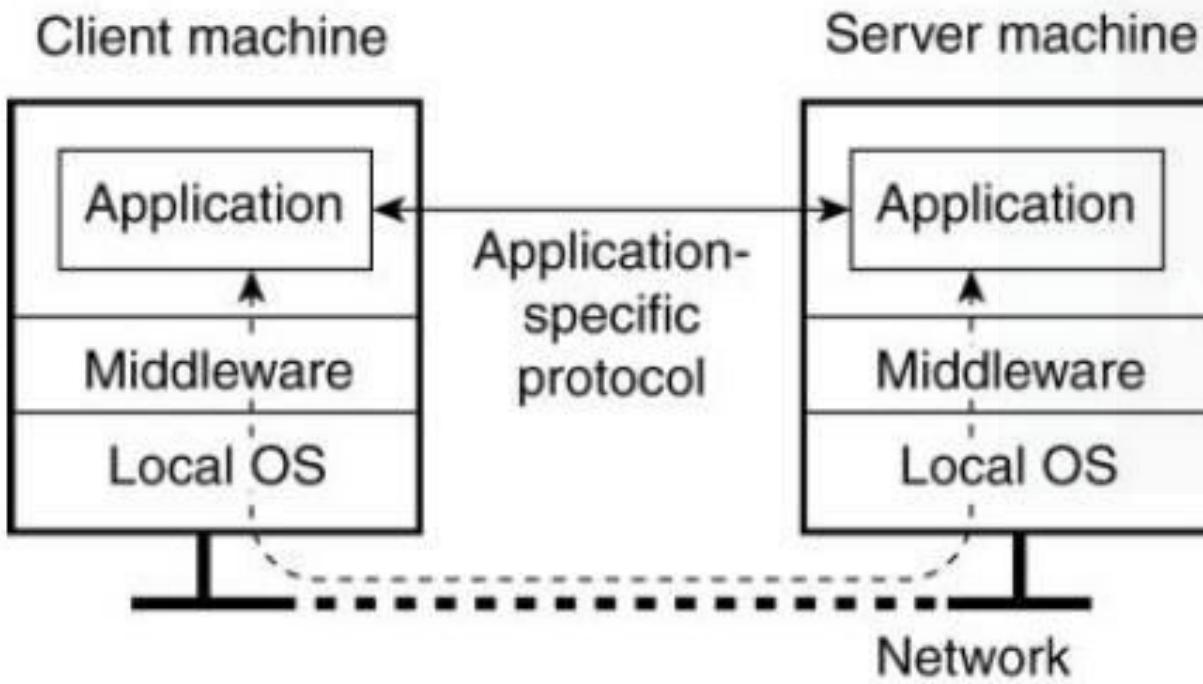
- ▶ A Program, Which interact with a human user or a remote servers
- ▶ Typically, the users interact with the client via a GUI
- ▶ The client is the machine (workstation or PC) running the front-end applications.
- ▶ It interacts with a user through the keyboard, display, and pointing device such as a mouse.
- ▶ The client has no direct data access responsibilities. It simply requests processes from the server and displays data managed by the server.

Networked User Interfaces

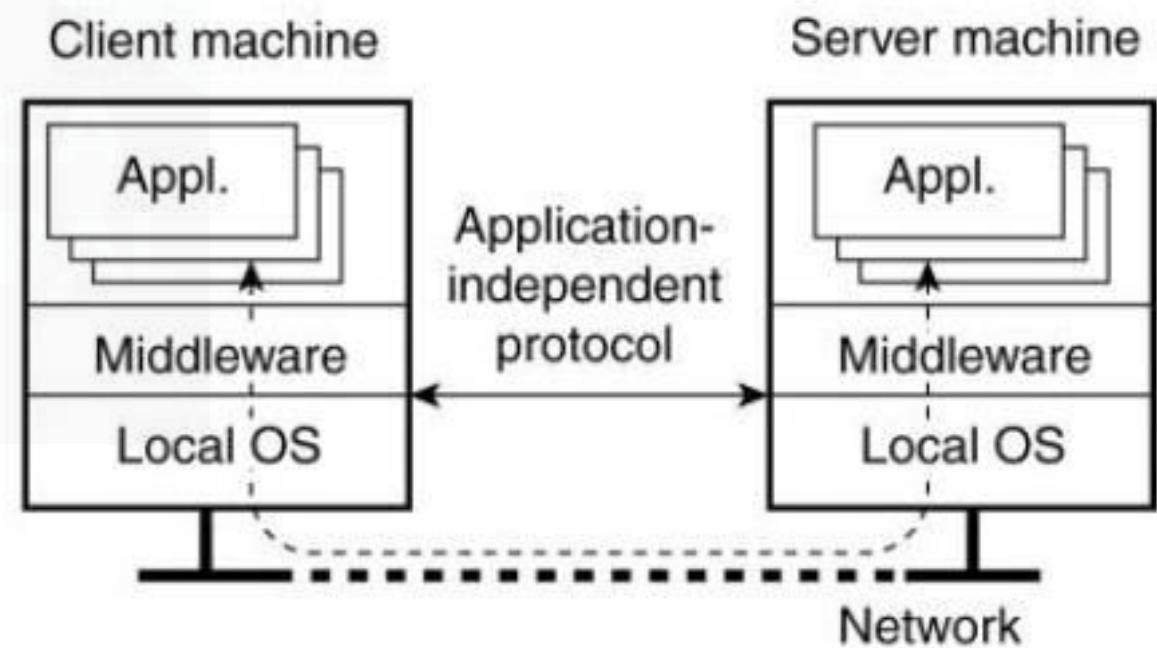
- ▶ A major task of client machines is to provide the means for users to interact with remote servers
- ▶ Two ways to support client-server interaction:
 1. For each remote service - the client machine will have a separate counterpart that can contact the service over the network.
 - Example: an agenda running on a user's PDA that needs to synchronize with a remote, possibly shared agenda.
 - In this case, an application-level protocol will handle the synchronization
 2. Provide direct access to remote services by only offering a convenient user interface.
 - The client machine is used only as a terminal with no need for local storage, leading to an application neutral solution.
 - In the case of networked user interfaces, everything is processed and stored at the server.
 - This thin-client approach is receiving more attention as Internet connectivity increases, and hand-held devices are becoming more sophisticated.

Networked User Interfaces

Networked application with its own protocol

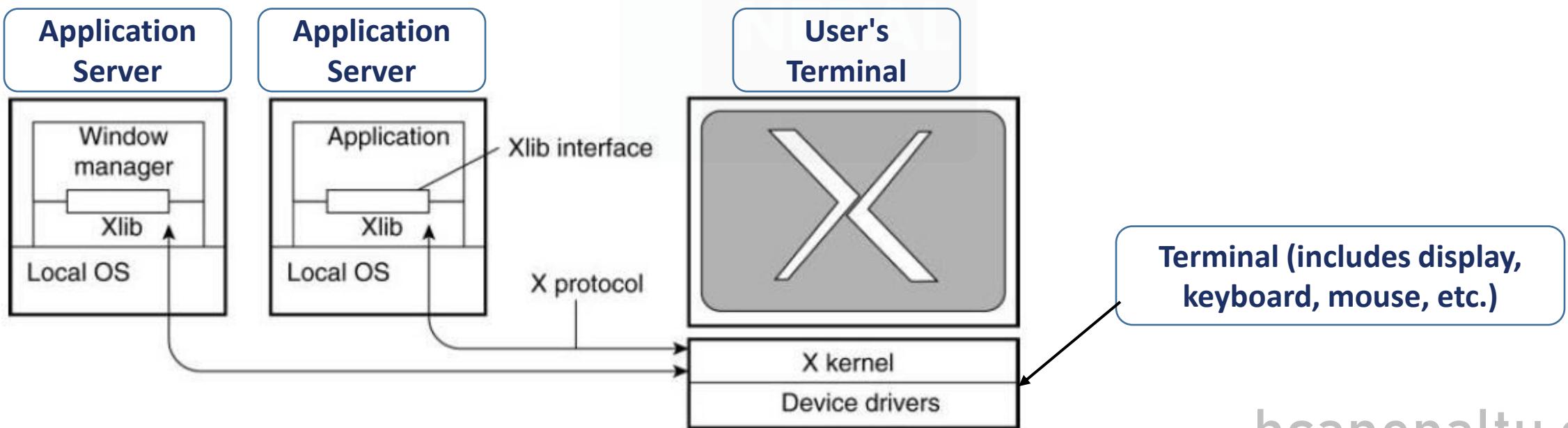


General solution to allow access to remote applications.



The X Window System

- ▶ Used to control bit-mapped terminals, which include a monitor, keyboard, and a pointing device such as a mouse.
- ▶ X kernel is heart of the system.
 - Contains all the terminal-specific device drivers - highly hardware dependent
 - X kernel offers a low-level interface for controlling the screen and for capturing events from the keyboard and mouse
 - This interface is made available to applications as a library called Xlib.



Compound Documents

- ▶ Modern user interfaces do a lot more than systems such as X or its simple applications. In particular, many user interfaces allow applications to share a single graphical window, and to use that window to exchange data through user actions. Additional actions that can be performed by the user include what are generally called drag-and-drop operations, and in-place editing, respectively.
- ▶ A typical **example of drag-and-drop functionality** is moving an icon representing a file A to an icon representing a trash can, resulting in the file being deleted. In this case, the user interface will need to do more than just arrange icons on the display: it will have to pass the name of the file A to the application associated with the trash can as soon as A's icon has been moved above that of the trash can application.
- ▶ **In-place editing** can best be illustrated by means of a document containing text and graphics. Imagine that the document is being displayed within a standard word processor. As soon as the user places the mouse above an image, the user interface passes that information to a drawing program to allow the user to modify the image.

- ▶ **For example**, the user may have rotated the image, which may effect the placement of the image in the document. The user interface therefore finds out what the new height and width of the image are, and passes this information to the word processor. The latter, in tum, can then automatically update the page layout of the document
- ▶ The key idea behind these user interfaces is the notion of a **compound document**, which can be defined as a collection of documents, possibly of very different kinds (like text, images, spreadsheets, etc.), which are seamlessly integrated at the user-interface level.
- ▶ A user interface that can handle compound documents hides the fact that different applications operate on different parts of the document.
- ▶ To the user, all parts are integrated in a seamless way. When changing one part affects other parts, the user interface can take appropriate measures, for example, by notifying the relevant applications.

Client-Side Software for Distribution Transparency

▶ Access transparency

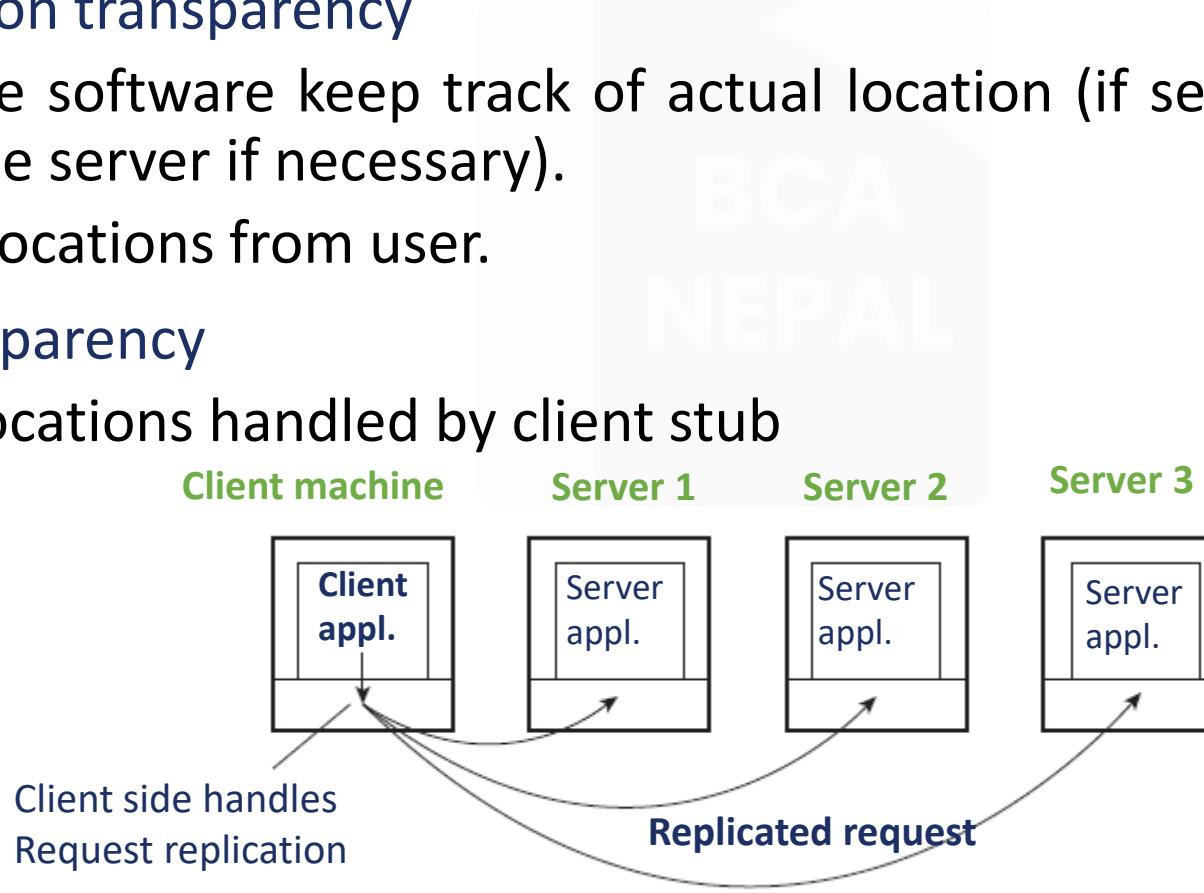
- Handled through client-side stubs for RPCs
- Same interface as at the server, hides different machine architectures

▶ Location/migration transparency

- let client-side software keep track of actual location (if server changes location: client rebinds to the server if necessary).
- Hide server locations from user.

▶ Replication transparency

- multiple invocations handled by client stub



Client-Side Software for Distribution Transparency

▶ Failure transparency

- Mask server and communication failures: done through client middleware,
- e.g. connect to another machine.

▶ Concurrency transparency

- Handled through special intermediate servers, notably transaction monitors, and requires less support from client software.

Servers

- ▶ A server is a process implementing a specific service on behalf of a collection of clients.
 - Each server is organized in the same way:
 - It waits for an incoming request from a client ensures that the request is fulfilled
 - It waits for the next incoming request.
- ▶ Types of server
 1. Iterative server
 2. Concurrent server

Iterative server and Concurrent server

▶ Iterative server

- Iterative server handles request, then returns results to the client; any new client requests must wait for previous request to complete (also useful to think of this type of server as sequential).
- Process one request at a time
- When an iterative server is handling a request, other connections to that port are blocked.
- The incoming connections must be handled one after another.
- Iterative servers support a single client at a time.
- Much easier to build, but usually much less efficient

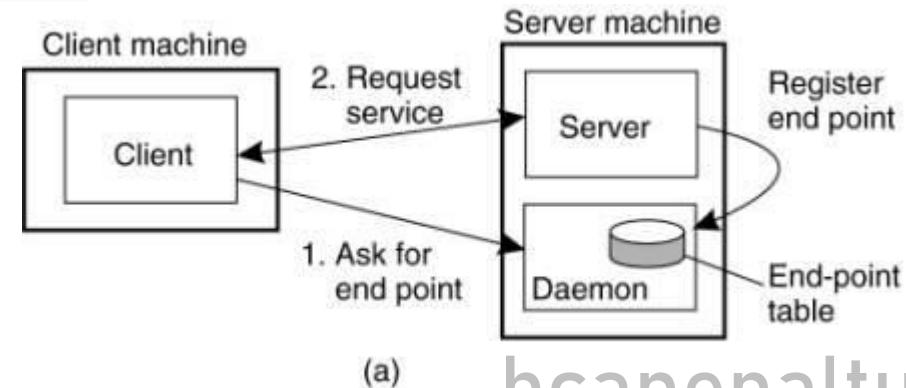
► Concurrent server

- Process multiple requests simultaneously.
- Concurrent servers support multiple clients concurrently (may or may not use concurrent processes)
- Clients queue for connection, then are served concurrently. The concurrency reduces latency significantly.
- Concurrent server does not handle the request itself; a separate thread or sub-process handles the request and returns any results to the client; the server is then free to immediately service the next client (i.e., there's no waiting, as service requests are processed in parallel).
- A multithreaded server is an example of a concurrent server.

- ▶ **Where do clients contact a server?**
- ▶ Clients send requests to an end point, also called a port, at the machine where the server is running.
- ▶ Each server listens to a specific end point.
- ▶ How do clients know the end point of a service?
- ▶ 1. Globally assign end points for well-known services.
- ▶ Examples:
 - ▶ 1. Servers that handle Internet FTP requests always listen to TCP port 21.
 - ▶ 2. An HTTP server for the World Wide Web will always listen to TCP port 80.
- ▶ These end points have been assigned by the Internet Assigned Numbers Authority (IANA).
- ▶ With assigned end points, the client only needs to find the network address of the machine where the server is running.

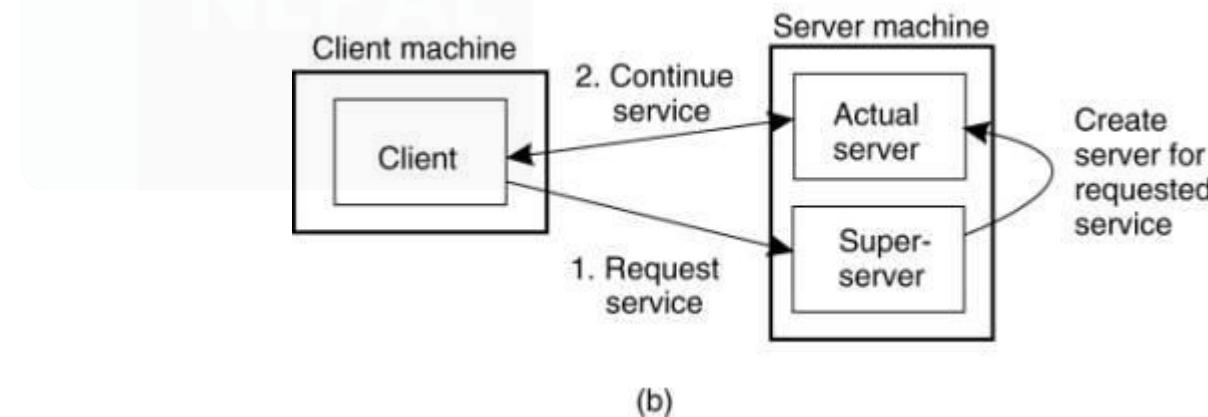
Client-to-server binding using a daemon

- ▶ Many services that do not require a pre-assigned end point.
- ▶ Example: A time-of-day server may use an end point that is dynamically assigned to it by its local operating system.
- ▶ A client will look need to up the end point.
- ▶ Solution: a daemon running on each machine that runs servers.
- ▶ The daemon keeps track of the current end point of each service implemented by a co-located server.
- ▶ The daemon itself listens to a well-known end point.
- ▶ A client will first contact the daemon, request the end point, and then contact the specific server (Figure(a))



Client-to-server binding using a superserver

- ▶ Common to associate an end point with a specific service.
- ▶ Implementing each service by means of a separate server may be a waste of resources.
- ▶ Example: UNIX system
- ▶ many servers run simultaneously, with most of them passively waiting for a client request.
- ▶ Instead of having to keep track of so many passive processes, it is often more efficient to have a single superserver listening to each end point associated with a specific service, (Figure (b))



Stateless server

- ▶ A stateless server is a server that treats each request as an independent transaction that is unrelated to any previous request.
- ▶ Example: A Web server is stateless.
 - It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or (most often) for fetching a file.
 - When the request has been processed, the Web server forgets the client completely.
 - The collection of files that a Web server manages (possibly in cooperation with a file server), can be changed without clients having to be informed.

Form of a stateless design - **soft state**.

- ➔ The server promises to maintain state on behalf of the client, but only for a limited time.
- ➔ After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client.

Stateful server

- ▶ A stateful server remembers client data (state) from one request to the next.
- ▶ Information needs to be explicitly deleted by the server.
- ▶ Example:
 - A file server that allows a client to keep a local copy of a file, even for performing update operations.
 - The server maintains a table containing (client, file) entries.
 - This table allows the server to keep track of which client currently has the update permissions on which file and the most recent version of that file
- ▶ Improves performance of read and write operations as perceived by the client

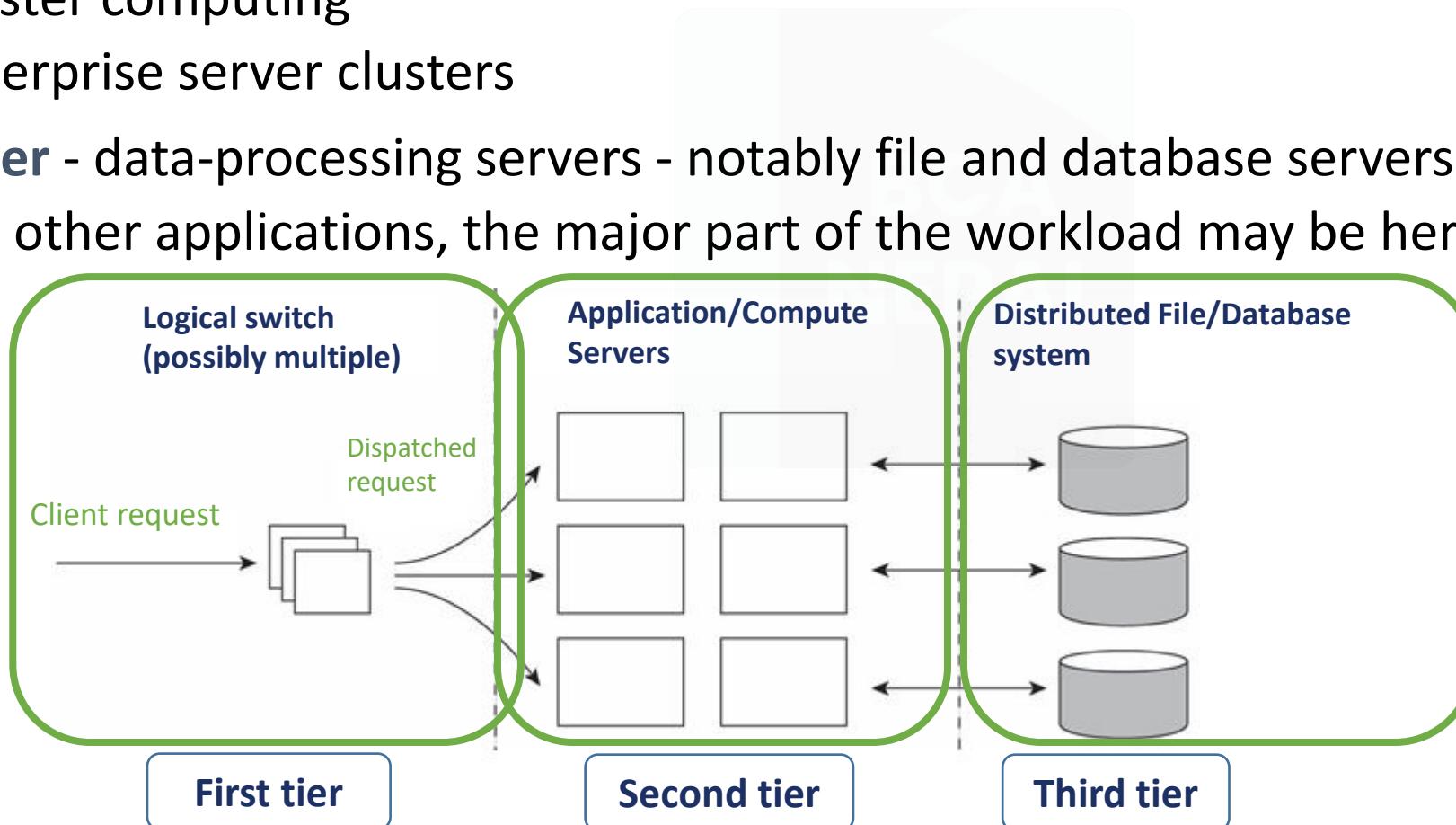
Server Clusters

- ▶ A server cluster is a collection of machines connected through a network, where each machine runs one or more servers.
- ▶ A server cluster is logically organized into three tiers
 - First tier
 - Second tier
 - Third tier



Server Clusters

- ▶ **First tier** - consists of a (logical) switch through which client requests are routed
 - The switch (access/replication transparency)
- ▶ **Second tier** - application processing
 - Cluster computing
 - Enterprise server clusters
- ▶ **Third tier** - data-processing servers - notably file and database servers
 - For other applications, the major part of the workload may be here



- ▶ **Issue:** When a server cluster offers, multiple different machines may run different application servers.
- ▶ The switch will have to be able to distinguish services or otherwise it cannot forward requests to the proper machines.
- ▶ Many second-tier machines run only a single application.
- ▶ This limitation comes from dependencies on available software and hardware, but also that different applications are often managed by different administrators.
- ▶ Consequence - certain machines are temporarily idle, while others are receiving an overload of requests.
- ▶ **Solution:** Temporarily migrate services to idle machines to balance load.
- ▶ Use virtual machines allowing a relative easy migration of code to real machines.

Distributed Servers

- ▶ The server clusters discussed so far are generally rather statically configured. In these clusters, there is often an separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.
- ▶ As we mentioned, most server clusters offer a single access point. When that point fails, the cluster becomes unavailable. To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available. For example, the Domain Name System (DNS) can return several addresses, all belonging to the same host name. This approach still requires clients to make several attempts if one of the addresses fails. Moreover, this does not solve the problem of requiring static access points.
- ▶ Having stability, like a long-living access point, is a desirable feature from a client's and a server's perspective. On the other hand, it also desirable to have a high degree of flexibility in configuring a server cluster, including the switch. This observation has lead to a design of a distributed server which effectively is nothing but a possibly dynamically changing set of machines, with also possibly varying access points, but which nevertheless appears to the outside world as a single, powerful machine.

- ▶ The basic idea behind a distributed server is that clients benefit from a robust, high-performing, stable server. These properties can often be provided by high-end mainframes, of which some have an acclaimed mean time between failure of more than 40 years. However, by grouping simpler machines transparently into a cluster, and not relying on the availability of a single machine, it may be possible to achieve a better degree of stability than by each component individually. For example, such a cluster could be dynamically configured from end-user machines, as in the case of a collaborative distributed system.

Code Migration

- ▶ Traditionally, communication in distributed systems is concerned with exchanging data between processes.
- ▶ Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target
- ▶ process migration in which an entire process is moved from one machine to another
- ▶ Code migration is often used for load distribution, reducing network bandwidth, dynamic customization, and mobile agents.
- ▶ Code migration increases scalability, improves performance, and provides flexibility.
- ▶ Reasons for Code Migration:
 - Performance
 - Flexibility

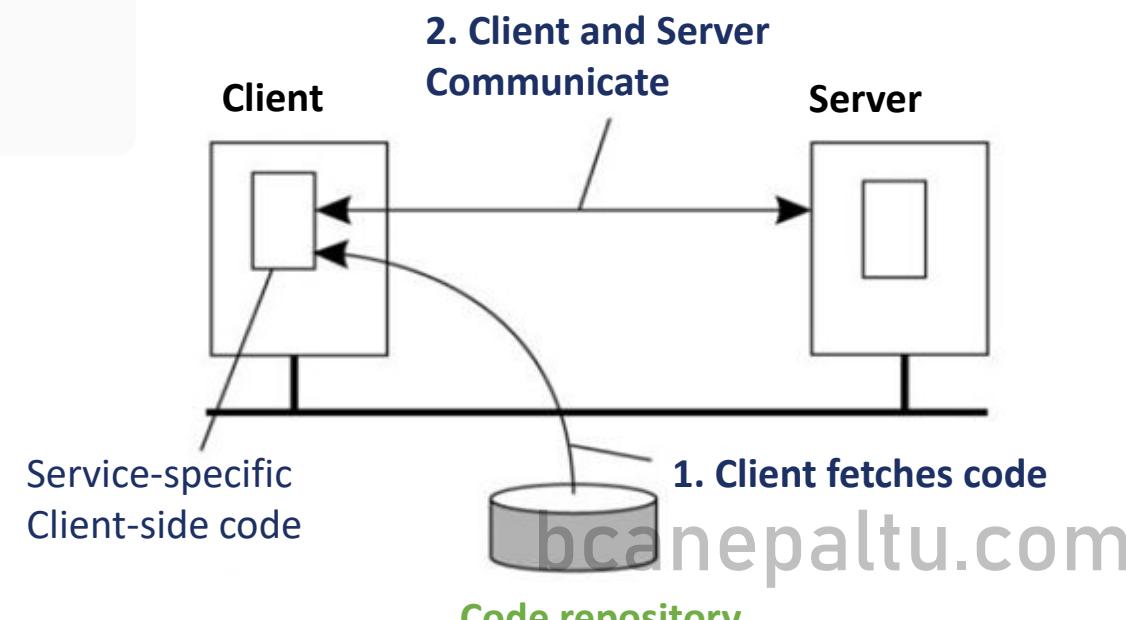
Performance in Code Migration

- ▶ Overall system performance can be improved if processes are moved from heavily-loaded to lightly loaded machines.
- ▶ How system performance is improved by code migration?
 - Using load distribution algorithms
 - Using qualitative reasoning
 - Migrating parts of the client to the server
 - Migrating parts of the server to the client

Flexibility in Code Migration

- ▶ The traditional approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed.
- ▶ For example,
 - Suppose a client program uses some proprietary APIs for doing some tasks that are rarely needed, and because of the huge size of the necessary API files, they are kept in a server.
 - If the client ever needs to use those APIs, then it can first dynamically download the APIs and then use them.

- ▶ **Advantage of this model:** Clients need not have all the software preinstalled to do common tasks.
- ▶ **Disadvantage of this model :** Security - blindly trusting that the downloaded code implements only the advertised APIs while accessing your unprotected hard disk



Models for Code Migration

- ▶ To get a better understanding of the different models for code migration, we use a framework described in Fuggetta et al. (1998).
- ▶ In this framework, a process consists of three segments.
 1. **The code segment:** It is the part that contains the set of instructions that make up the program that is being executed.
 2. **The resource segment:** It contains references to external resources needed by the process, such as files, printers, devices, other processes, and so on.
 3. **The execution segment:** It is used to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

Weak Mobility Vs. Strong Mobility

Parameters	Weak Mobility	Strong Mobility
Definition	In this model, it is possible to transfer In contrast to weak mobility, in systems that support only the code segment, along with strong mobility the execution segment can be perhaps some initialization data transferred well.	
Characteristic Feature	A transferred program is always started from its initial state	A running process can be stopped, subsequently moved to another machine, and then resume execution where it left off.
Example	Java applets – which always start D'Agents execution from the beginning	
Benefit	Simplicity	Much more general than weak mobility

Migration Initiation

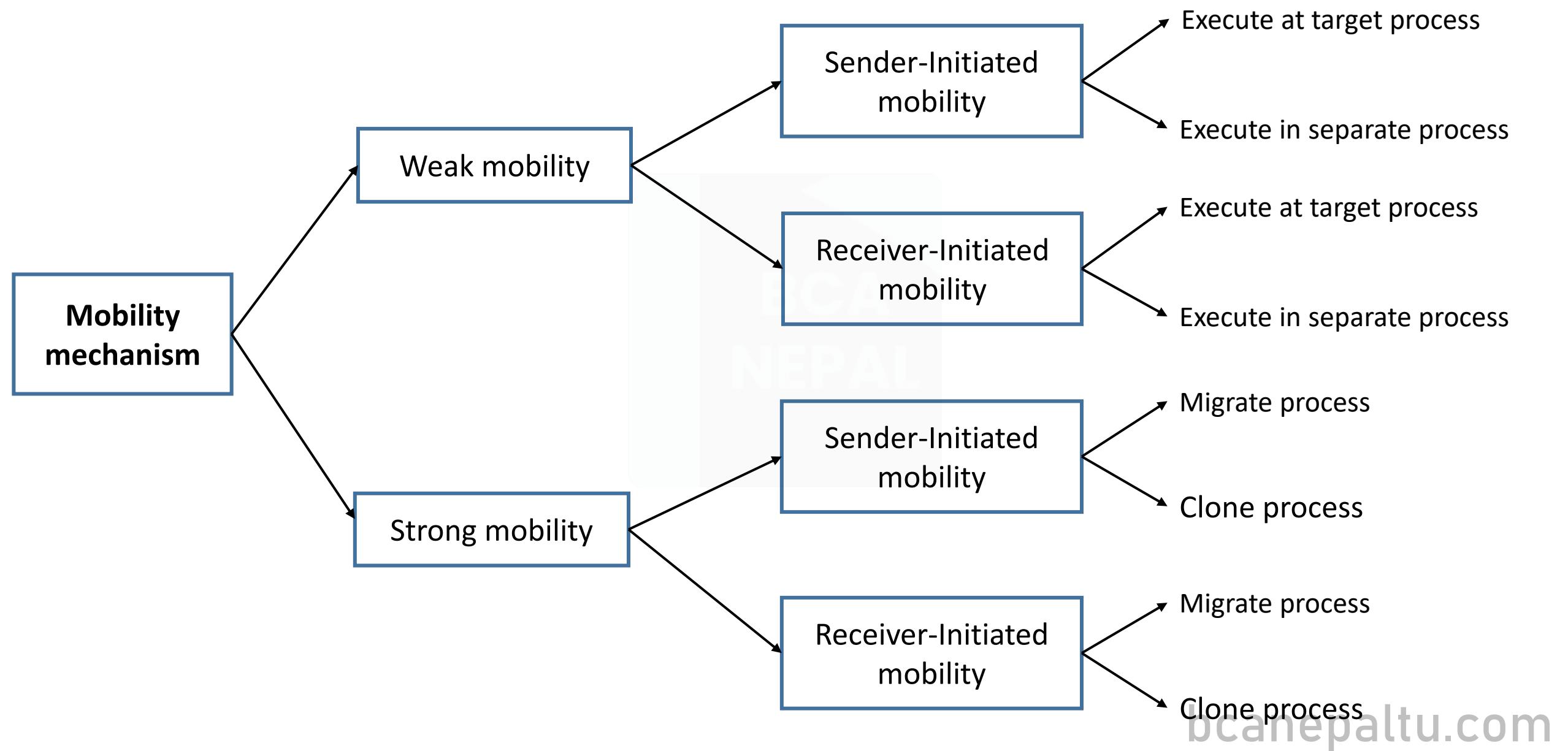
Sender-Initiated Migration

- ▶ It is initiated at the machine where the code currently resides or is being executed
- ▶ Examples:
 - Uploading programs to a compute server.
 - Sending a search program across the Internet to a web database server to perform the queries at that server.

Receiver-Initiated Migration

- ▶ The initiative for code migration is taken by the target machine
- ▶ Example: Java applets.

Alternatives for code migration



Execute Migrated Code for weak mobility

- ▶ In the case of weak mobility, it also makes a difference if the migrated code is executed by the target process, or whether a separate process is started.
- ▶ For example, Java applets are simply downloaded by a web browser and are executed in the browser's address space.
- ▶ **Benefit for executing code at target process:** There is no need to start a separate process, thereby avoiding communication at the target machine.
- ▶ **Drawback for executing code at target process:** The target process needs to be protected against malicious or inadvertent code executions.

Migrate or Clone Process (for strong mobility)

- ▶ Instead of moving a running process, also referred to as process migration, strong mobility can also be supported by remote cloning.
- ▶ In contrast to process migration, cloning yields an exact copy of the original process, but now running on a different machine.
- ▶ The cloned process is executed in parallel to the original process.
- ▶ **Benefit of cloning process:** The model closely resembles the one that is already used in many applications. The only difference is that the cloned process is executed on a different machine.
- ▶ In this sense, migration by cloning is a simple way to improve distribution transparency.

Migration and Local Resources

- ▶ What often makes code migration so difficult is that the resource segment cannot always be simply transferred along with the other segments without being changed.
- ▶ When the process moves to another location, it will have to give up the port and request a new one at the destination.
- ▶ Process-to-Resource Bindings
 1. Binding by Identifier
 2. Binding by Value
 3. Binding by Type

Process-to-Resource Bindings

Binding by Identifier

- ▶ A process refers to a resource by its identifier. In that case, the process requires precisely the referenced resource, and nothing else.
- ▶ Examples:
 - A URL to refer to a specific web site.
 - Local communication endpoints (IP, port etc.)

Binding by Value

- ▶ Only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide the same value.
- ▶ Example: Standard libraries for programming languages.

Binding by Type

- ▶ A process indicates it needs only a resource of a specific type.
- ▶ Example: References to local devices, such as monitors, printers and so on.

Resource Types

- ▶ When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding.
- ▶ Resource Types:

Unattached resources :

- They can be easily moved between different machines.
- Example: Typically (data) files associated only with the program that is to be migrated.

Fastened resources:

- They may be copied or moved, but only at relatively high costs.
- Example: Local databases and complete web sites.
- Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment.

Fixed resources:

- They are intimately bound to a specific machine or environment and cannot be moved.
- Example: Local devices, local communication end points

Managing Local Resources

- ▶ Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code

		Resource-to-machine Binding		
		Unattached	Fastened	Fixed
Process-to-resource Binding	By Identifier	MV (or GR)	GR (or MV)	GR
	By Value	CP (or MV, GR)	GR (or CP)	GR
	By Type	RB (or MV, CP)	RB (or GR, CP)	RB (or GR)

GR Establish global system wide reference

MV Move the resource

CP Copy the value of the resource

RB Re-bind to a locally available resource

Migration in Heterogeneous Systems

- ▶ Heterogeneous System: different OS and machine architecture
- ▶ If weak mobility
 - No runtime information (execution segment) needed to be transferred
 - Just compiler the source code to generate different code segments, one for each potential target machine
- ▶ But how execution segment is migrated at strong mobility?
 - Execution segment: data private to the process, stack, PC
 - Idea: avoid having execution depend on platform-specific data (like register values in the stack)

Migration in Heterogeneous Systems

Solutions:

- ▶ First, code migration is restricted to specific points
 - Only when a next subroutine is called
- ▶ Then, running system maintains its own copy of the stack, called migration stack, in a machine-independent way
 - Updated when call a subroutine or return from a subroutine
- ▶ Finally, when migrate
 - The global program-specific data are marshaled along with the migration stack are sent
 - The dest. load the code segment fit for its arch. and OS.
- ▶ It only works if compiler supports such as stack and a suitable runtime system

Distributed System



Unit:4

Unit 4. Communication

- 4.1 Foundations**
- 4.2 Remote Procedure Call**
- 4.3 Message-Oriented Communication**
- 4.4 Multicast Communication**
- 4.5 Case Study: Java RMI and Message Passing Interface (MPI)**

Introduction

- ▶ Interprocess communication is at the heart of all distributed systems.
- ▶ It makes no sense to study distributed systems without carefully examining the ways that processes on different machines can exchange information.
- ▶ Communication in distributed systems is always based on low-level message passing as offered by the underlying network.
- ▶ Expressing communication through message passing is harder than using primitives based on shared memory, as available for nondistributed platforms.
- ▶ Modern distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet.
- ▶ Unless the primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

Layered Protocols

- ▶ Due to the absence of shared memory, all communication in distributed systems is based on sending and receiving (low level) messages.
- ▶ When process A wants to communicate with process B, it first builds a message in its own address space.
- ▶ Then it executes a system call that causes the operating system to send the message over the network to B.
- ▶ International Standards Organization (ISO) reference model: Open Systems Interconnection (OSI) Reference Model (Day and Zimmerman, 1983).
- ▶ Protocols that were developed as part of the OSI model were never widely used.
- ▶ Underlying model useful for understanding computer networks.

- ▶ OSI model is designed to allow open systems to communicate.
- ▶ An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received.
- ▶ Rules are formalized into protocols.
- ▶ Groups of computers communicate over a network by agreeing on the protocols to be used.

- ▶ Two general types of protocols:
- ▶ 1. Connection oriented protocols
- ▶ Before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol to use.
- ▶ When done communicating, the connection is terminated.
- ▶ e.g. telephone is a connection-oriented communication system.
- ▶ 2. Connectionless protocols
- ▶ No advance setup.
- ▶ The sender transmits the first message when it is ready.
- ▶ e.g. Dropping a letter in a mailbox is an example of connectionless communication.

The OSI model

- ▶ Communication is divided up into seven levels or layers.
- ▶ Each layer deals with one specific aspect of the communication.
- ▶ Each layer provides an interface to the one above it.
- ▶ The interface consists of a set of operations that together define the service the layer is prepared to offer its users.



Layered Network Communication Protocols

Physical layer:

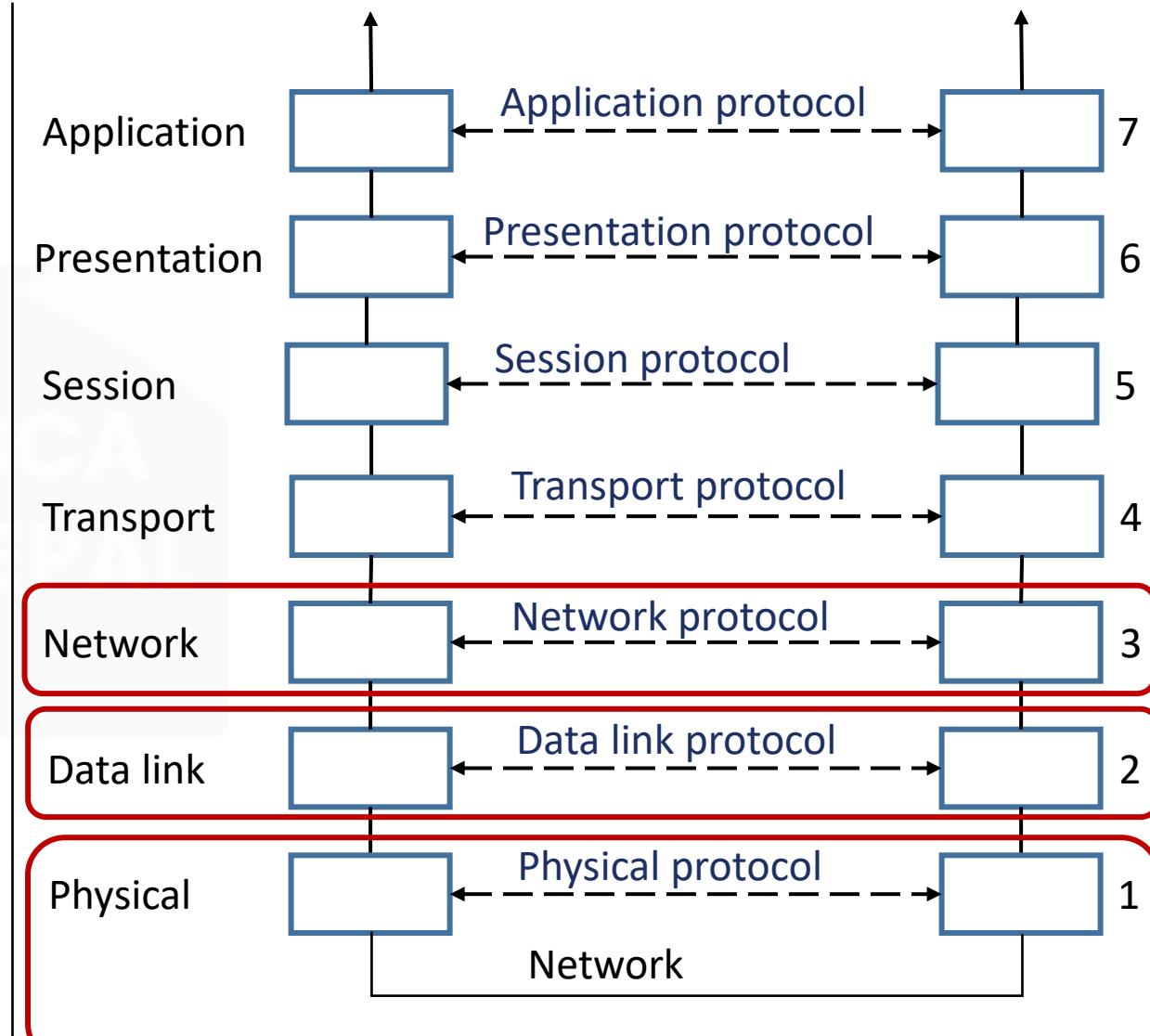
- ▶ Transmitting bits between sender and receiver
- ▶ Functions of a Physical layer: Line Configuration, Data Transmission, Topology, Signals

Data link layer:

- ▶ transmitting frames over a link, error detection and correction
- ▶ Functions of a Datalink layer: Framing, Flow Control, Error Control, Access Control

Network layer:

- ▶ Routing of packets between source host and destination host
 - IP – Internet's network layer protocol



Layered Network Communication Protocols

Transport layer:

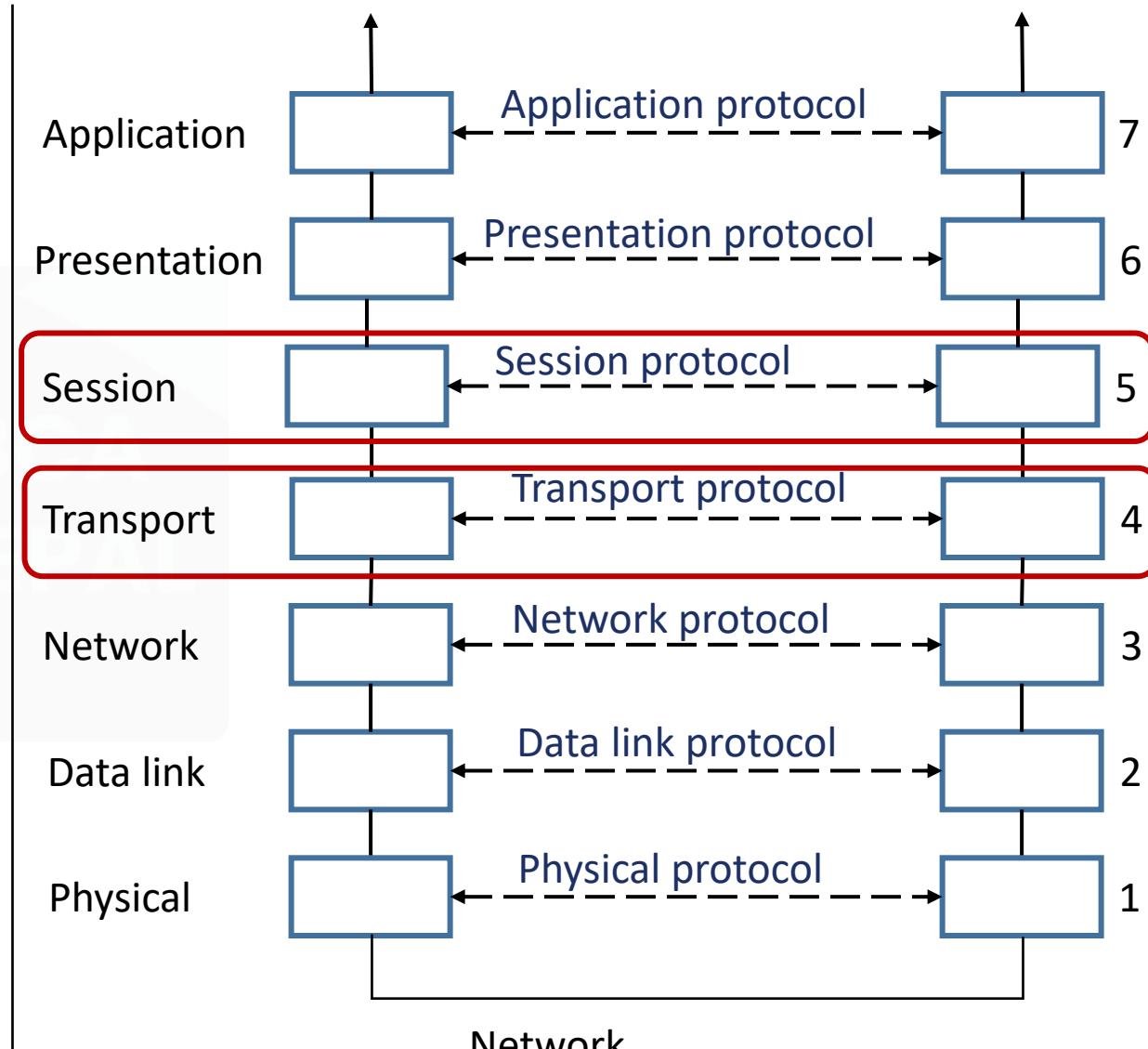
- ▶ Process-to-process communication
- ▶ **TCP** and **UDP** - Internet's transport layer protocols
 - **TCP**: Connection-oriented, reliable communication
 - **UDP**: Connectionless, unreliable communication

Reliable

Unreliable

Session layer:

- ▶ It establishes, manages, and terminates the connections between the local and remote application.
- ▶ Functions of Session layer: Dialog control, Synchronization



Layered Network Communication Protocols

Presentation layer:

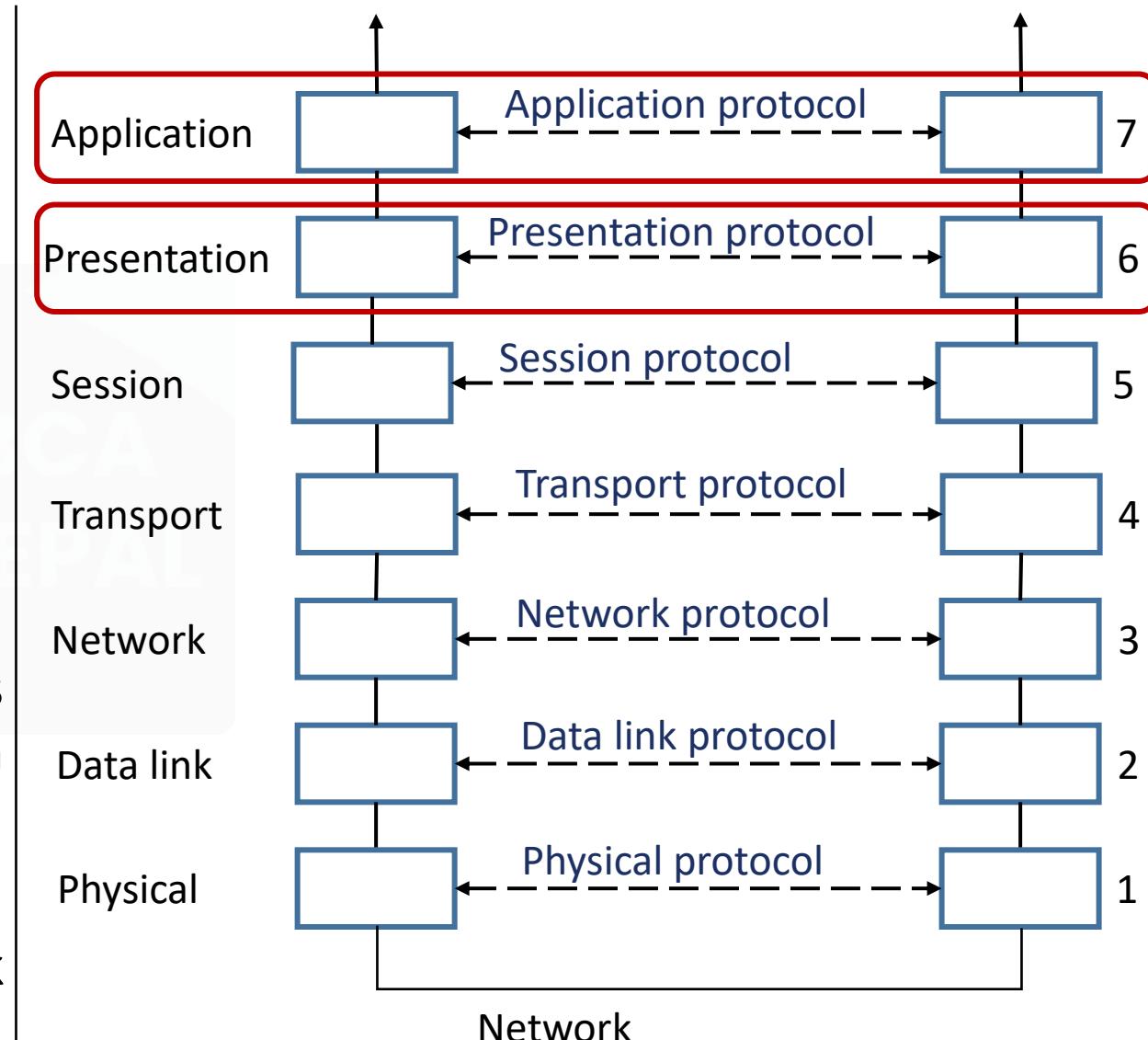
- ▶ Transforms data to provide a standard interface for the Application layer.

Functions of Presentation layer:

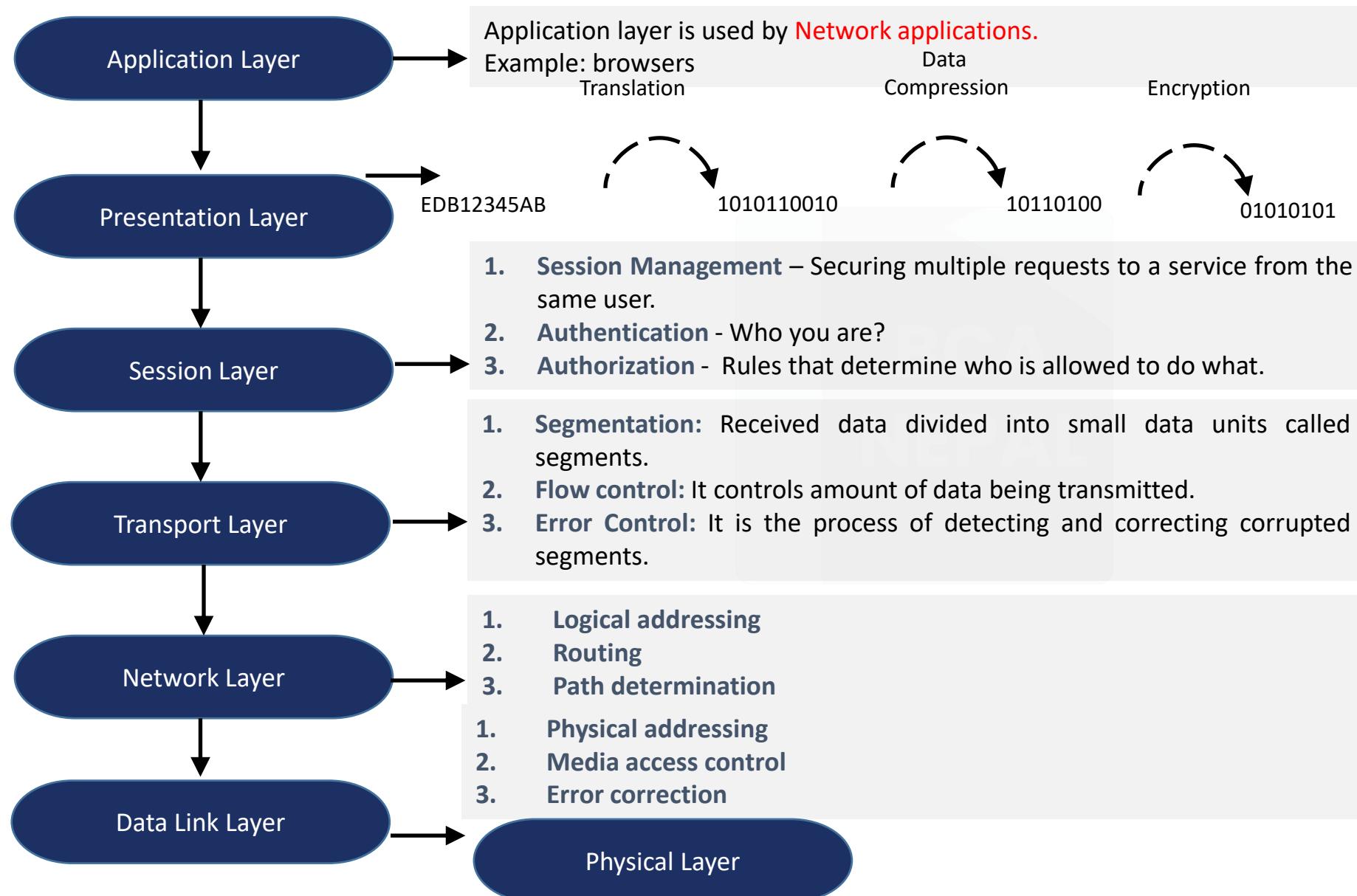
- MIME encoding
- Data compression
- Data encryption

Application layer:

- ▶ Provides means for the user to access information on the network through an application
- ▶ It serves as a window for users and application processes to access network service.



OSI Model

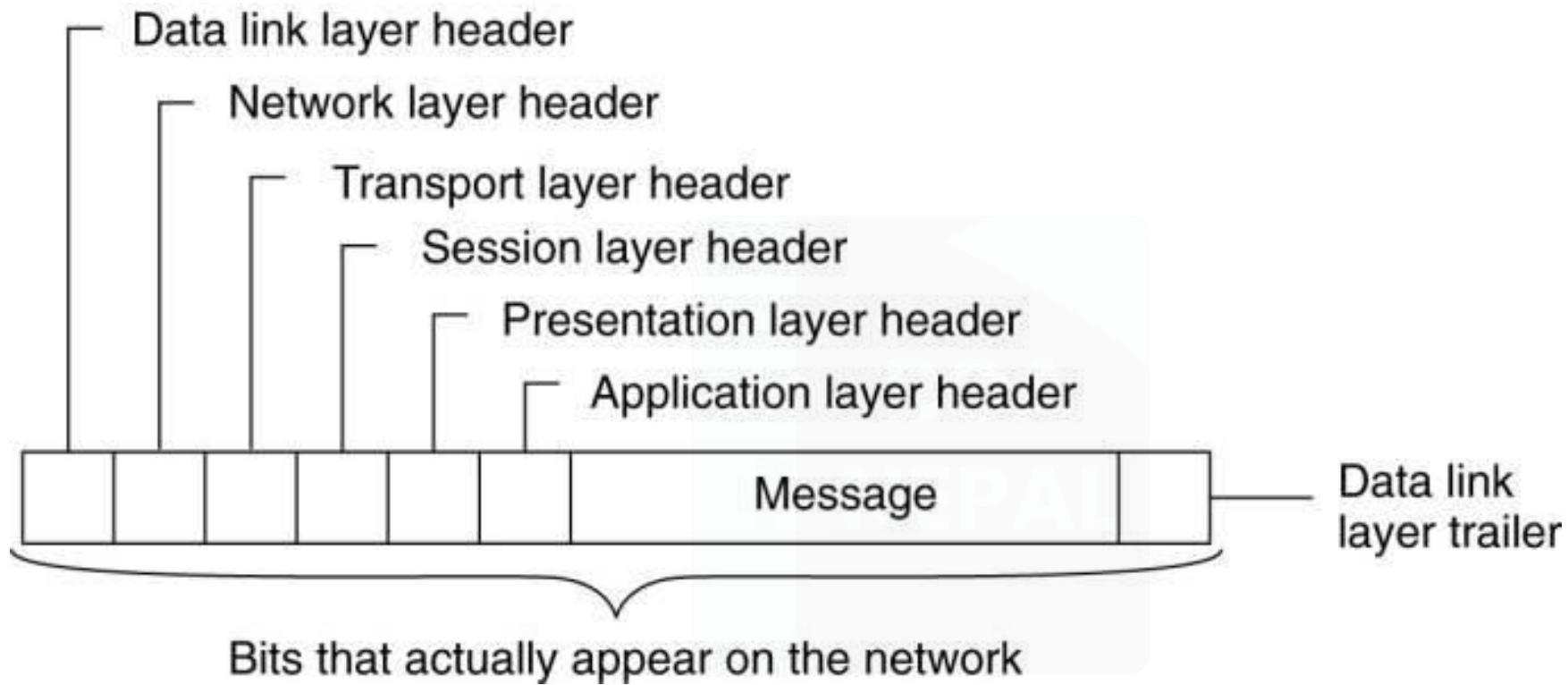


OSI Model (Layers & Activities)

OSI Layers	Activities
Application	To allow access to network resources.
Presentation	To translate, compress, and encrypt/decrypt data.
Session	To establish, manage, and terminate session.
Transport	To provide reliable process-to-process message delivery and error recovery.
Network	To move packets from source to destination; To provide internetworking.
Data Link	To organize bits into frames; To provide hop-to-hop delivery.
Physical	To transmit bits over a medium; To provide mechanical and electrical specifications.

- ▶ When process A on machine 1 wants to communicate with process B on machine 2, it builds a message and passes the message to the application layer on its machine.
- ▶ This layer might be a library procedure, for example, but it could also be implemented in some other way (e.g., inside the operating system, on an external network processor, etc.).
- ▶ The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer.
- ▶ The presentation layer in turn adds its own header and passes the result down to the session layer, and so on. Some layers add not only a header to the front, but also a trailer to the end.
- ▶ When it hits the bottom, the physical layer actually transmits the message (which by now might look as shown in Fig in next slide by putting it onto the physical transmission medium).

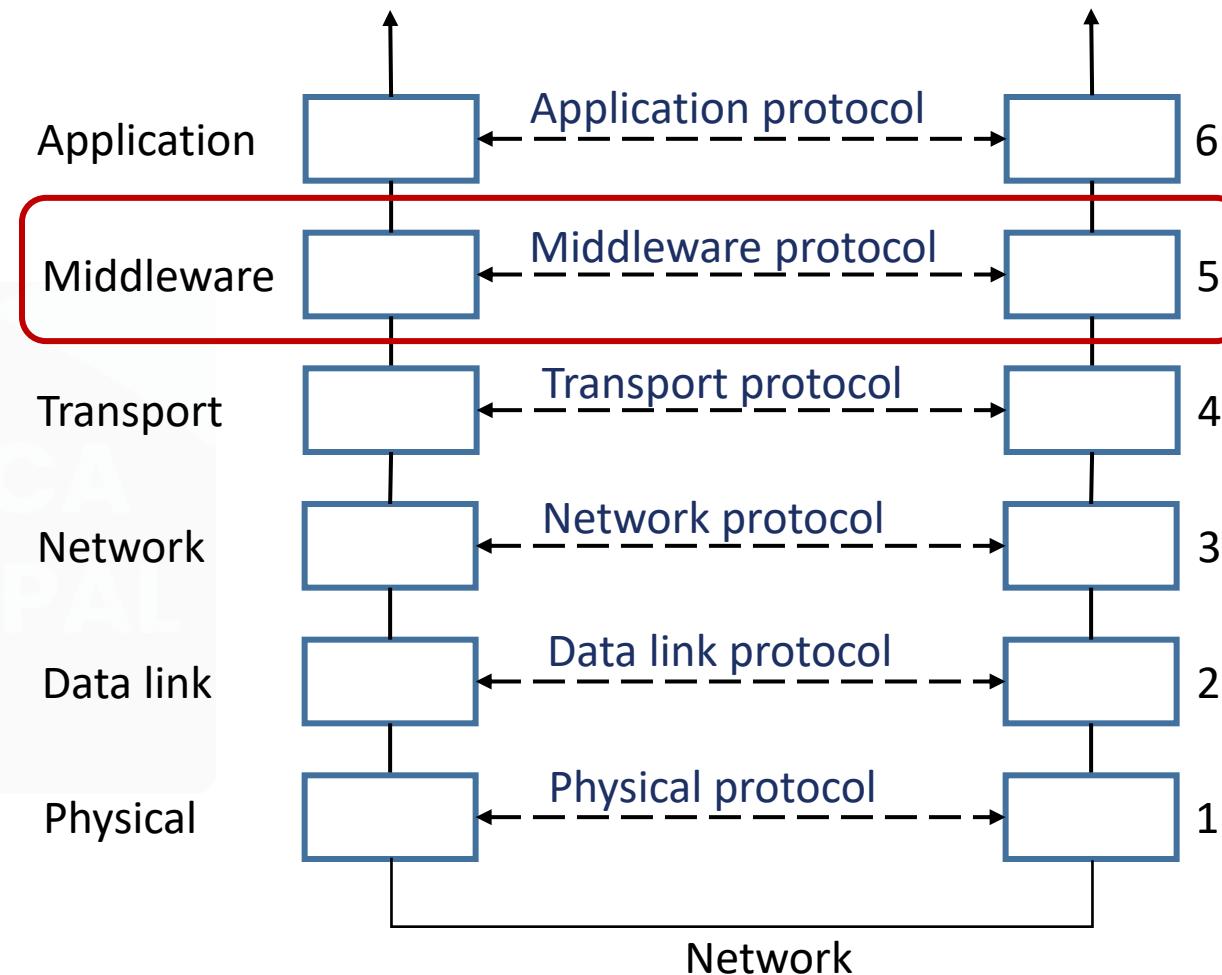
A typical message as it appears on the network



When the message arrives at machine 2, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver, process B, which may reply to it using the reverse path. The information in the layer n header is used for the layer n protocol.

Middleware Layer

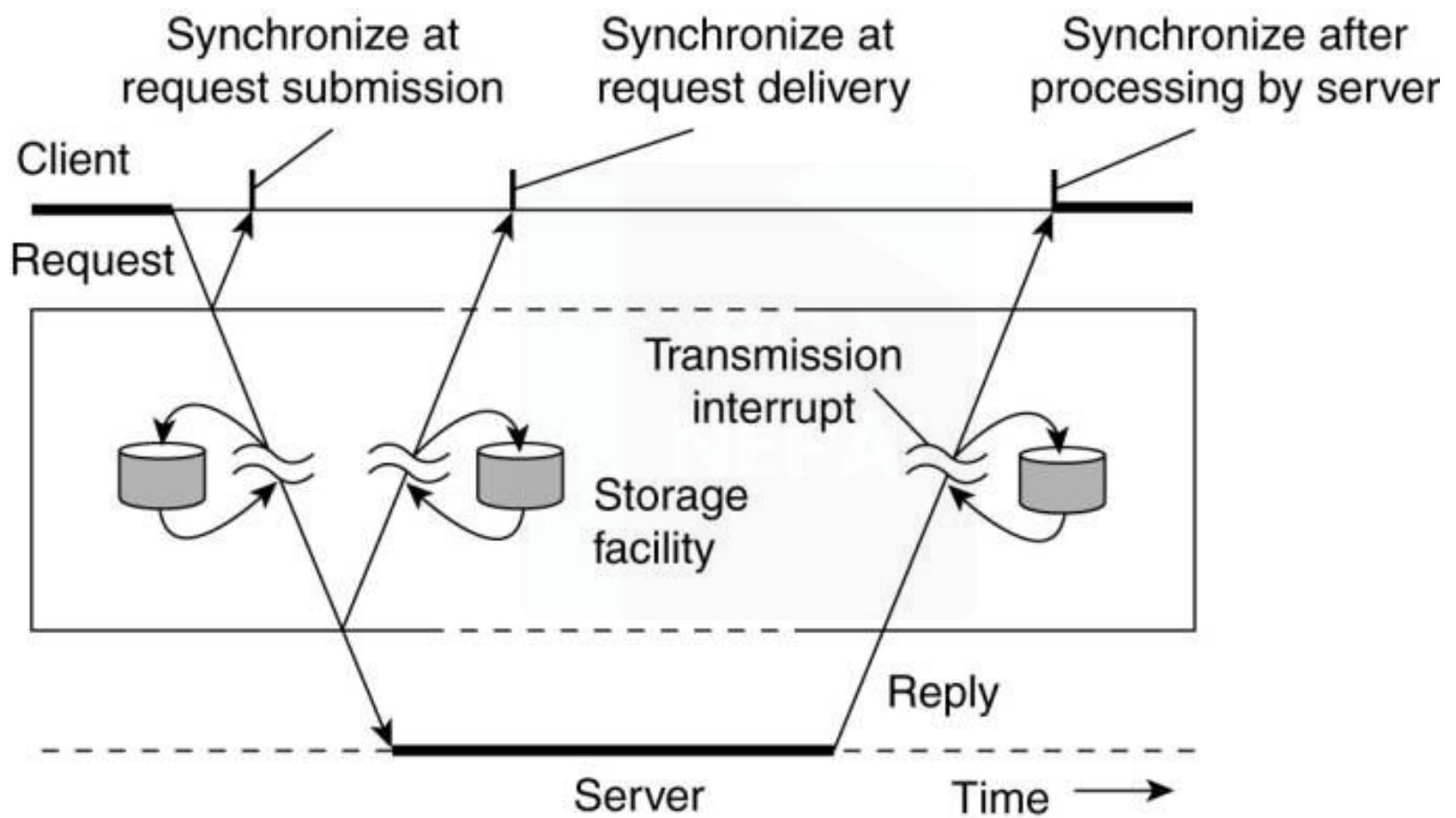
- ▶ Middleware is an application that logically lives (mostly) in the application layer, but contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications.
- ▶ **Middleware** provides common services and protocols that can be used by many different applications
 - High-level communication services, e.g., RPC, multicasting
 - Security protocols, e.g., authentication protocols, authorization protocols
 - Distributed locking protocols for mutual exclusion
 - Distributed commit protocols



Types of Communication

- ▶ View middleware as an additional service in client-server computing,
- ▶ Viewing middleware as an intermediate (distributed) service in application-level communication.
- ▶ Example: An electronic mail system.
- ▶ The core of the mail delivery system viewed as a middleware communication service.
- ▶ Each host runs a user agent allowing users to compose, send, and receive e-mail.
- ▶ A sending user agent passes such mail to the mail delivery system, expecting it, to deliver the mail to the intended recipient.
- ▶ The user agent at the receiver's side connects to the mail delivery system to see whether any mail has arrived.
- ▶ If so, the messages are transferred to the user agent so that they can be displayed and read by the user.

Viewing middleware as an intermediate (distributed) service in application-level communication.



Types of Communication

- ▶ Transient Vs. Persistent communication
- ▶ Synchronous Vs. Asynchronous communication



Transient Vs. Persistent communication

Transient Communication

- ▶ Here, Sender and receiver run at the same time based on request/response protocol, the message is expected otherwise it will be discarded.
- ▶ Middleware discards a message if it cannot be delivered to receiver immediately
- ▶ Messages exist only while the sender and receiver are running.
- ▶ Communication errors or inactive receiver cause the message to be discarded.
- ▶ Transport-level communication is transient

Persistent Communication

- ▶ Messages are stored by middleware until receiver can accept it
- ▶ Receiving application need not be executing when the message is submitted.
- ▶ Example: Email

Synchronous Vs Asynchronous Communication

Synchronous Communication

- ▶ Sender blocks until its request is known to be accepted
- ▶ Sender and receiver must be active at the same time
- ▶ Sender execution is continued only if the previous message is received and processed.

Asynchronous Communication

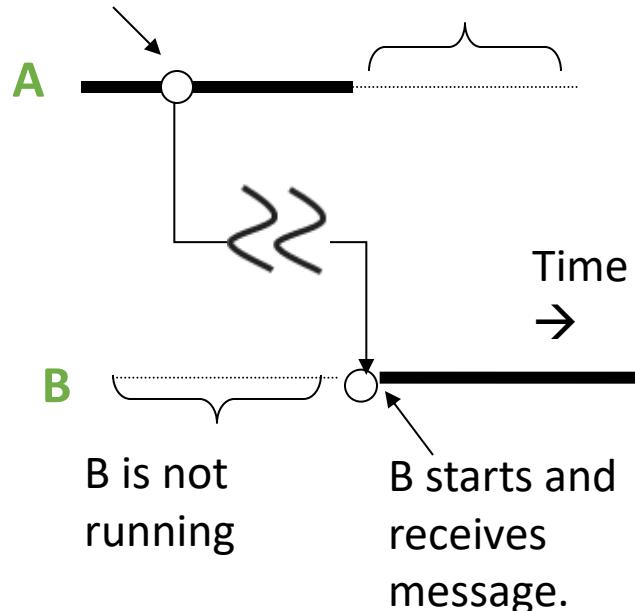
- ▶ Sender continues execution immediately after sending a message
- ▶ Message stored by middleware upon submission
- ▶ Message may be processed later at receiver's convenience

Distributed Communications Classifications

1. Persistent asynchronous communication.
2. Persistent synchronous communication.
3. Transient asynchronous communication.
4. Transient synchronous communication.
 - Receipt-based transient synchronous communication.
 - Delivery-based transient synchronous communication at message delivery.
 - Response-based transient synchronous communication.

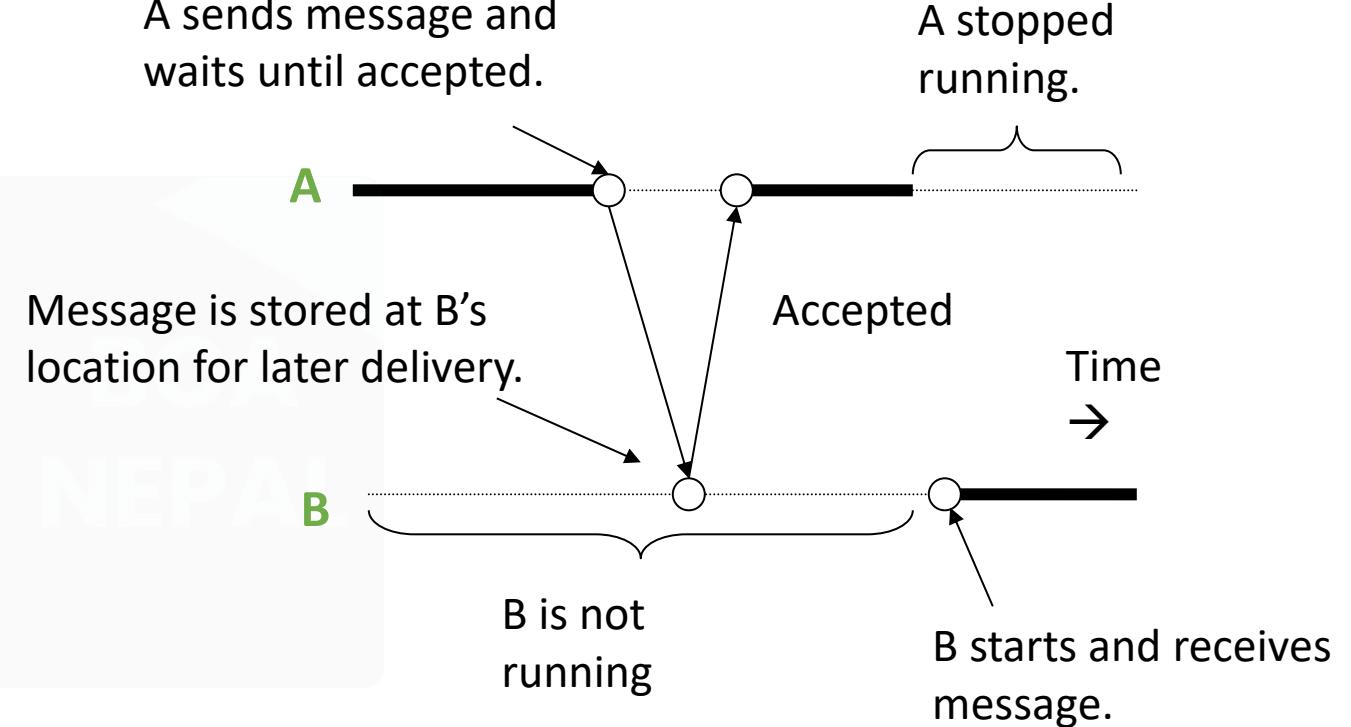
Distributed Communications Classifications

A sends message and continues.



Persistent asynchronous communication

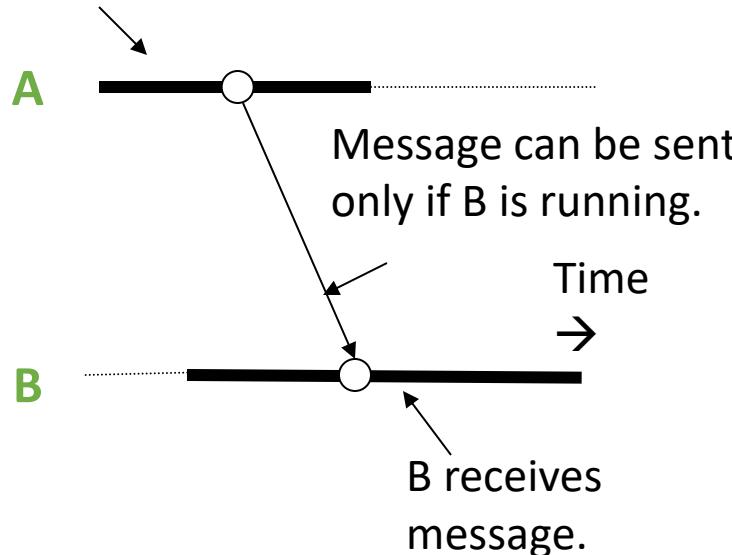
A sends message and waits until accepted.



Persistent synchronous communication

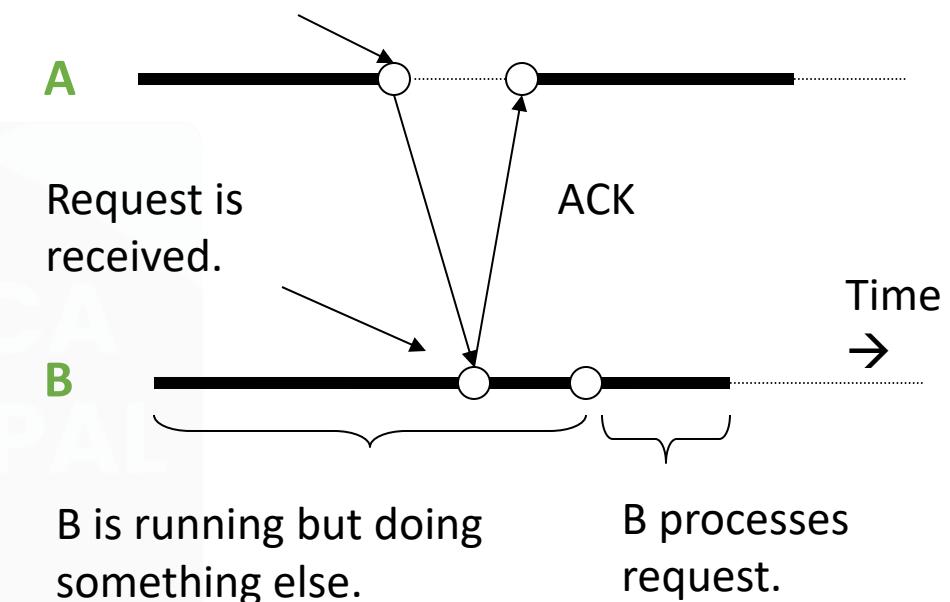
Distributed Communications Classifications

A sends message
and continues.



**Transient asynchronous
communication**

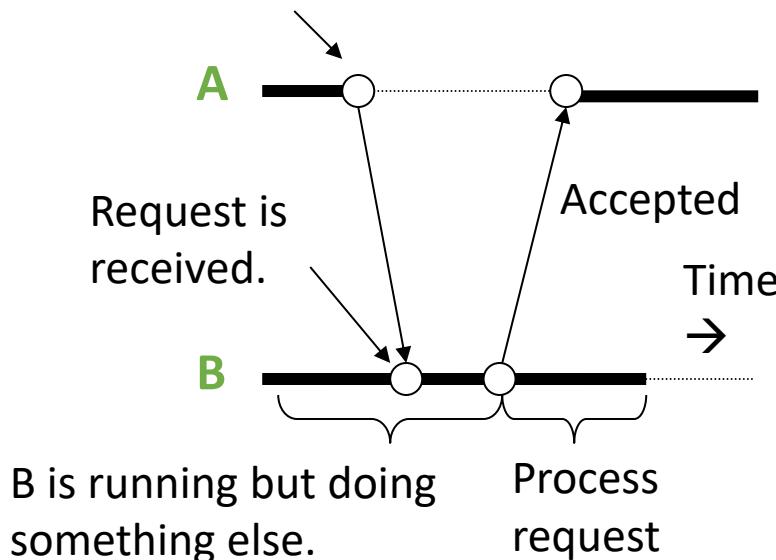
A sends message and
waits until received.



**Receipt-based synchronous
communication**

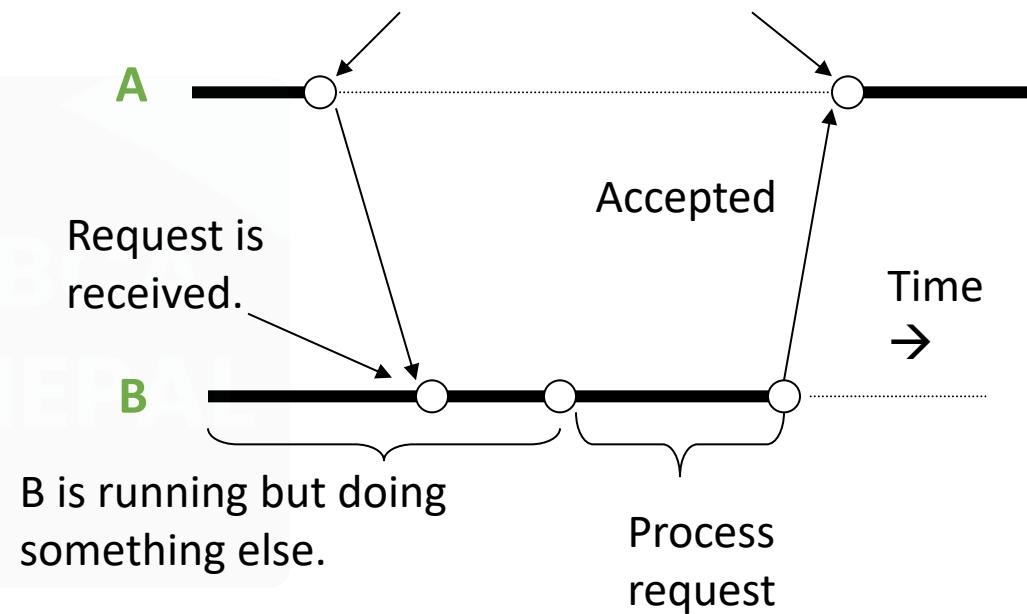
Distributed Communications Classifications

A sends request and waits until accepted.



Delivery-based synchronous communication

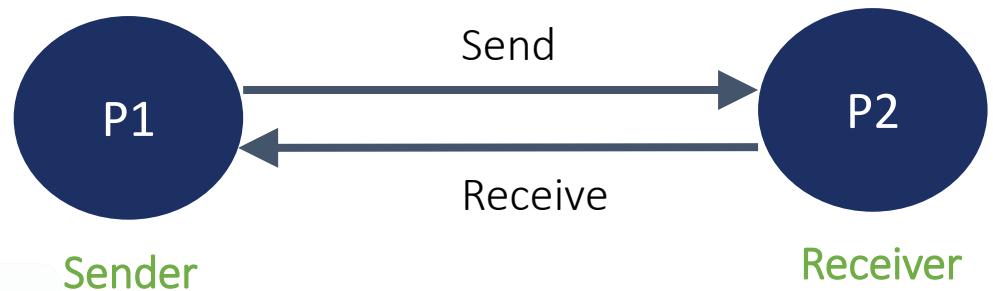
A sends request and waits for reply.



Response-based synchronous communication

Message Passing

- ▶ It refers to means of communication between
 - Different thread with in a process .
 - Different processes running on same node.
 - Different processes running on different node.
- ▶ In this mode, a sender or a source process send a message to a non receiver or destination process.
- ▶ Message has a predefined structure and message passing uses two system call: Send and Receive
 - `send(name of destination process, message)`
 - `receive(name of source process, message)`
- ▶ Mode of communication between two process can take place through two methods
 1. Direct Addressing
 2. Indirect Addressing



Client Server Model

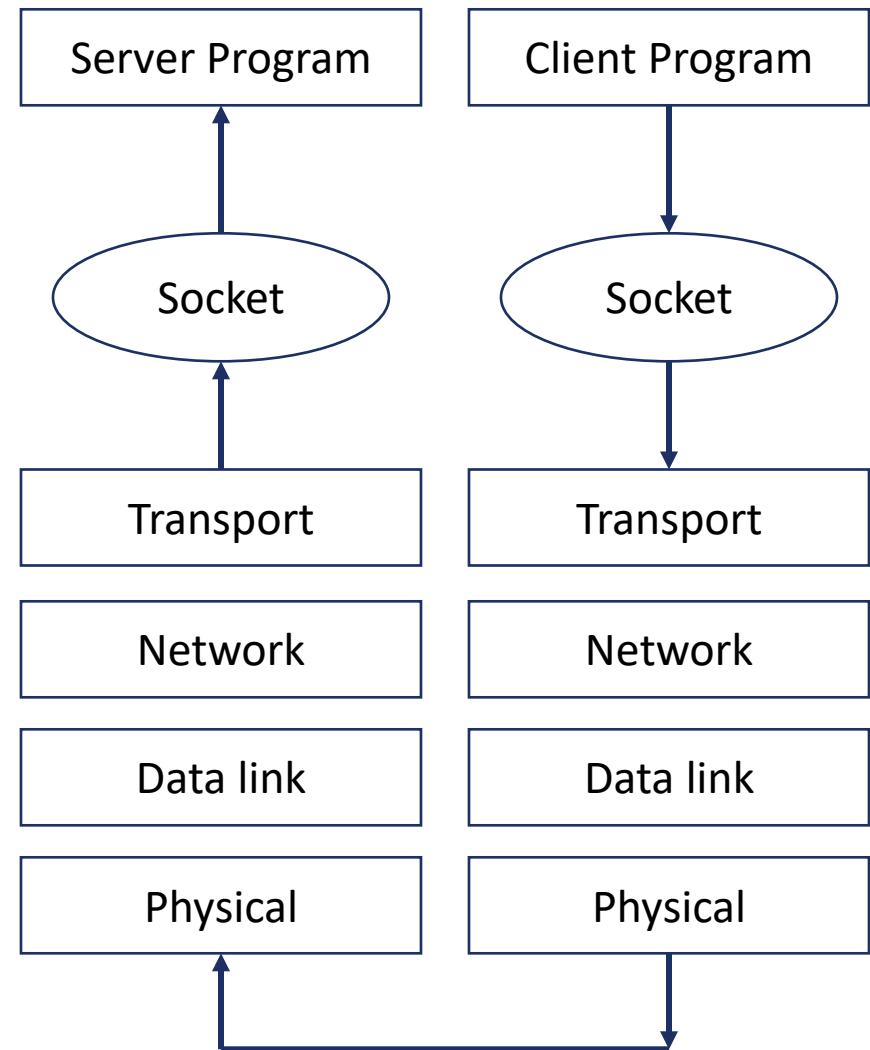
- ▶ Following are different types of packets transmitted across the network.
 - ➔ **REQ:** Request packet is used to **send the request** from the client to the server.
 - ➔ **Reply:** This message is used to **carry the result** from the server to the client.
 - ➔ **ACK:** Acknowledgement packet is used to send the **correct receipt** of the packet to the sender.
 - ➔ **Are You Alive (AYA)?:** This packet is sent in case the server takes a long time to complete the client's request.
 - ➔ **I am Alive (IAA):** The server, if active, replies with this packet.

Client Server Model Interaction

- ▶ Two processes in client-server model can interact in various ways:
 - ▶ Sockets
 - ▶ Remote Procedure Calls (RPC)

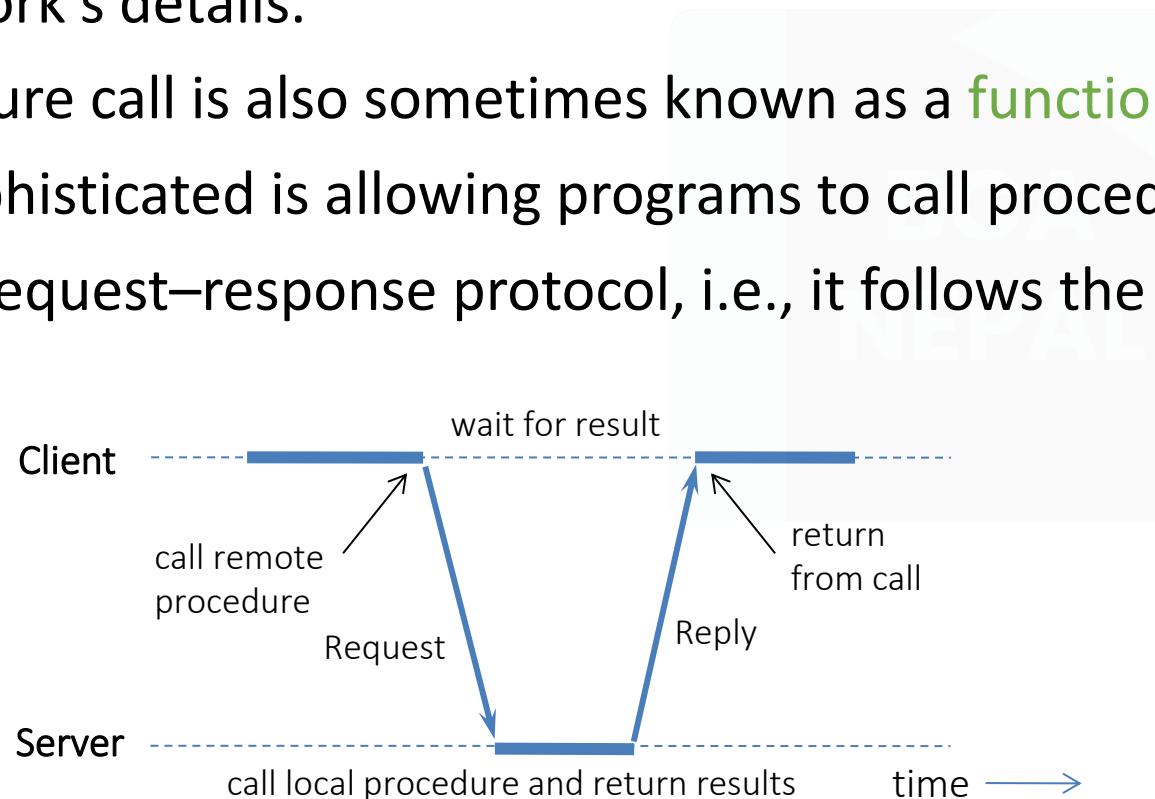
Socket

- ▶ The process acting as server opens a socket using a well-known port and waits until some client request comes.
- ▶ The second process acting as a client also opens a socket but instead of waiting for an incoming request, the client processes 'requests first'.



Remote Procedure Call (RPC)

- ▶ Low level message passing is based on send and receive primitives.
- ▶ Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.
- ▶ A procedure call is also sometimes known as a **function call** or a **subroutine call**.
- ▶ More sophisticated is allowing programs to call procedures located on other machines.
- ▶ RPC is a request–response protocol, i.e., it follows the **client-server model**



RPC Model

- ▶ It is similar to commonly used procedure call model. It works in the following manner:
 1. For making a procedure call, the **caller places arguments** to the procedure in some well specified location.
 2. **Control is then transferred** to the sequence of instructions that constitutes the body of the procedure.
 3. The **procedure body is executed** in a newly created execution environment that includes copies of the arguments given in the calling instruction.
 4. After the procedure execution is over, **control returns** to the calling point, returning a result.

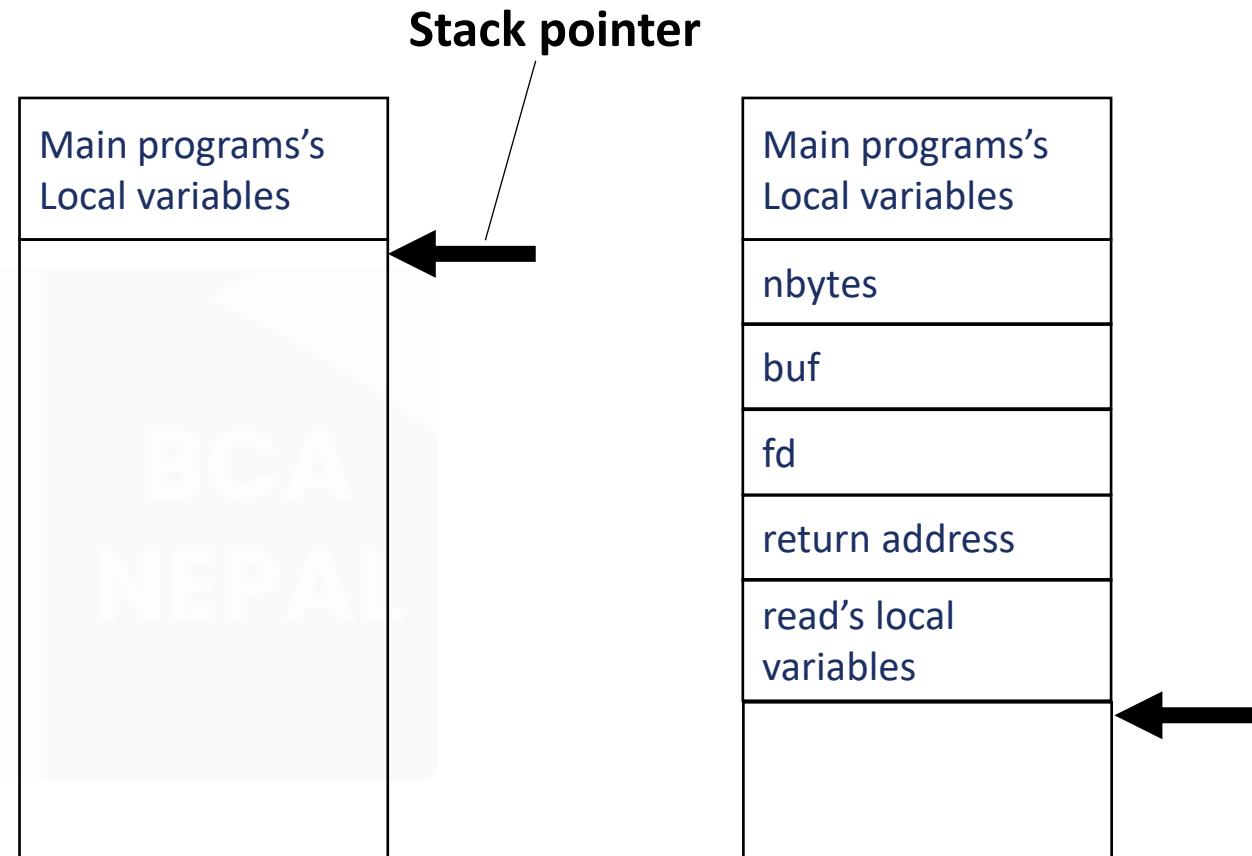
Conventional Procedure Call

- ▶ Consider a call in C like

count = read(fd, buf, nbytes);

- ▶ where

- ▶ **fd** is an integer indicating a file
- ▶ **buf** is an array of characters into which data are read
- ▶ **nbytes** is another integer telling how many bytes to read



The stack before the call

The stack while the called procedure is active

Functions of RPC Elements

The Client

- ▶ It is user process which initiates a remote procedure call
- ▶ The client makes a perfectly normal call that invokes a corresponding procedure in the client stub.

The Client stub

- ▶ On receipt of a request, it packs a requirements into a message and asks the local RPCRuntime to send it to the server stub.
- ▶ On receipt of a result it unpacks the result and passes it to client.

Functions of RPC Elements

RPCRuntime

- ▶ It handles transmission of messages between client and server.
- ▶ It is responsible for
 - Retransmission
 - Acknowledgement
 - Routing and Encryption



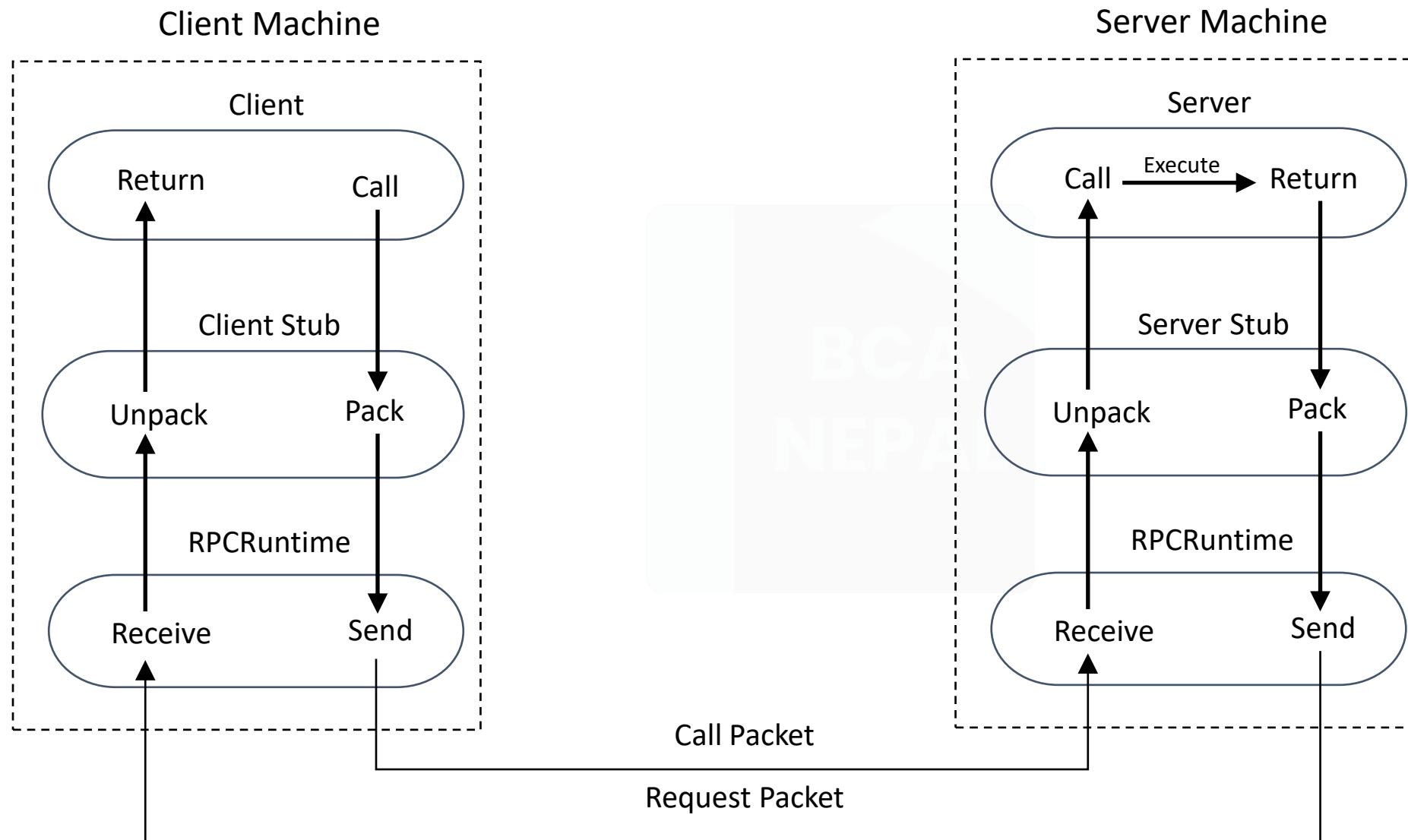
The Server stub

- ▶ It unpacks a call request and make a perfectly normal call to invoke the appropriate procedure in the server.
- ▶ On receipt of a result of procedure execution it packs the result and asks to RPCRuntime to send.

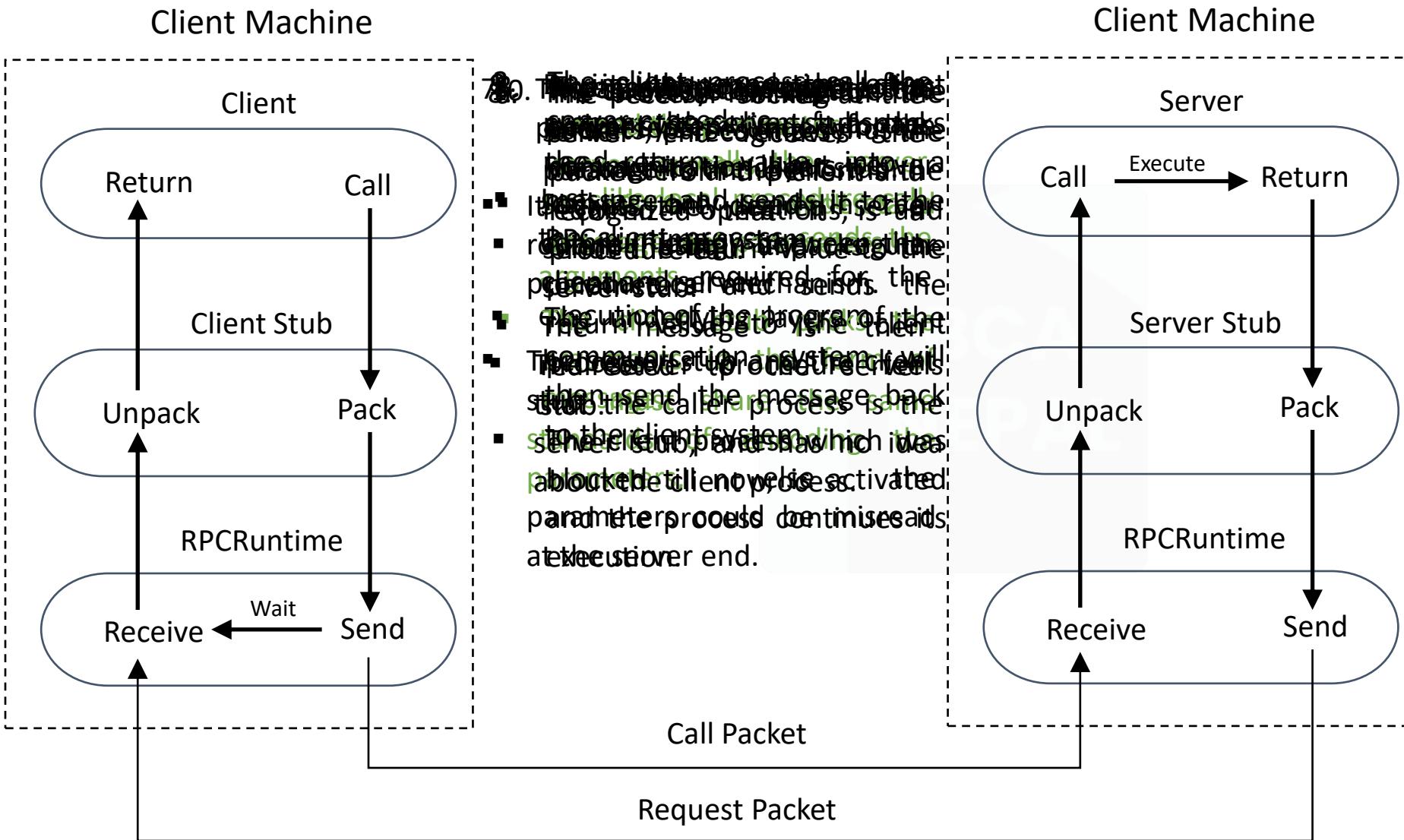
The Server

- ▶ It executes a appropriate procedure and returns the result from a server stub.

RPC Mechanism

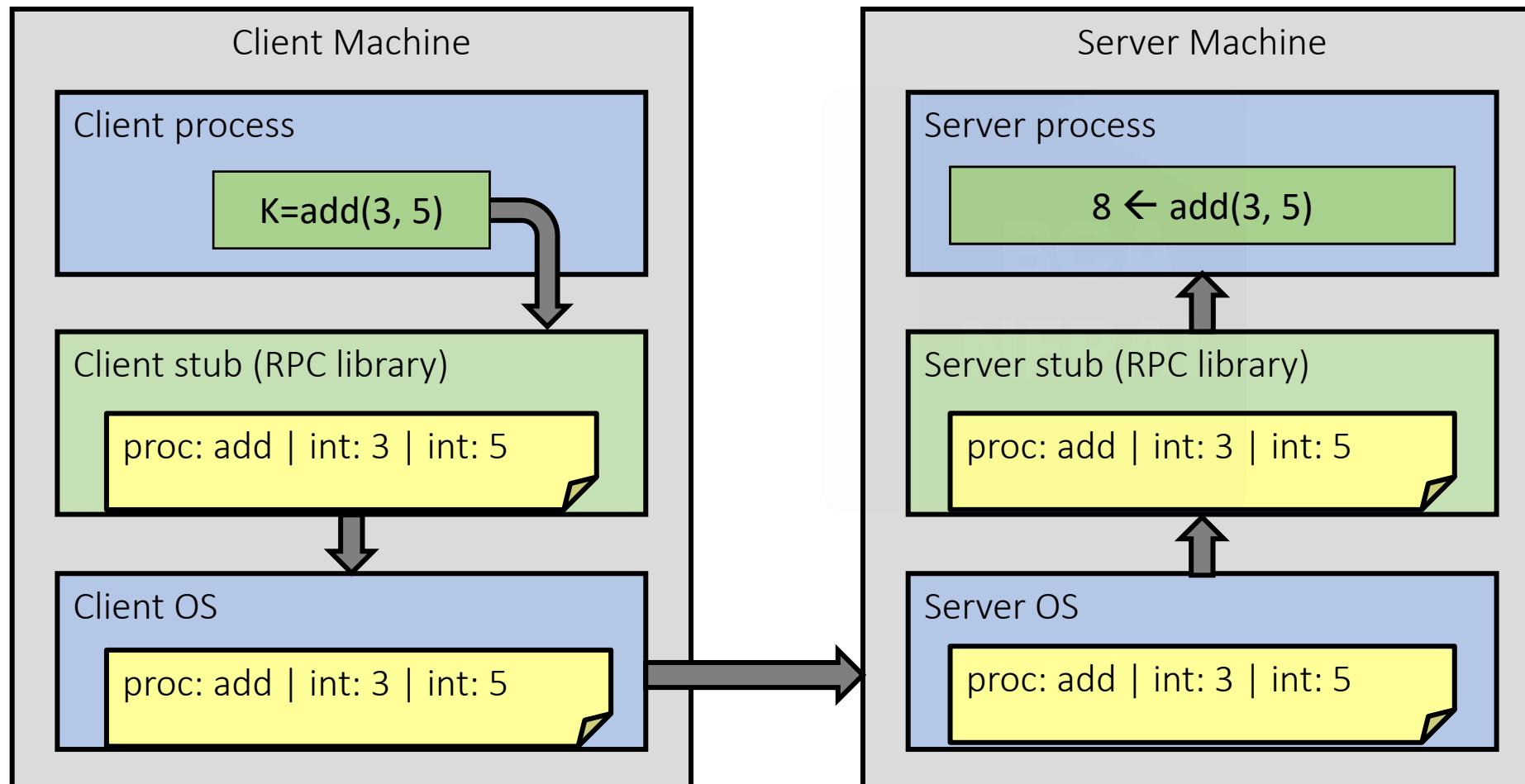


RPC Mechanism



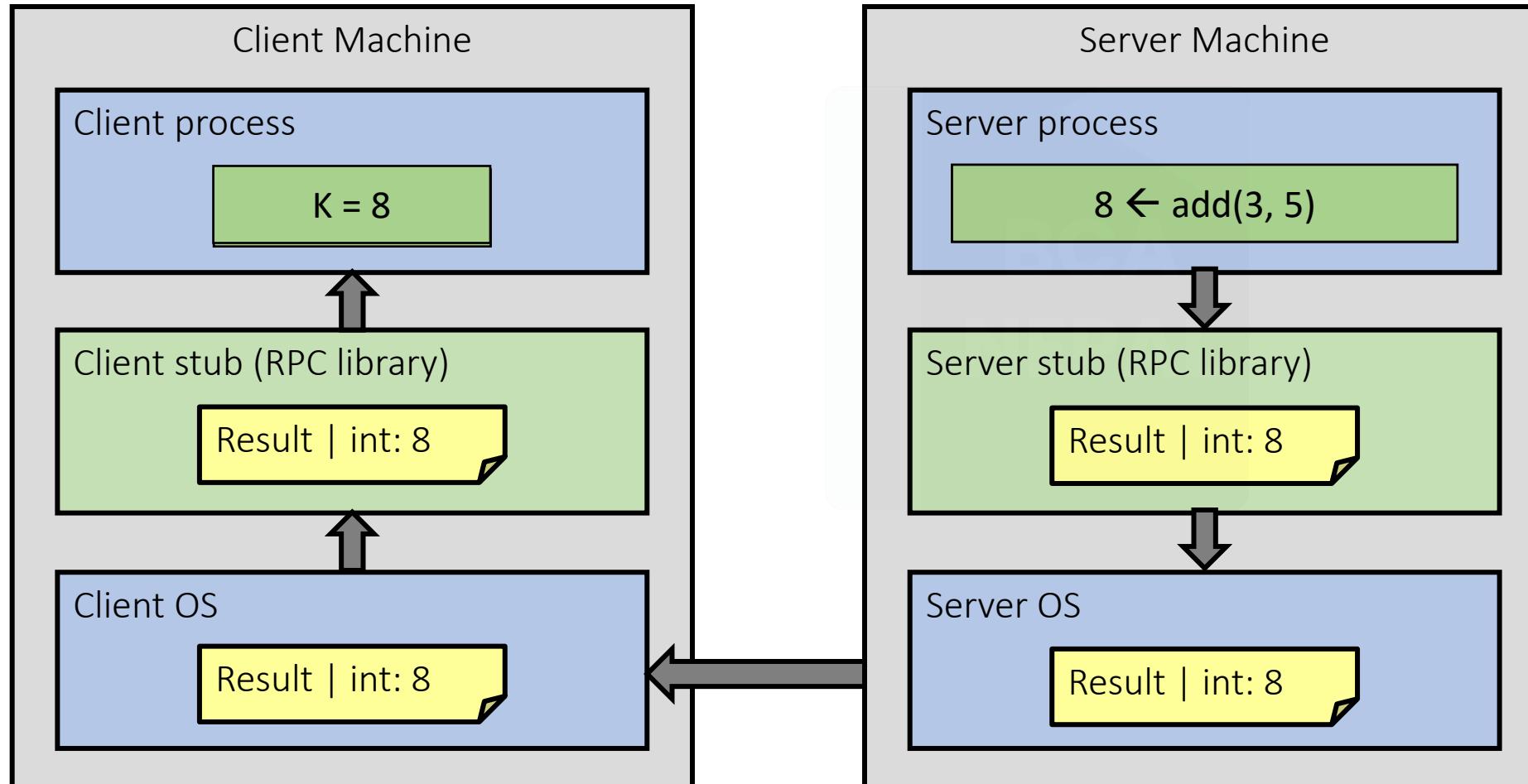
RPC Mechanism

3. Client sends request to server.



RPC Mechanism

Q. ~~Server's OS sends message to Client's OS.~~



Steps in a Remote Procedure Call

1. The client calls a local procedure, called the client stub.
2. Network messages are sent by the client stub to the remote system (via a system call to the local kernel using sockets interfaces).
3. Network messages are transferred by the kernel to the remote system via some protocol (either connectionless or connection-oriented).
4. A server stub, sometimes called the skeleton, receives the messages on the server. It unmarshals the arguments from the messages.
5. The server stub calls the server function, passing it the arguments that it received from the client.
6. When server function is finished, it returns to the server stub with its return values.
7. The server stub converts the return values, if necessary, and marshals them into one or more network messages to send to the client stub.

Steps in a Remote Procedure Call

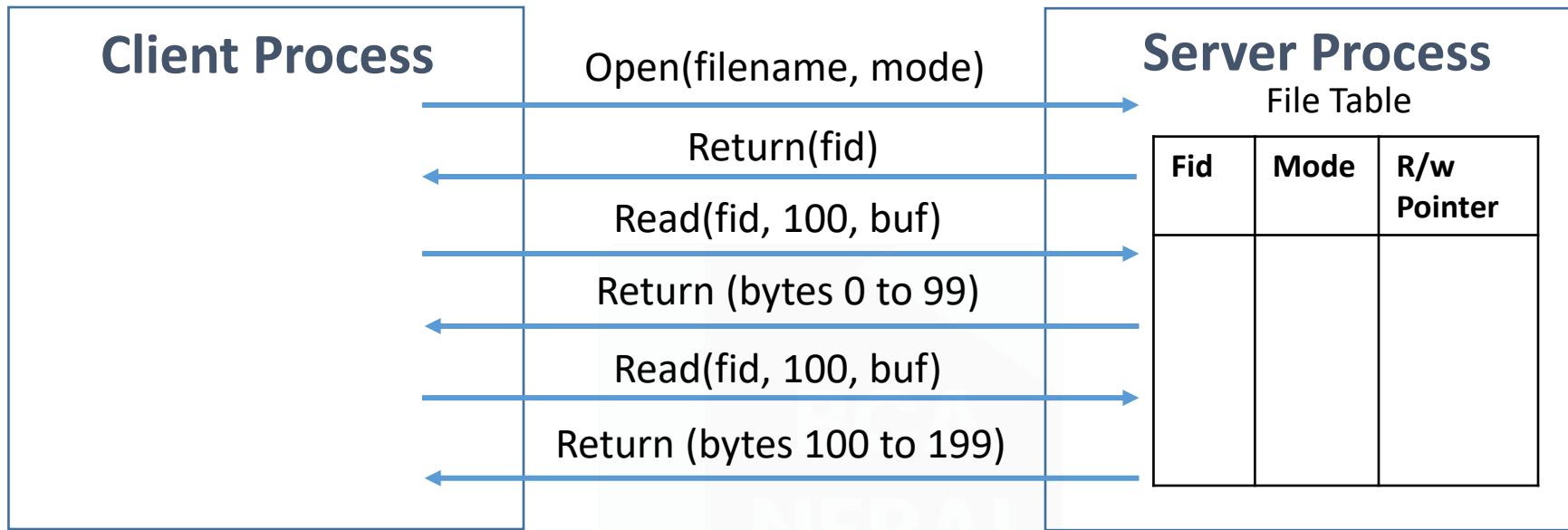
8. Messages get sent back across the network to the client stub.
9. The client stub reads the messages from the local kernel.
10. The client stub then returns the results to the client function, converting them from the network representation to a local one if necessary.

Server Management

Stateful File Servers

- ▶ A stateful server **maintains client's state information** from one remote procedure call to the next.
- ▶ These clients state information is subsequently used at the time of executing the second call.
- ▶ To illustrate how a stateful file server works, let us consider a file server for byte-stream files that allows the following operations on files:
 - Open(filename, mode)
 - Read(fid, n, buffer)
 - Write(fid, n, buffer)
 - Seek(fid, position)
 - Close(fid)

Stateful File Server

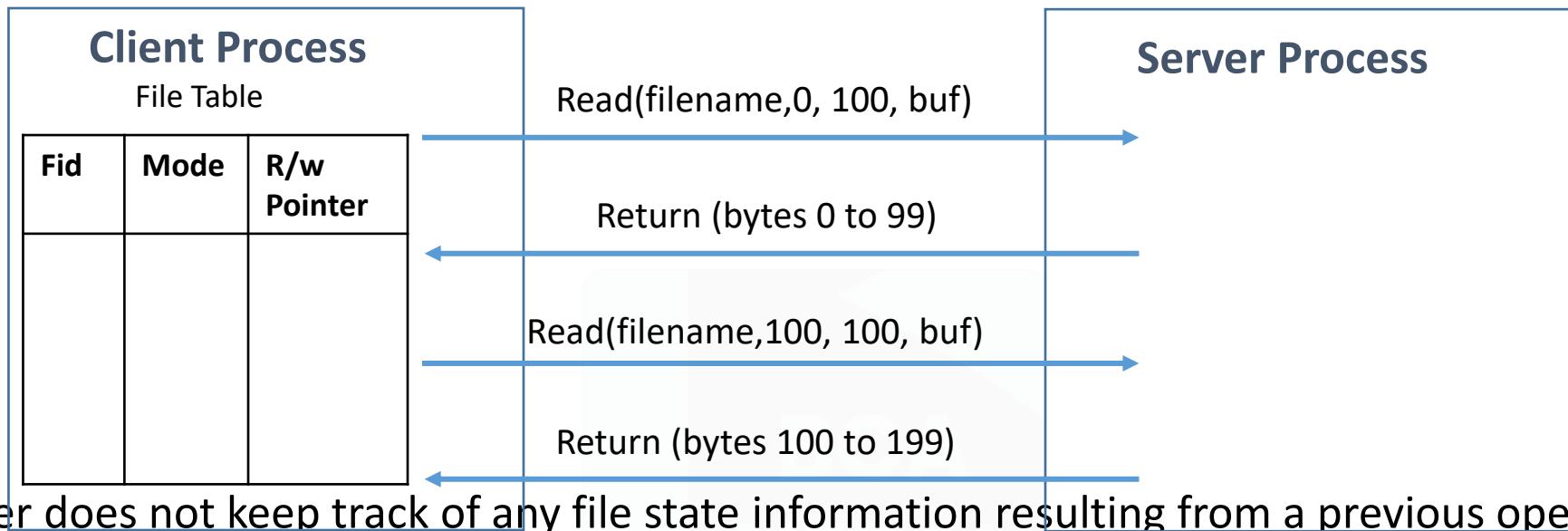


- After opening a file, if a client makes two subsequent Read (fid, 100, buf) requests, for the first request the first 100 bytes (bytes 0 to 99) will be read and for the second request the next 100 bytes (bytes 100 to 199) will be read.

Stateless File Server

- ▶ A stateless file server **does not maintain any client state information.**
- ▶ Therefore every request from a client must be accompanied with all the necessary parameters to successfully carry out the desired operation.
- ▶ Each request identifies the file and the position in the file for the read/write access.
- ▶ Operations on files in Stateless File server:
 - *Read(filename, position, n, buffer)*: On execution, the server returns n bytes of data of the file identified by filename.
 - *Write(filename, position, n, buffer)*: On execution, it takes n bytes of data from the specified buffer and writes it into the file identified by filename.

Stateless File Server



- ▶ This file server does not keep track of any file state information resulting from a previous operation.
- ▶ Therefore, if a client wishes to have similar effect as previous figure, the following two read operations must be carried out:
 - Read(filename, 0, 100, buffer)
 - Read(filename, 100, 100, buffer)

Difference between Stateful & Stateless

Parameters	Stateful	Stateless
State	A Stateful server remembers client data (state) from one request to the next.	A Stateless server does not remember state information.
Programming	Stateful server is harder to code.	Stateless server is straightforward to code.
Efficiency	More because clients do not have to provide full file information every time they perform an operation.	Less because information needs to be provided.
Crash recovery	Difficult due to loss of information.	Can easily recover from failure because there is no state that must be restored.
Information transfer	The client can send less data with each request.	The client must specify complete file names in each request.
Operations	Open, Read, Write, Seek, Close	Read, Write

Passing Value Parameters

- ▶ Packing parameters into a message is called **parameter marshaling**.
- ▶ Client and server machines may have different data representations (think of byte ordering)
- ▶ Wrapping a parameter means transforming a value into a sequence of bytes
- ▶ Client and server have to agree on the same encoding:
 - How are basic data values represented (integers, floats, characters)
 - How are complex data values represented (arrays, unions)
- ▶ Client and server need to properly interpret messages, transforming them into machine-dependent representations.
- ▶ Possible problems:
 - IBM mainframes use EBCDIC char code and IBM PC uses ASCII code
 - Integer: one's complement and 2's complement
 - Little endian and big endian

Passing Value Parameters

	3	2	1	0
0	0	0	5	
L	L	I	J	

Original message on
the Pentium

0	1	2	3
5	0	0	0
4	5	6	7

J I L L

The message after
receipt on the SPARC

0	1	2	3
0	0	0	5
4	5	6	7

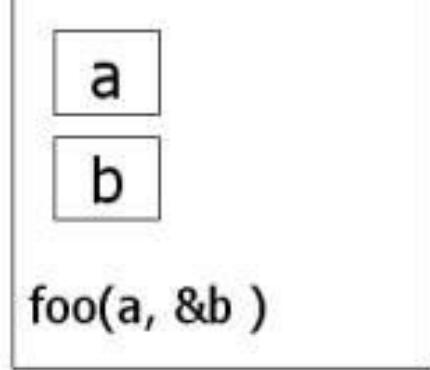
L L I J

The message after
being inverted

Passing Reference Parameters

- ▶ How are pointers, or in general, references passed?
- ▶ Only with the greatest of difficulty, if at all.
- ▶ A pointer is meaningful only within the address space of the process in which it is being used
- ▶ **Solution: copy/restore**
- ▶ Copy the array into the message and send it to the server.
- ▶ The server stub can then call the server with a pointer to this array, even though this pointer has a different address than the second parameter of read.
- ▶ Changes the server makes using the pointer (e.g., storing data into it) directly affect the message buffer inside the server stub.
- ▶ When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client.
- ▶ In effect, call-by-reference has been replaced by copy/restore.

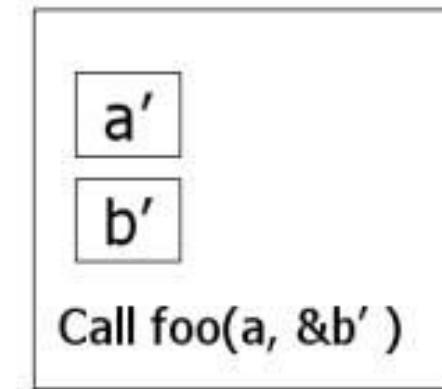
Machine A



Copy value a and contents of loc b
into a' and loc b'

Return Copy contents of loc b' into b

Machine B



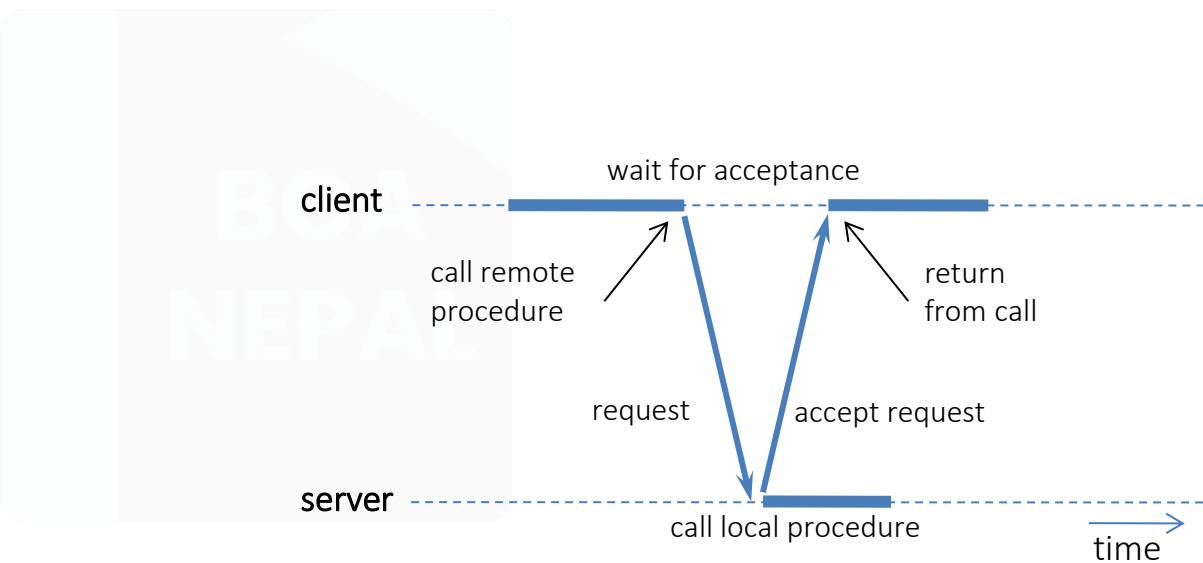
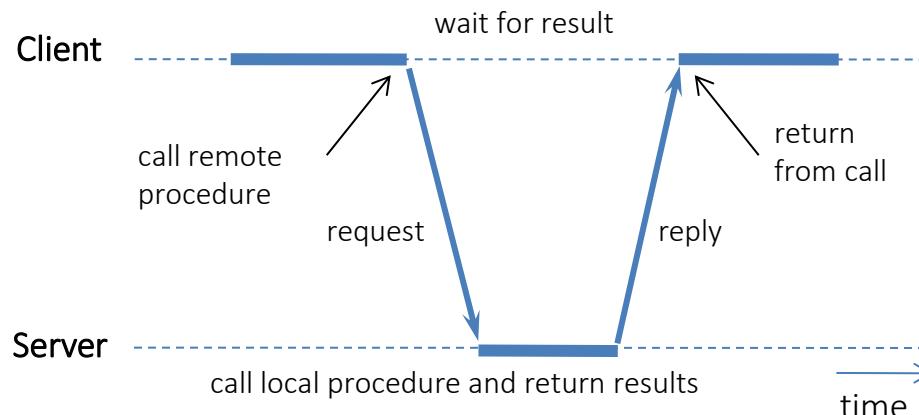
Asynchronous RPC

- ▶ A shortcoming of the original model: no need of blocking for the client in some cases.
- ▶ There are two cases
 1. If there is no result to be returned
 2. If the result can be collected later

Asynchronous RPC

1. If there is no result to be returned

- e.g., inserting records in a database, ...
- The server immediately sends an ack promising that it will carryout the request



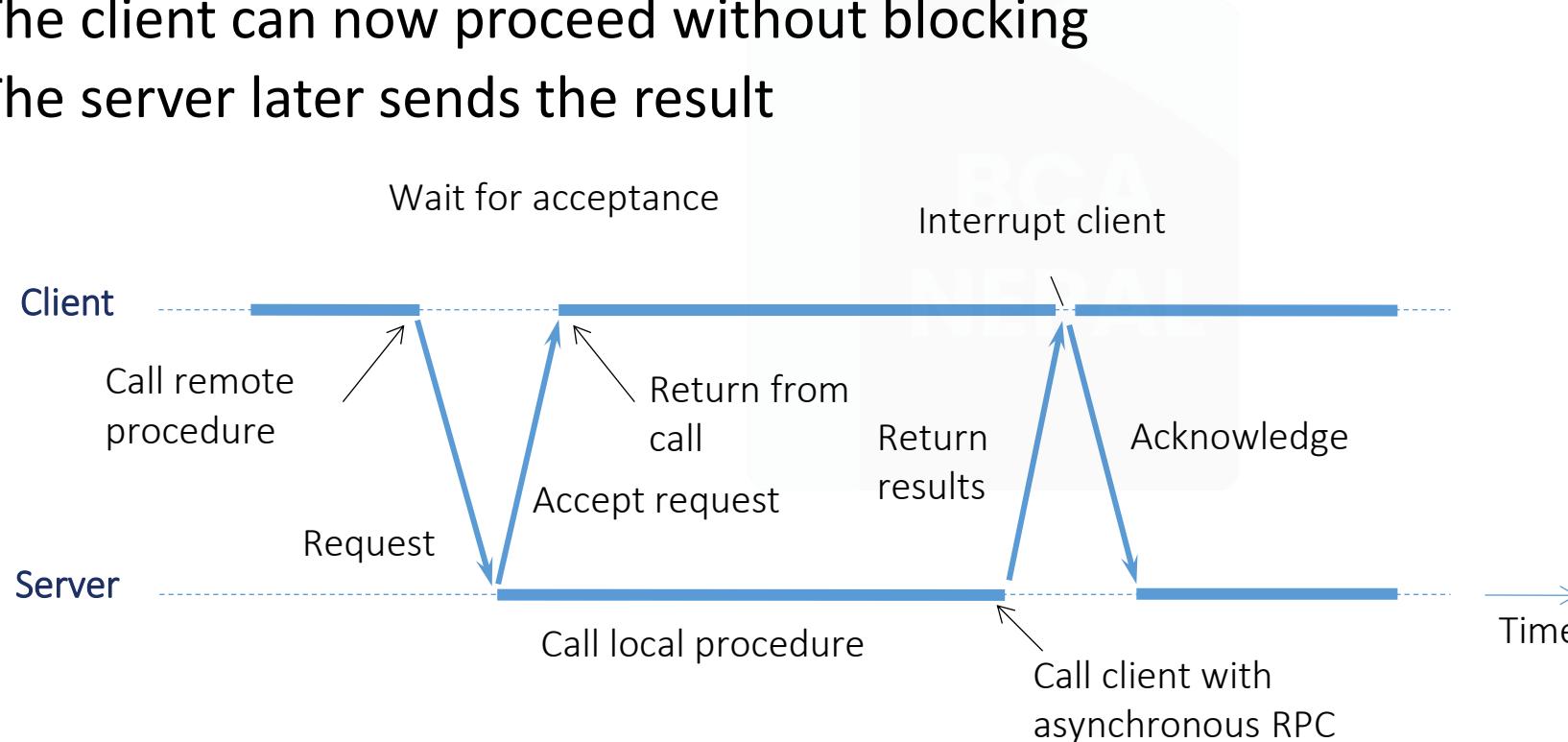
The interconnection between client and server in a traditional RPC

The interaction using **asynchronous RPC**

Asynchronous RPC

1. If the result can be collected later

- Example: prefetching network addresses of a set of hosts, ...
- The server immediately sends an ACK promising that it will carryout the request
- The client can now proceed without blocking
- The server later sends the result



A client and server interacting through two asynchronous RPCs

Message-oriented Transient Communication

- ▶ Messages are sent through a channel abstraction
- ▶ The channel connects two running processes
- ▶ Time coupling between sender and receiver
- ▶ Transmission time is measured in terms of milliseconds, typically
- ▶ Examples
 - Berkeley Sockets — typical in TCP/IP-based networks
 - MPI (Message-Passing Interface) — typical in high-speed interconnection networks among parallel processes
- ▶ **Socket** - communication end point to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read.

Analogy to Telephony Network Communication

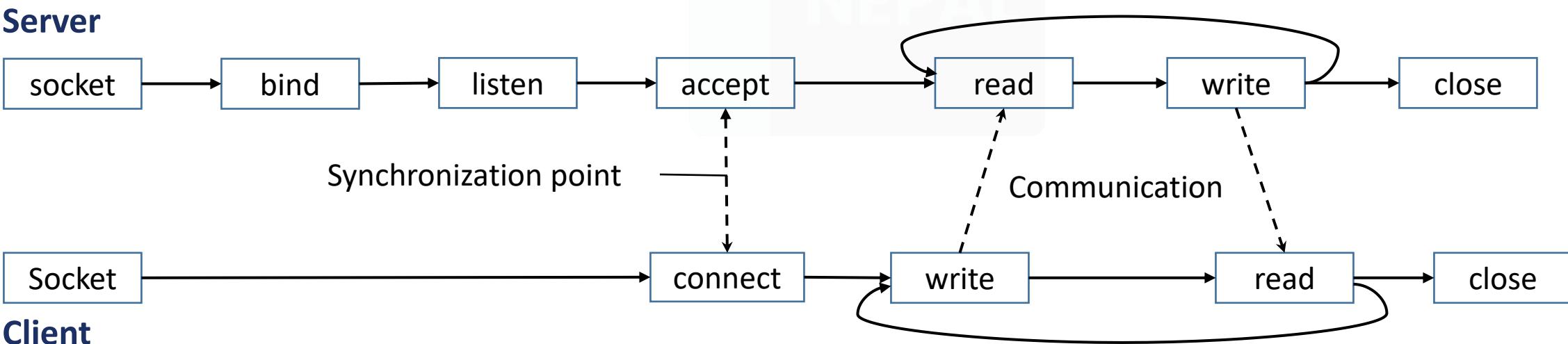
- `Socket()` – Endpoint for communication
- `Bind()` - Assign a unique telephone number.
- `Listen()` – Wait for a caller.
- `Connect()` - Dial a number.
- `Accept()` – Receive a call.
- `Send()`, `Recv()` – Talk.
- `Close()` – Hang up.

A

Berkeley Sockets

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives

Primitive	Meaning
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection
Write	Send data on the connection
Read	Get data that was sent on the connection



- ▶ Servers generally execute the first four primitives, normally in the order given. When calling the socket primitive, the caller creates a new communication end point for a specific transport protocol. Internally, creating a communication end point means that the local operating system reserves resources to accommodate sending and receiving messages for the specified protocol.
- ▶ The bind primitive associates a local address with the newly-created socket. For example, a server should bind the IP address of its machine together with a (possibly well-known) port number to a socket. Binding tells the operating system that the server wants to receive messages only on the specified address and port.
- ▶ The listen primitive is called only in the case of connection-oriented communication. It is a nonblocking call that allows the local operating system to reserve enough buffers for a specified maximum number of connections that the caller is willing to accept.

- ▶ A call to accept blocks the caller until a connection request arrives. When a request arrives, the local operating system creates a new socket with the same properties as the original one, and returns it to the caller. This approach will allow the server to, for example, fork off a process that will subsequently handle the actual communication through the new connection. The server, in the meantime, can go back and wait for another connection request on the original socket.
- ▶ Let us now take a look at the client side. Here, too, a socket must first be created using the socket primitive, but explicitly binding the socket to a local address is not necessary, since the operating system can dynamically allocate a port when the connection is set up. The connect primitive requires that the caller specifies the transport-level address to which a connection request is to be sent. The client is blocked until a connection has been set up successfully, after which both sides can start exchanging information through the send and receive primitives. Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the close primitive. The general pattern followed by a client and server for connection-oriented communication using sockets is shown in Fig (Slide 54)

The Message-Passing Interface (MPI)

- ▶ Message Passing Interface (MPI) is a subroutine or a library for passing messages between processes in a distributed memory model.
- ▶ MPI is not a programming language. MPI is a programming model that is widely used for parallel programming in a cluster.
- ▶ It is a standard API that can be used to create applications for high-performance multicomputers.
- ▶ Specific network protocols (not TCP/IP)
- ▶ Message-based communication
- ▶ Primitives for all 4 forms of transient communication (+ variations)
- ▶ Vendors + Open Source
 - IBM, Intel, TMC, Meiko, Cray, Convex, Ncube, OpenMPI

Message-passing primitives of MPI

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

- ▶ Transient asynchronous communication is supported by means of the `MPI_bsend` primitive. The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system. When the message has been copied, the sender continues. The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive primitive.
- ▶ There is also a blocking send operation, called `MPI_send`, of which the semantics are implementation dependent. The primitive `MPI_send` may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation. Synchronous communication by which the sender blocks until its request is accepted for further processing is available through the `MPI_ssend` primitive. Finally, the strongest form of synchronous communication is also supported: when a sender calls `MPI_sendrecv`, it sends a request to the receiver and blocks until the latter returns a reply. Basically, this primitive corresponds to a normal RPC.

- ▶ Both `MPI_send` and `MPI_ssend` have variants that avoid copying messages from user buffers to buffers internal to the local MPI runtime system. These variants correspond to a form of asynchronous communication. With `MPI_isend`, a sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. To prevent overwriting the message before communication completes, MPI offers primitives to check for completion, or even to block if required. As with `MPI_send`, whether the message has actually been transferred to the receiver or that it has merely been copied by the local MPI runtime system to an internal buffer is left unspecified.
- ▶ Likewise, with `MPI_issend`, a sender also passes only a pointer to the MPI runtime system. When the runtime system indicates it has processed the message, the sender is then guaranteed that the receiver has accepted the message and is now working on it.
- ▶ The operation `MPI_recv` is called to receive a message; it blocks the caller until a message arrives. There is also an asynchronous variant, called `MPI_irecv`, by which a receiver indicates that it is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or block until one does.

- ▶ The semantics of MPI communication primitives are not always straightforward, and different primitives can sometimes be interchanged without affecting the correctness of a program. The official reason why so many different forms of communication are supported is that it gives implementers of MPI systems enough possibilities for optimizing performance. Cynics might say the committee could not make up its collective mind, so it threw in everything. MPI has been designed for high-performance parallel applications, which makes it easier to understand its diversity in different communication primitives.

Why MPI ?

- ▶ There are many reasons for using MPI as our parallel programming model:
- ▶ MPI is a standard message passing library, and it is supported on all high-performance computer platforms.
- ▶ An MPI program is able to run on different platforms that support the MPI standard without changing your source codes.
- ▶ Because of its parallel features, programmers are able to work on a much larger problem size with the faster computation.
- ▶ There are many useful functions available in the MPI Library.
- ▶ A variety of implementations are available.

More to study on MPI

- ▶ References
- ▶ <https://csis.pace.edu/~marchese/CS865/Papers/forum94mpi.pdf>
- ▶ <https://www.mcs.anl.gov/research/projects/mpi/>

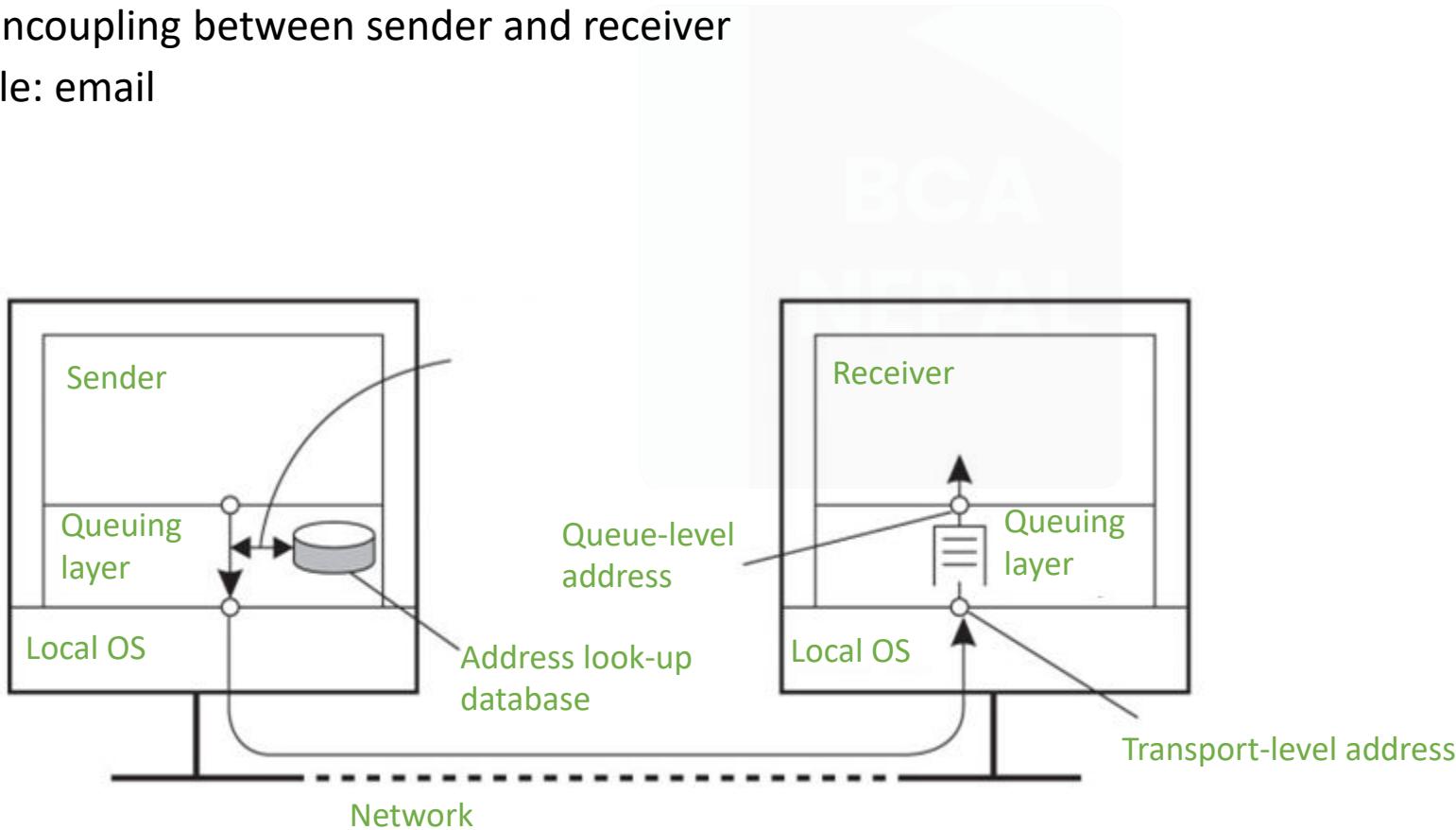
Message-Oriented Persistent Communication

- ▶ We now come to an important class of message-oriented middleware services, generally known as message-queuing systems, or just Message-Oriented Middleware (MOM).
- ▶ Message-queuing systems provide extensive support for persistent asynchronous communication.
- ▶ The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission.
- ▶ An important difference with Berkeley sockets and MPI is that message-queuing systems are typically targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds.
- ▶ We first explain a general approach to message-queuing systems, and conclude this section by comparing them to more traditional systems, notably the Internet e-mail systems.

Message-Oriented Persistent Communication

Message-queuing systems –Message-Oriented Middleware (MOM)

- Basic idea: MOM provides message storage service.
- A message is put in a queue by the sender, and delivered to a destination queue
- The target(s) can retrieve their messages from the queue
- Time uncoupling between sender and receiver
- Example: email

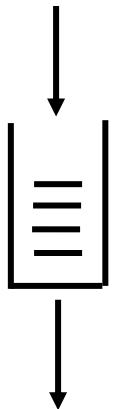
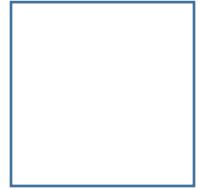


Message-Queuing Model

- ▶ The persistency is fulfilled by the various queues in the system
- ▶ Each application has a private queue to which other applications can send messages
- ▶ Message-queuing systems guarantee that the message eventually will be inserted in recipient's queue
 - No guarantee about when it is delivered
 - No guarantee whether the message will ever be read at all
- ▶ Only guarantee: your message will eventually make it into the receiver's message queue.
- ▶ Loosely-coupled communication
- ▶ Systemwide unique name of the destination queue is used for addressing

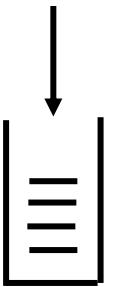
Loosely-Coupled Communication

Sender
running



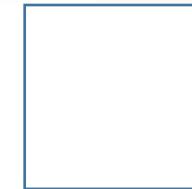
Receiver
running

Sender
running



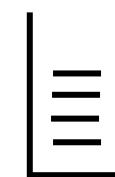
Receiver
passive

Sender
passive



Receiver
running

Sender
passive



Receiver
passive

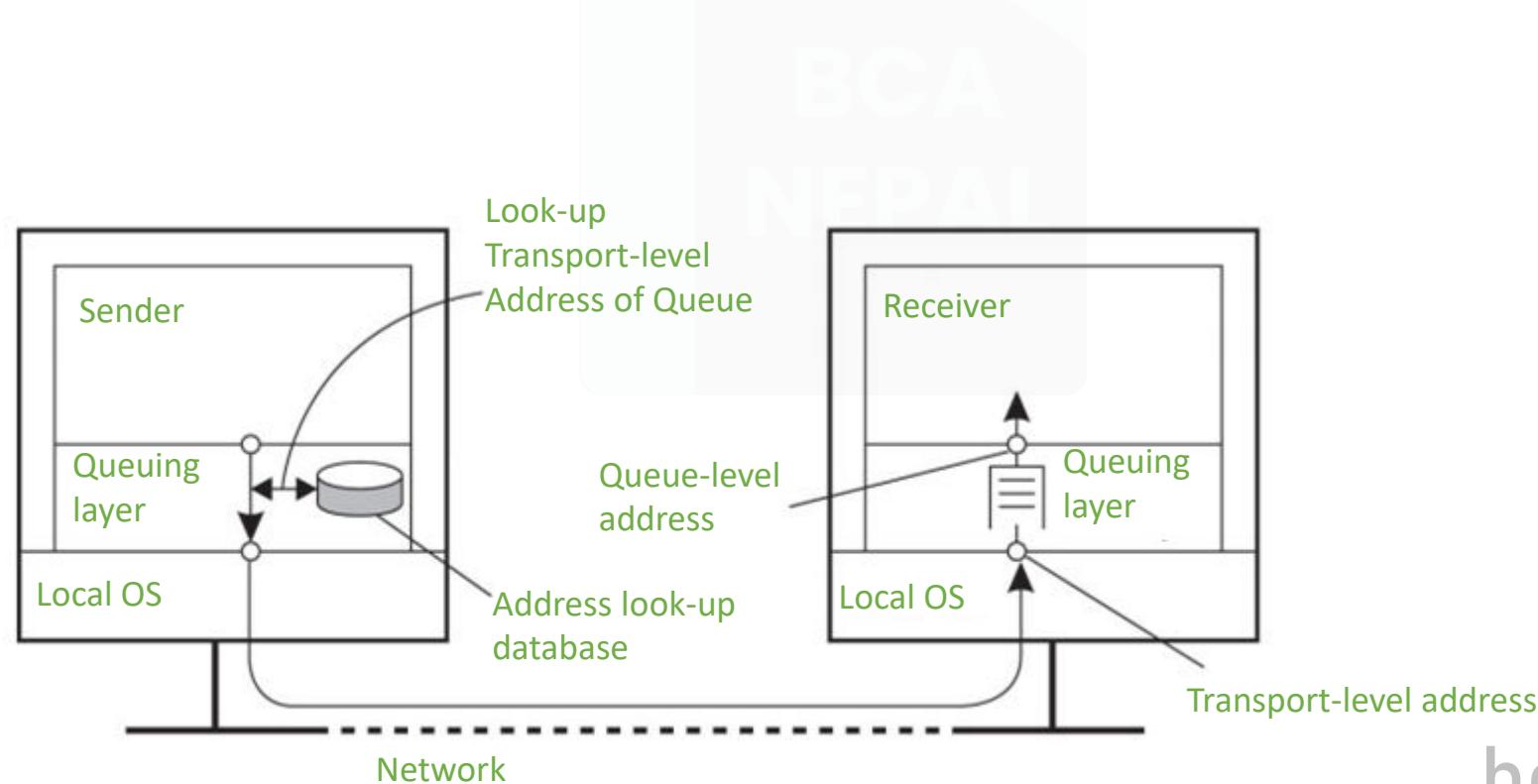
Message-Oriented Persistent Communication

Four Commands:

Primitive	Meaning
Put	Append a message to a specified queue.
Get	Block until the specified queue is nonempty, and remove the first message.
Poll	Check a specified queue for messages, and remove the first. Never block.
Notify	Install a handler to be called when a message is put into the specified queue.

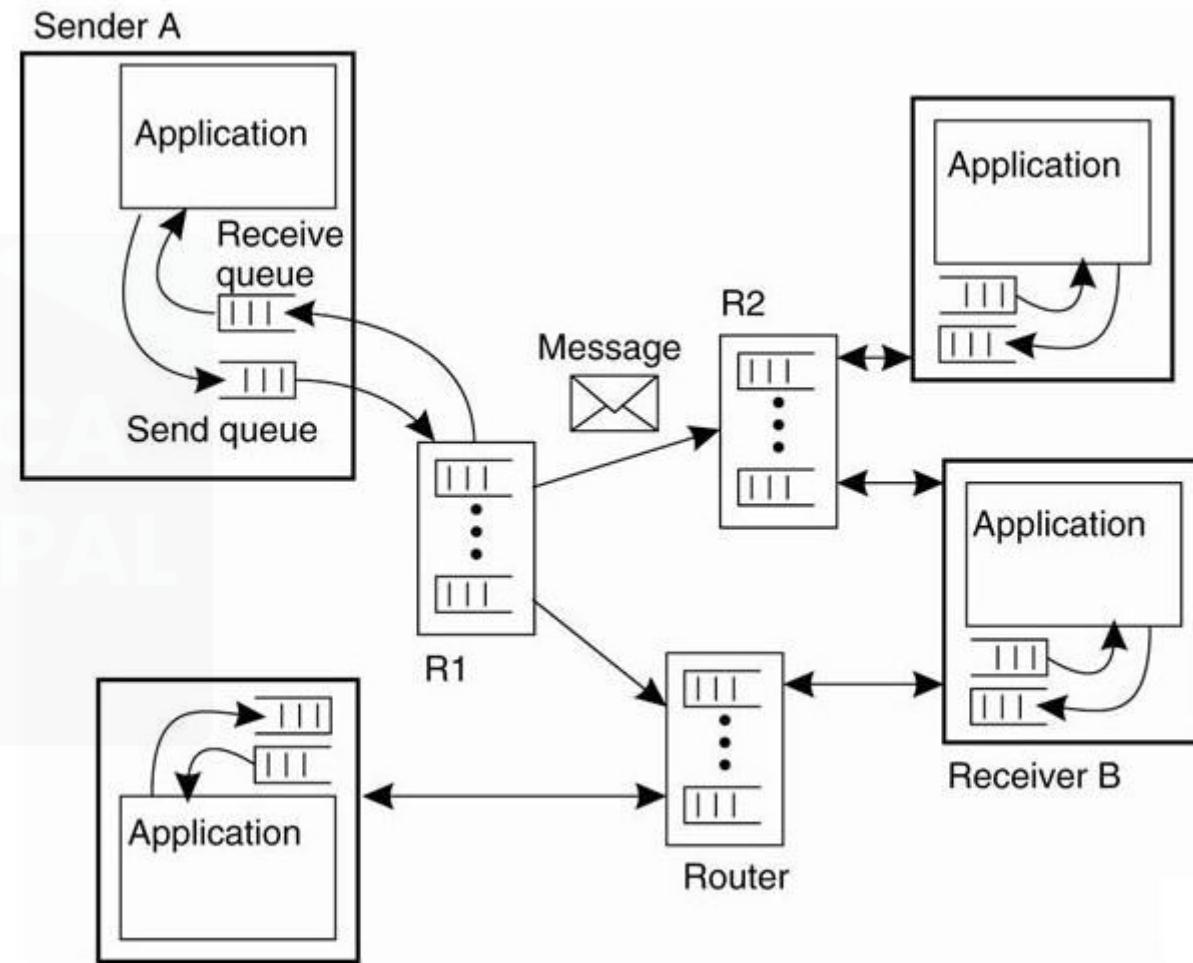
General Architecture of a Message-Queuing System

- ▶ Messages can be put only into queues that are local to the sender (same machine or on a nearby machine on a LAN)
- ▶ Such a queue is called the source queue
- ▶ Messages can also be read only from local queues
- ▶ A message put into a local queue must contain the specification of the destination queue; hence a messagequeuing system must maintain a mapping of queues to network locations; like in DNS



General organization of a message-queuing system with routers

- ▶ Messages are managed by queue managers
- ▶ They generally interact with the application that sends and receives messages
- ▶ Some also serve as routers or relays, i.e., they forward incoming messages to other queue managers
- ▶ However, each queue manager needs a copy of the queue-to-location mapping, leading to network management problems for large-scale queuing systems
- ▶ The solution is to use a few routers that know about the network topology



Stream-Oriented Communication

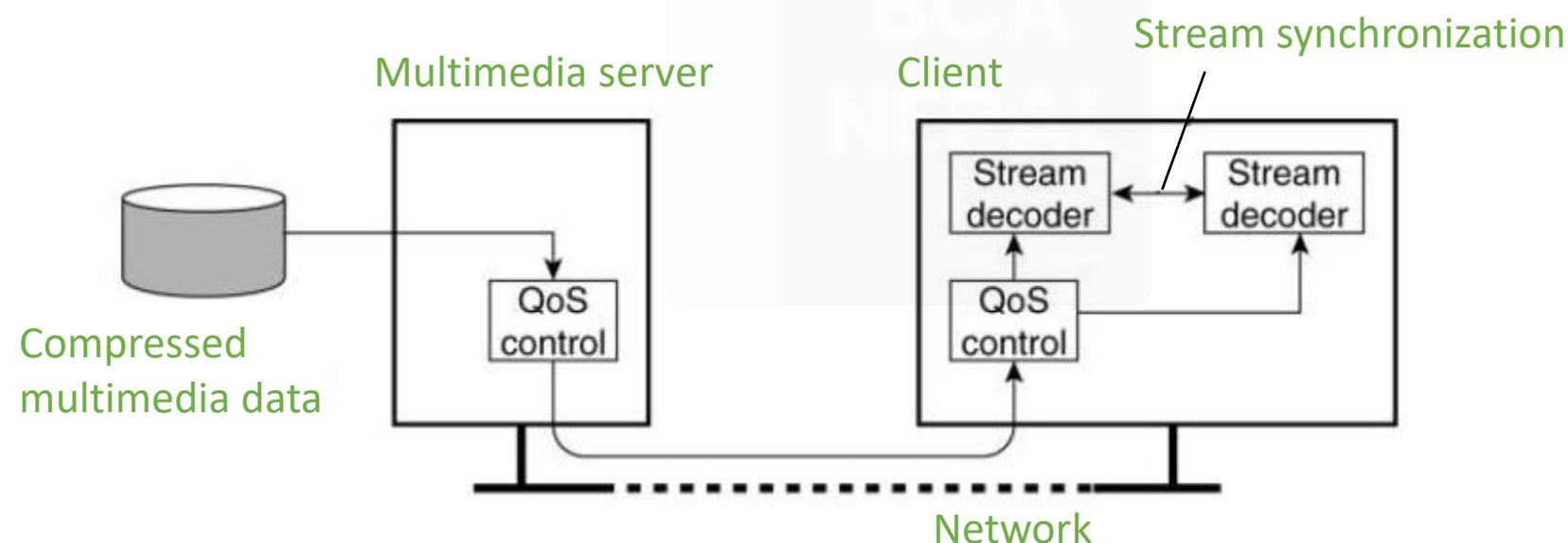
- ▶ RPC, RMI, message-oriented communication are based on the exchange of discrete messages
 - Timing might affect performance, but not correctness
- ▶ In stream-oriented communication the message content must be delivered at a certain rate, as well as correctly.
 - e.g., music or video
- ▶ Audio and video are time-dependent data streams – if the timing is off, the resulting “output” from the system will be incorrect.
- ▶ Time-dependent information – known as “continuous media” communications.
- ▶ Example:
 - Voice: PCM: 1/44100 sec intervals on playback.
 - Video: 30 frames per second (30-40 msec per image).

Transmission Modes in Stream-Oriented Communication

- ▶ **Asynchronous transmission mode** – the data stream is transmitted in order, but there's **no timing constraints** placed on the actual delivery (e.g., File Transfer).
- ▶ **Synchronous transmission mode** – the **maximum end-to-end delay** is defined (but data can travel faster).
- ▶ **Isochronous transmission mode** – data transferred “**on time**” – there's a maximum and minimum end-to-end delay (known as “**bounded jitter**”).
 - Known as “**streams**” – isochronous transmission mode is very useful for multimedia systems.
 - e.g., audio & video

Types of Streams in Stream-Oriented Communication

- ▶ Simple Streams – one single sequence of data, for example: voice
- ▶ Complex Streams – several sequences of data (substreams) that are “related” by time.
 - Think of a lip synchronized movie, with sound and pictures, together with sub-titles
 - This leads to data synchronization problems ... which are not at all easy to deal with



A general architecture for streaming stored multimedia data over a network

Quality of Service

- ▶ Ensuring that the temporal relationships in the stream can be preserved
- ▶ Streams are all about timely delivery of data.
- ▶ How do you specify this Quality of Service (QoS)?

Basics:

- The required bit rate at which data should be transported
- The maximum delay until a session has been set up (i.e., when an application can start sending data)
- The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient)
- The maximum delay variance, or jitter
- The maximum round-trip delay

Multicast Communication

1. Application-level multicasting
2. Gossip-based data dissemination



Application-level multicasting

- ▶ Basic idea: organize nodes of a distributed system into an overlay network and use that network to disseminate data

Multicast tree construction in Chord

- ▶ Initiator generates a multicast identifier mid.
- ▶ Lookup $\text{succ}(\text{mid})$, the node responsible for mid.
- ▶ Request is routed to $\text{succ}(\text{mid})$, which will become the root.
- ▶ If P wants to join, it sends a join request to the root
- ▶ When request arrives at Q
 - Q has not seen a join request for mid before → it becomes forwarder; P becomes child of Q. Join request continues to be forwarded.
 - Q is already a forwarder for mid → P becomes child of Q. No need to forward join request anymore.

Gossip-Based Data Dissemination

- ▶ Use epidemic algorithm to rapidly propagate information among a large collection of nodes with no central coordinator
 - Assume all updates for a specific data item are initiated at a single node
 - Upon an update, try to “infect” other nodes as quickly as possible
 - Pair-wise exchange of updates (like pair-wise spreading of a disease)
 - Eventually, each update should reach every node
- ▶ Terminology:
 - **Infected node:** node with an update it is willing to spread
 - **Susceptible node:** node that is not yet updated

Distributed System

Course Code: CACS352
Year/Sem: III/VI

Unit:5

Unit 5. Naming

- 5.1 Name Identifiers, and Addresses**
- 5.2 Structured Naming**
- 5.3 Attribute-based naming**
- 5.4 Case Study: The Global Name Service**



Chapter_05: Naming

- Names
- Identifiers
- Addresses
- Flat Naming
- Structured Naming
- Attribute-Based Naming
- Case Study: The Global Name Services

Introduction

- ▶ Names play a very important role in all computer systems.
- ▶ They are used to share resources, to uniquely identify entities, to refer to locations, and more.
- ▶ An important issue with naming is that a name can be resolved to the entity it refers to.
- ▶ Name resolution thus allows a process to access the named entity. To resolve names, it is necessary to implement a naming system.
- ▶ The difference between naming in distributed systems and nondistributed systems lies in the way naming systems are implemented.
- ▶ In a distributed system, the implementation of a naming system is itself often distributed across multiple machines.
- ▶ How this distribution is done plays a key role in the efficiency and scalability of the naming system.
- ▶ In this chapter, we concentrate on three different, important ways that names are used in distributed systems.

Names, identifiers, and addresses

- ▶ **Name:** set of *bits/characters* used to identify/refer to an entity, a collective of entities, etc. in a context.
 - Example of entity: hosts, printers, disks, files, webpages, users, messages, network connections and so on.
 - Simply comparing two names, we might not be able to know if they refer to the same entity
- ▶ **Identifier:** a name that uniquely identifies an entity
 - The identifier is *unique* and refers to only one entity
- ▶ **Address:** the name of an *access point*, the location of an entity
 - e.g. phone number, or IP-address + port for a service

Naming

- ▶ Names (character or bit strings) are used to denote entities in a distributed system.
- ▶ Entity: resources, processes, users, mailboxes, newsgroups, queues, web pages, etc.
- ▶ To operate on an entity, we need an **access point**.
- ▶ An access points is a special kind of entity whose name is called **address**
 - The address of an access point of an entity is also called an address of that entity
 - An entity may offer more than one access point
 - Access point of an entity may change
- ▶ Can be human-friendly(or not) and location dependent(or not)
- ▶ Naming systems are classified into three classes based on the type of names used:
 - ➔ Flat naming
 - ➔ Structured naming
 - ➔ Attribute-based naming

Identifiers

- ▶ A true identifier is a name with the following properties:
 - ➔ **P1:** An identifier refers to at most one entity
 - ➔ **P2:** Each entity is referred to by at most one identifier
 - ➔ **P3:** An identifier always refers to the same entity
- ▶ Addresses and identifiers are two important types of names that are each used for different purposes
- ▶ Human-friendly names - UNIX file names

True identifier

- ▶ The purpose of a name is to identify an entity and act as an access point to it.
- ▶ To fulfil the role of true identifier,
 - Name refers to only one entity
 - Different names refer to different entities
 - Name always refers to the same entity (i.e. ***it is not reused***)

Name Resolution

- ▶ Given a path name, it is easy to lookup information stored in the node for the entity in name space
- ▶ The process of looking up a name is called **name resolution**.
- ▶ Example:
 - N:<label-1, label-2, ..., label-n>
 - Resolution starts at N, looking up the directory table returns the identifier for the node that label-1 refers to, N1
 - Second, looking up the label-2 in directory table of the node N1, and so on
 - Resolution stops at node referred to by label-n by returning the content of that node

Flat Naming

- ▶ Flat names are fixed size bit strings
 - Can be efficiently handled by machines
 - Identifiers are flat names
- ▶ In flat naming, identifiers are simply random bits of strings (known as unstructured or flat names)
- ▶ Flat name does not contain any information on how to locate an entity
- ▶ How to resolve flat names?
 - Broadcasting
 - Forwarding pointers
 - Home-based approaches: Mobile IP
 - Distributed Hash Tables: Chord

Structured Naming

- ▶ Flat names are difficult for humans to remember
- ▶ Structured names are composed of simple human-readable names
 - Names are arranged in a specific structure
- ▶ Examples:
 - File-systems utilize structured names to identify files
 - /home/userid/work/dist-systems/naming.txt
 - Websites can be accessed through structured names
 - www.diet.engg.cse.ce

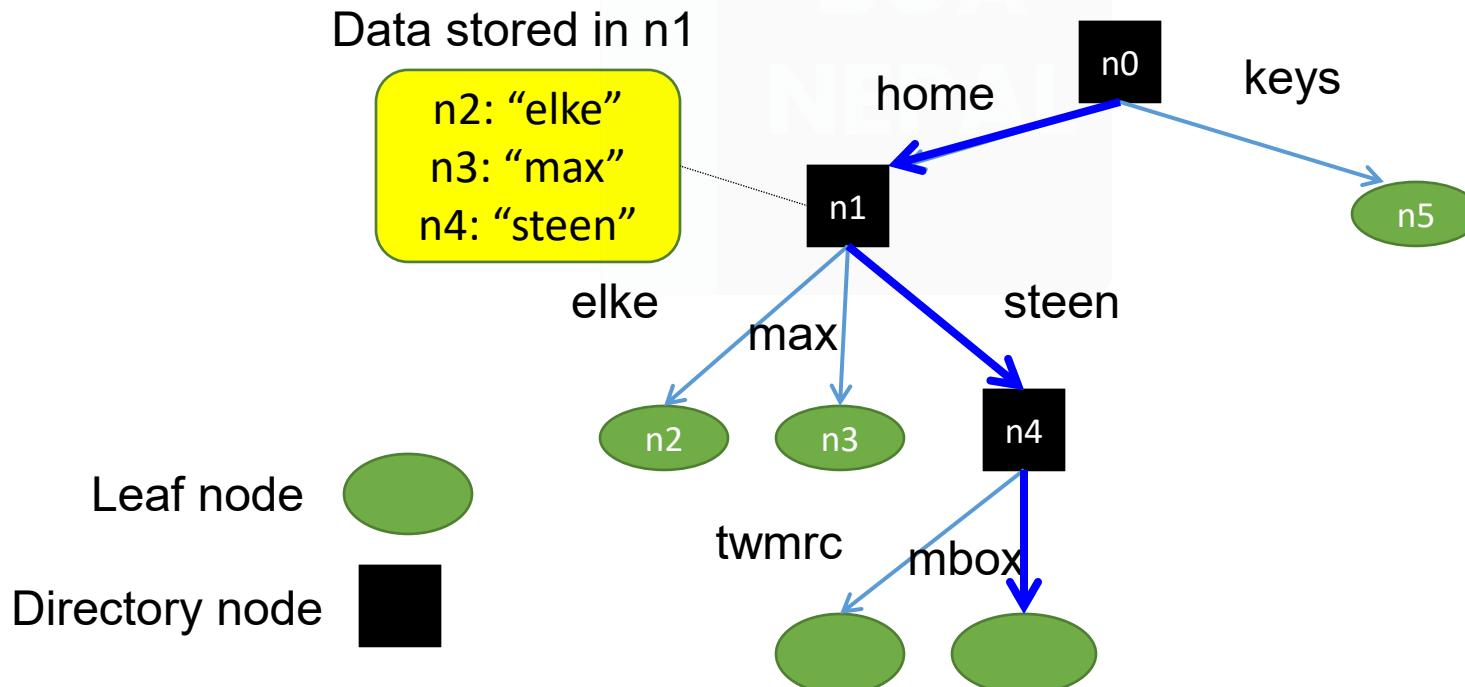
Name Spaces

- ▶ Structured names are organized into **name spaces**
- ▶ A name space is a directed graph consisting of:
 - ➔ Leaf nodes
 - ➔ Directory nodes
- ▶ **Leaf nodes:**
 - Each leaf node represents an entity
 - A leaf node generally stores the address of an entity (e.g., in DNS), or the state of (or the path to) an entity (e.g., in file systems)
- ▶ **Directory nodes**
 - Directory node refers to other leaf or directory nodes
 - Each outgoing edge is represented by (edge label, node identifier)
- ▶ Each node can store any type of data
- ▶ I.e., State and/or address (e.g., to a different machine) and/or path

Name Spaces - An Example

- ▶ A path name is used to refer to a node in the graph
- ▶ A path name is a sequence of edge labels leading from one node to another
 - An **absolute path** name starts from the root node (e.g., /home/steen/mbox)
 - A **relative path** name does not start at the root node (e.g., steen/mbox)

Looking up for the entity with name “/home/steen/mbox”



Name Resolution

- ▶ The process of looking up a name is called *name resolution*
- ▶ Closure mechanism:
 - Knowing how and where to start name resolution is generally referred to as a closure mechanism.
 - Name resolution cannot be accomplished without an initial directory node
 - Essentially, a closure mechanism deals with selecting the initial node in a name space from which name resolution is to start
 - Examples:
 - ➔ www.qatar.cmu.edu: start at the DNS Server
 - ➔ /home/steen/mbox: start at the root of the file-system

Name Linking

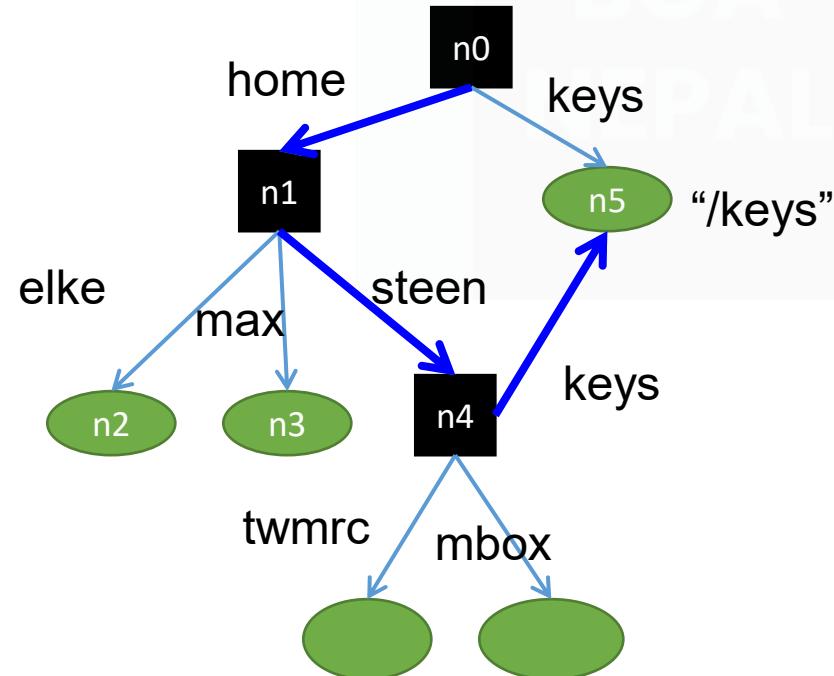
- ▶ The name space can be effectively used to link two different entities
- ▶ Two types of links can exist between the nodes:
 1. Hard Links
 2. Symbolic Links



Hard Links

- ▶ There is a ***directed link*** from the hard link to the actual node
- ▶ Name resolution: Similar to the general name resolution
- ▶ Constraint: There should be ***no cycles*** in the graph

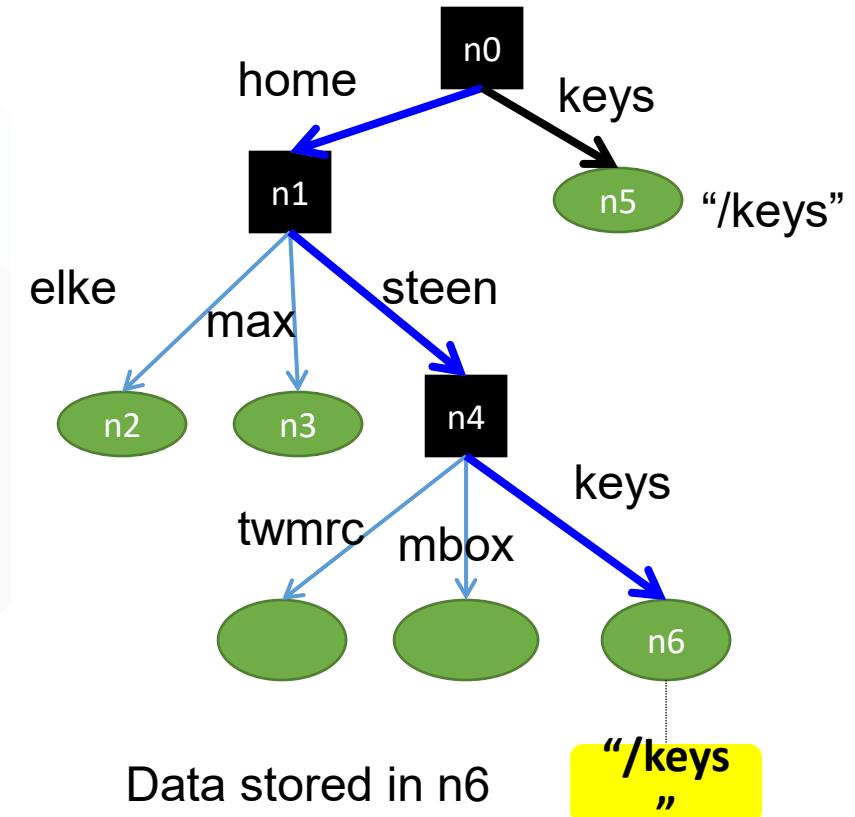
“/home/steen/keys” is a hard link to “/keys”



Symbolic Links

- ▶ Symbolic link stores the name of the original node as data
- ▶ Name resolution for a symbolic link SL
 - First resolve SL's name
 - Read the content of SL
 - Name resolution continues with content of SL
- ▶ Constraint: No cyclic references should be present

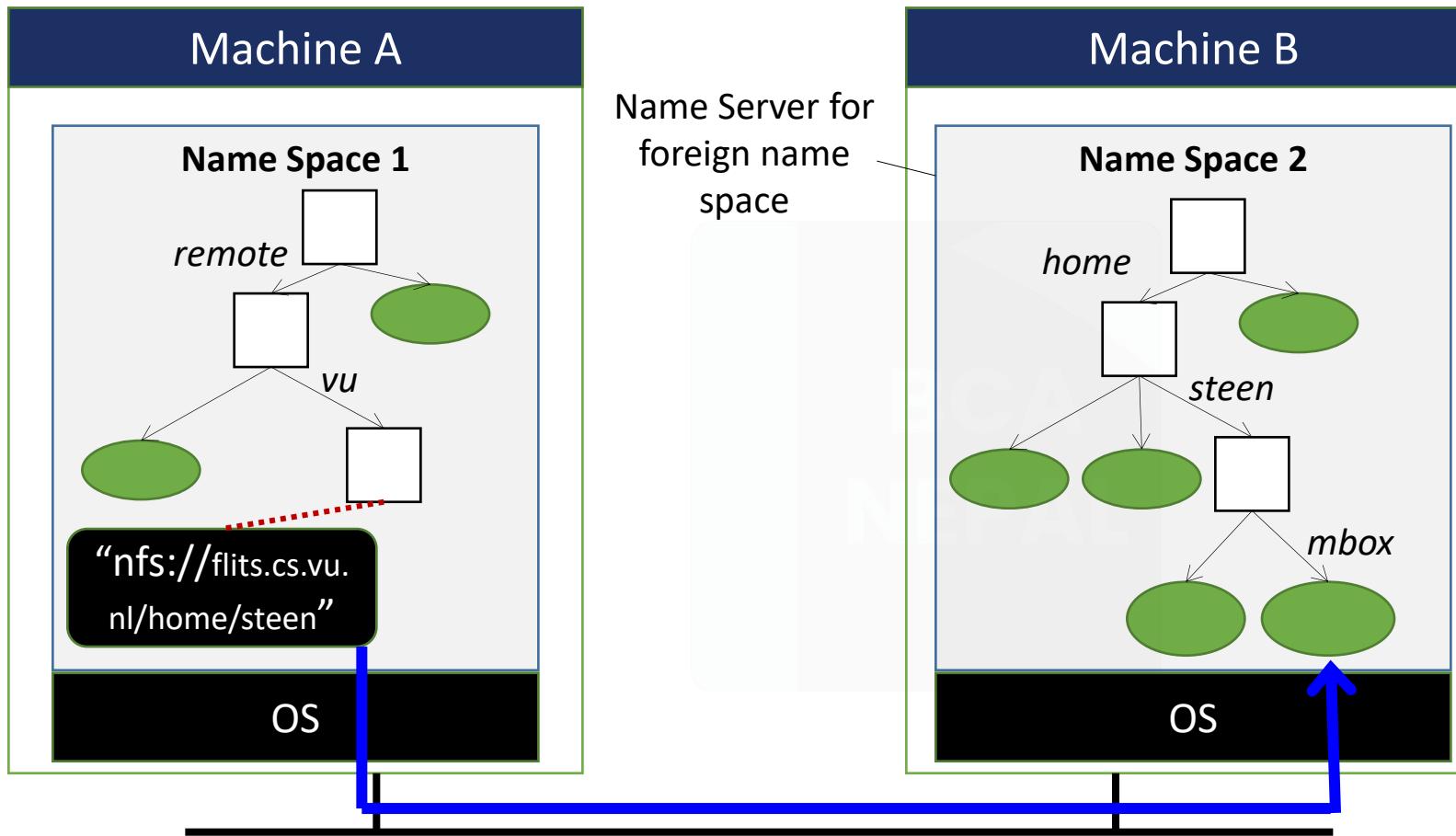
“/home/steen/keys” is a symbolic link to “/keys”



Mounting of Name Spaces

- ▶ Two or more name spaces can be merged transparently by a technique known as mounting
- ▶ With mounting, a directory node in one name space will store the identifier of the directory node of another name space
- ▶ Network File System (NFS) is an example where different name spaces are mounted
 - NFS enables transparent access to remote files

Example of Mounting Name Spaces in NFS



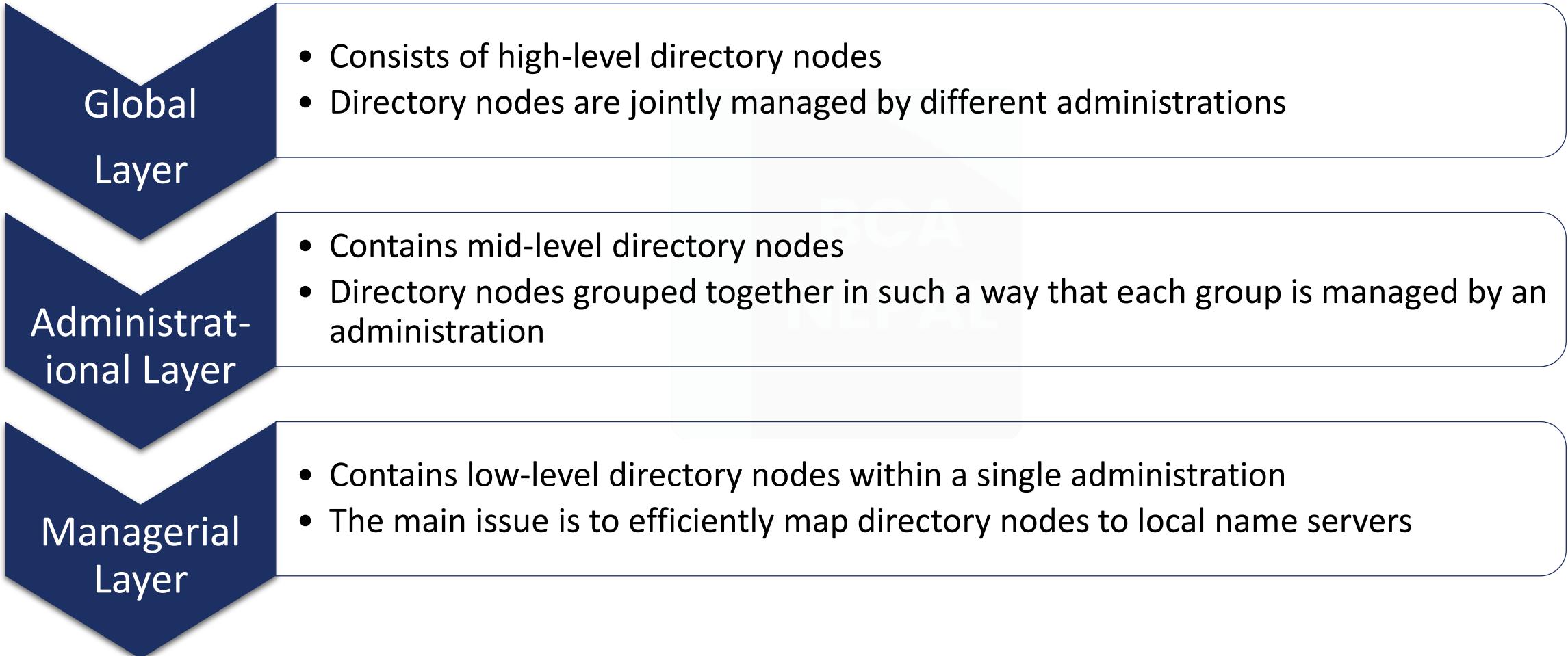
Name resolution for “/remote/vu/home/steen/mbox” in a distributed file system

Distributed Name Spaces

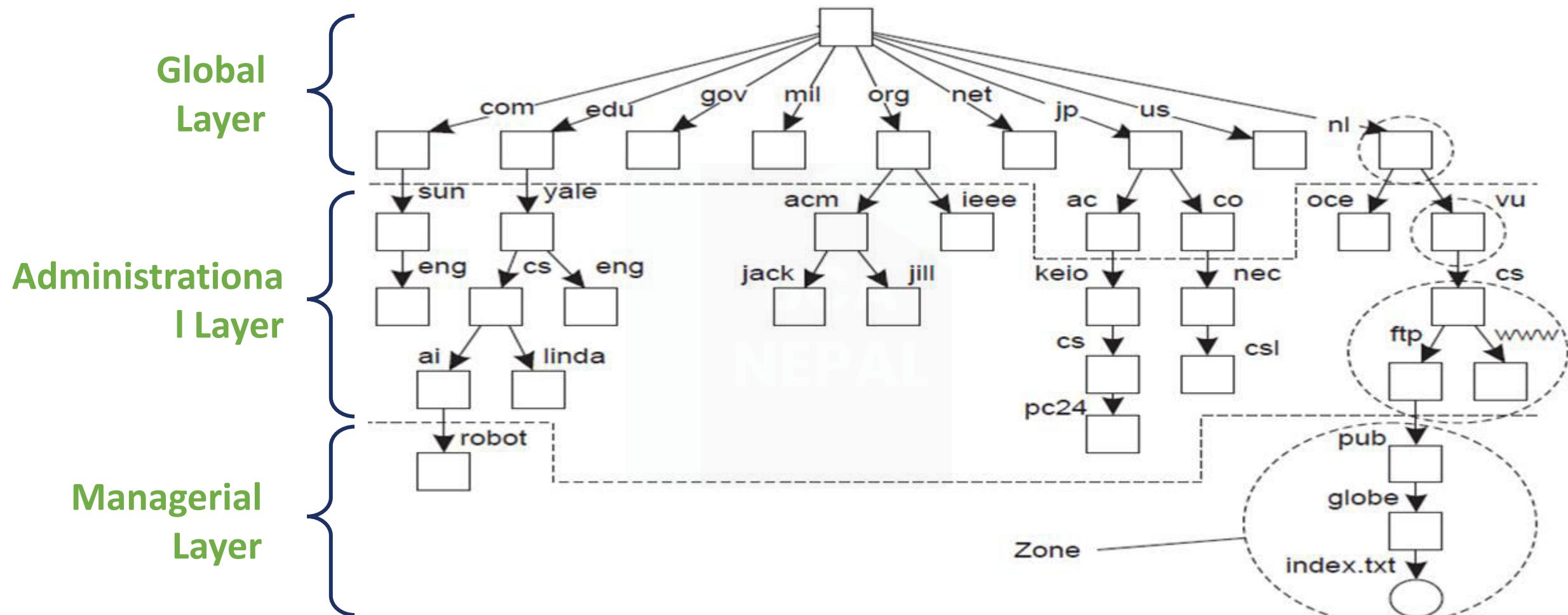
- ▶ In large-scale distributed systems, it is essential to distribute name spaces over multiple name servers
 - ➔ Distribute the nodes of the naming graph
 - ➔ Distribute the name space management
 - ➔ Distribute the name resolution mechanisms

Layers in Distributed Name Spaces

- ▶ Distributed name spaces can be divided into three layers



Distributed Name Spaces – An Example



Comparison of Name Servers at Different Layers

	Global	Administrational	Managerial
Geographical scale of the network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Number of replicas	Many	None or few	None
Update propagation	Lazy	Immediate	Immediate
Is client side caching applied?	Yes	Yes	Sometimes
Responsiveness to lookups	Seconds	Milliseconds	Immediate

DNS name space.

The most important types of resource records forming the contents of nodes in the DNS name space.

Type of record	Associated entity	Description
SOA (start of authority)	Zone	Holds information on the represented zone, such as an e-mail address of the system administrator
A (address)	Host	Contains an IP address of the host this node represents
MX (mail exchange)	Domain	Refers to a mail server to handle mail addressed to this node; it is a symbolic link; e.g. name of a mail server
SRV	Domain	Refers to a server handling a specific service
NS (name server)	Zone	Refers to a name server that implements the represented zone
CNAME	Node	Contains the canonical name of a host
PTR (pointer)	Host	Symbolic link with the primary name of the represented node
HINFO (host info)	Host	Holds information on the host this node represents; such as machine type and OS
TXT	Any kind	Contains any entity-specific information considered useful

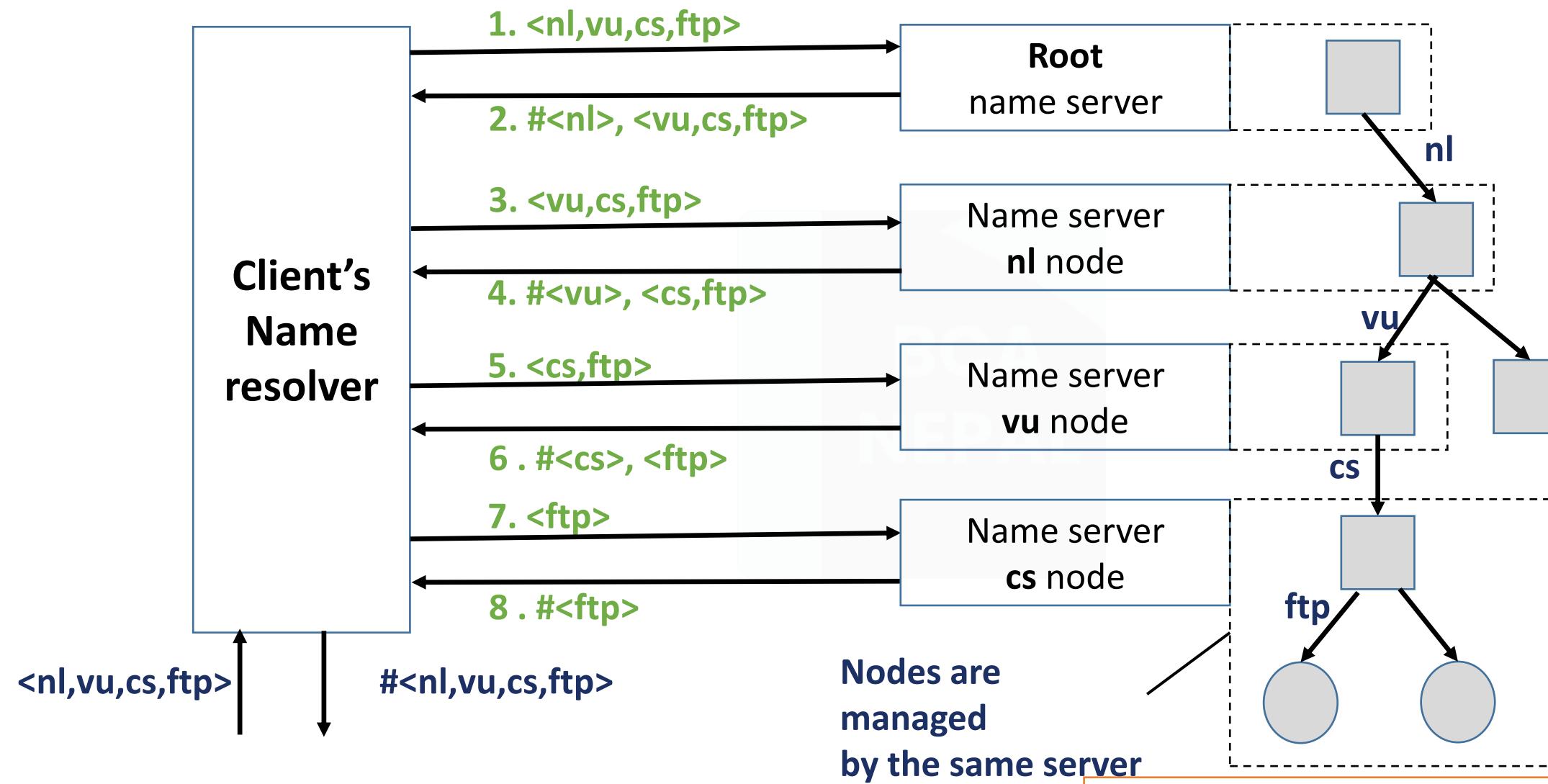
Distributed Name Resolution

- ▶ Distributed name resolution is responsible for mapping ***names to addresses*** in a system where:
 - Name servers are distributed among ***participating nodes***
 - Each name server has a ***local name resolver***
- ▶ Two distributed name resolution algorithms:
 1. Iterative Name Resolution
 2. Recursive Name Resolution

Iterative Name Resolution

- ▶ Client hands over the complete name to root name server
- ▶ Root name server resolves the name as far as it can, and returns the result to the client
- ▶ The root name server returns the address of the next-level name server (say, NLNS) if address is not completely resolved
- ▶ Client passes the unresolved part of the name to the NLNS
- ▶ NLNS resolves the name as far as it can, and returns the result to the client (and probably its next-level name server)
- ▶ The process continues until the full name is resolved

Iterative Name Resolution – An Example



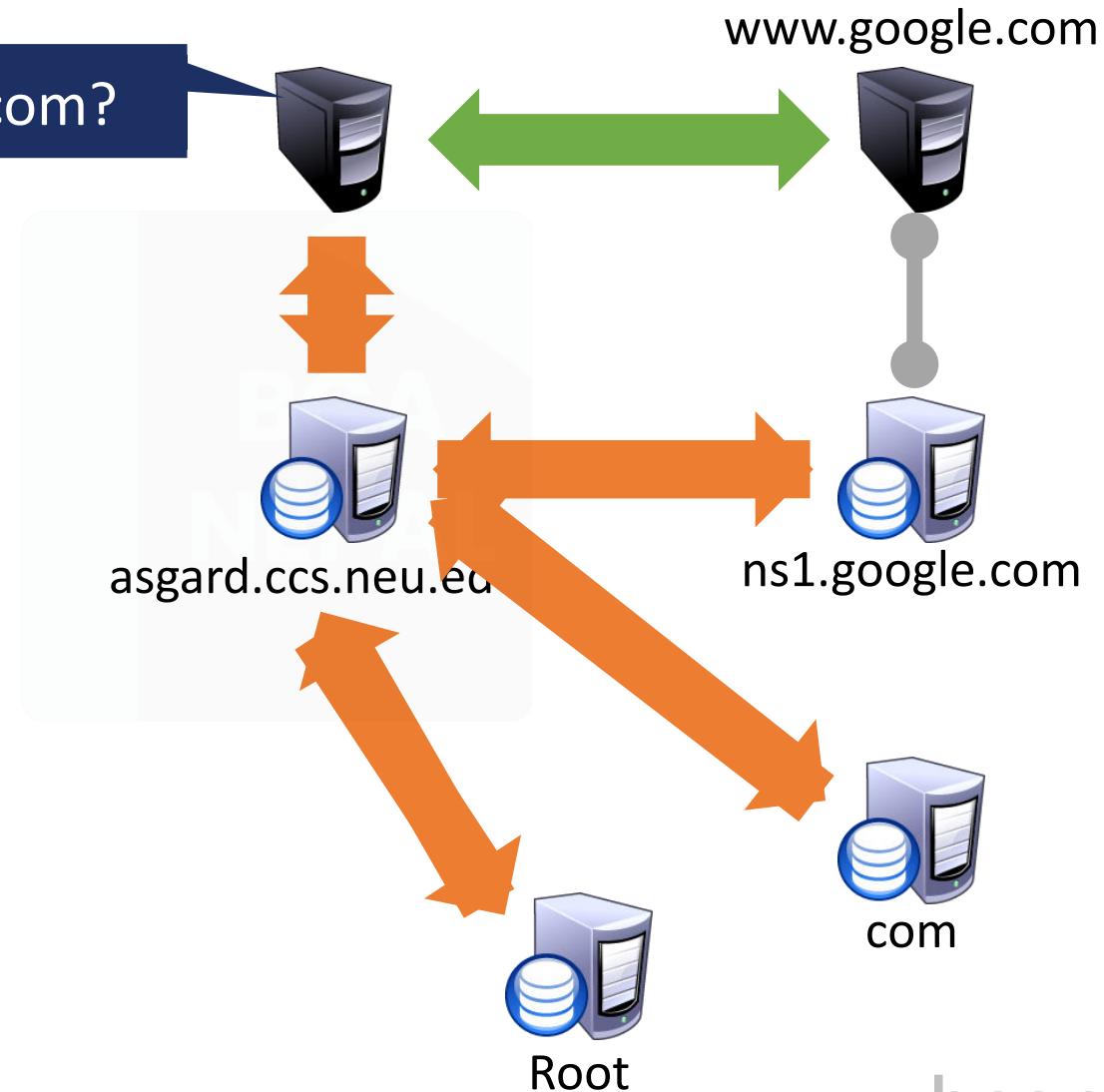
Resolving the name “*ftp.cs.vu.nl*”

<a,b,c> = structured name in a sequence
#<a> = address of node with name “a”

Iterated DNS query

Where is www.google.com?

- ▶ Contact server replies with the name of the next authority in the hierarchy
- ▶ “I don’t know this name, but this other server might”



Recursive Name Resolution

Approach:

- ▶ Client provides the name to the root name server
- ▶ The root name server passes the result to the next name server it finds
- ▶ The process continues till the name is fully resolved

Drawback:

- ▶ Large overhead at name servers (especially, at the high-level name servers)

Recursive Name Resolution

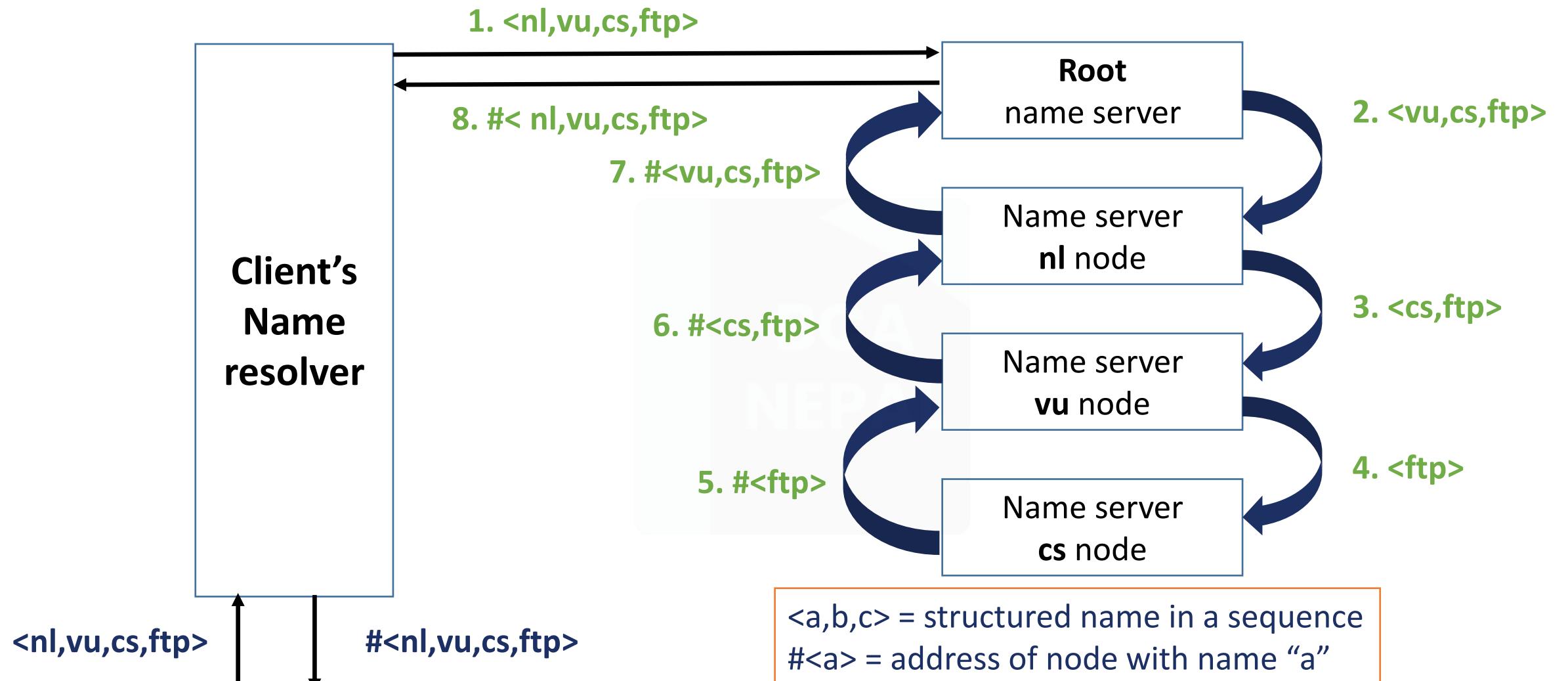
Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	<ftp>	#<ftp>	--	--	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Recursive Name Resolution – An example

Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	<ftp>	#<ftp>	--	--	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Recursive name resolution of <nl, vu, cs, ftp>
name servers cache intermediate results for subsequent lookups

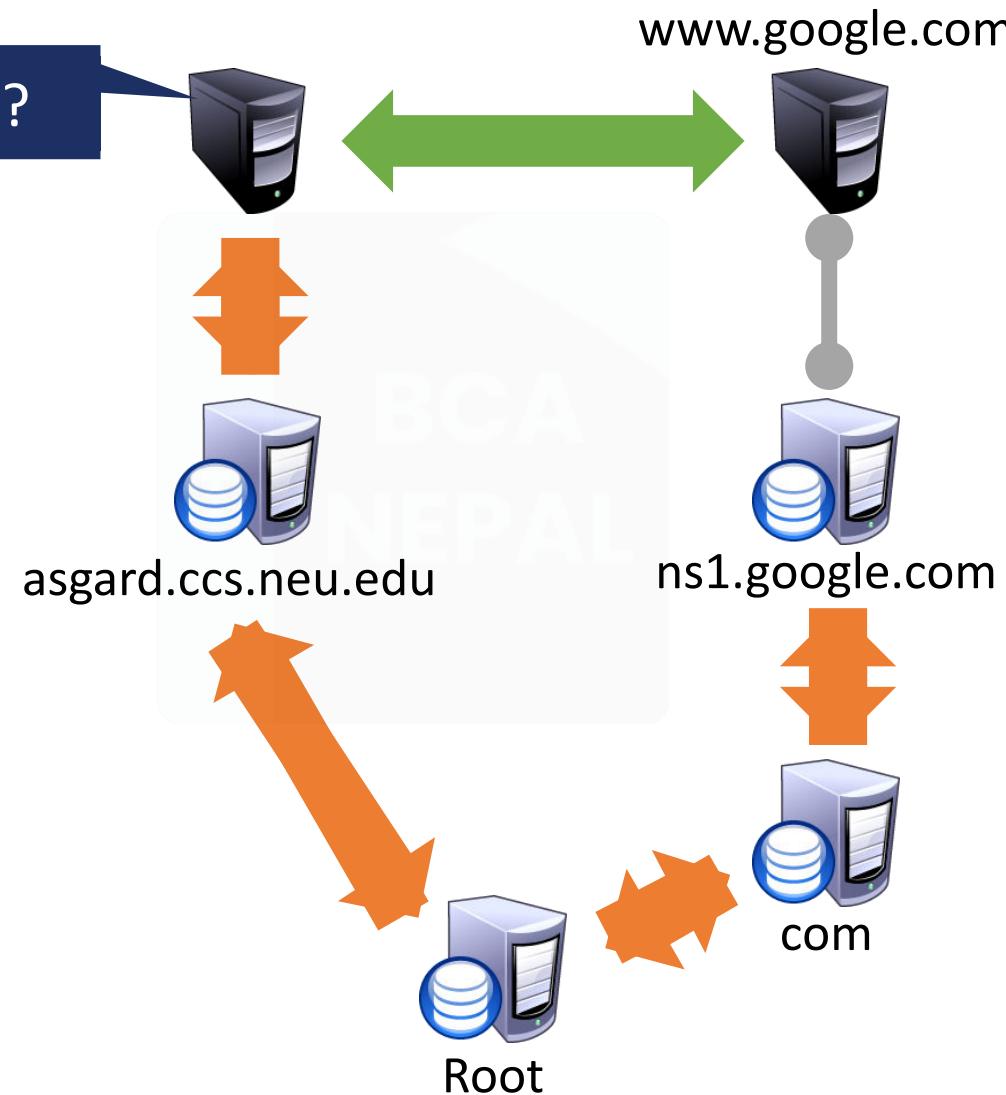
Recursive Name Resolution - An Example



Resolving the name "*ftp.cs.vu.nl*"

Recursive DNS Query

Where is www.google.com?



Recursive Name Resolution - Advantages and drawbacks

- ▶ Recursive name resolution puts a higher performance demand on each name server; hence name servers in the global layer support only iterative name resolution
- ▶ Caching is more effective with recursive name resolution; each name server gradually learns the address of each name server responsible for implementing lower-level nodes; eventually lookup operations can be handled efficiently
- ▶ The comparison between recursive and iterative name resolution:

Method	Advantage(s)
Recursive	Less Communication cost; Caching is more effective
Iterative	Less performance demand on name servers

Attribute-based Naming

- ▶ In many cases, it is much more convenient to name, and look up entities by means of their attributes
 - Similar to traditional directory services
- ▶ However, the lookup operations can be extremely expensive
 - They require to match requested attribute values, against actual attribute values, which might require inspecting all entities
- ▶ **Solution:** Implement basic directory service as a database, and combine it with traditional structured naming system
- ▶ Example: Light-weight Directory Access Protocol (LDAP)

Light-weight Directory Access Protocol (LDAP)

- ▶ LDAP directory service consists of a number of records called “directory entries”
 - Each record is made of (attribute, value) pairs
 - LDAP standard specifies five attributes for each record
- ▶ Directory Information Base (DIB) is a collection of all directory entries
 - Each record in a DIB is unique
 - Each record is represented by a distinguished name

E.g., /C=NL/O=Vrije Universiteit/OU=Comp. Sc.

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

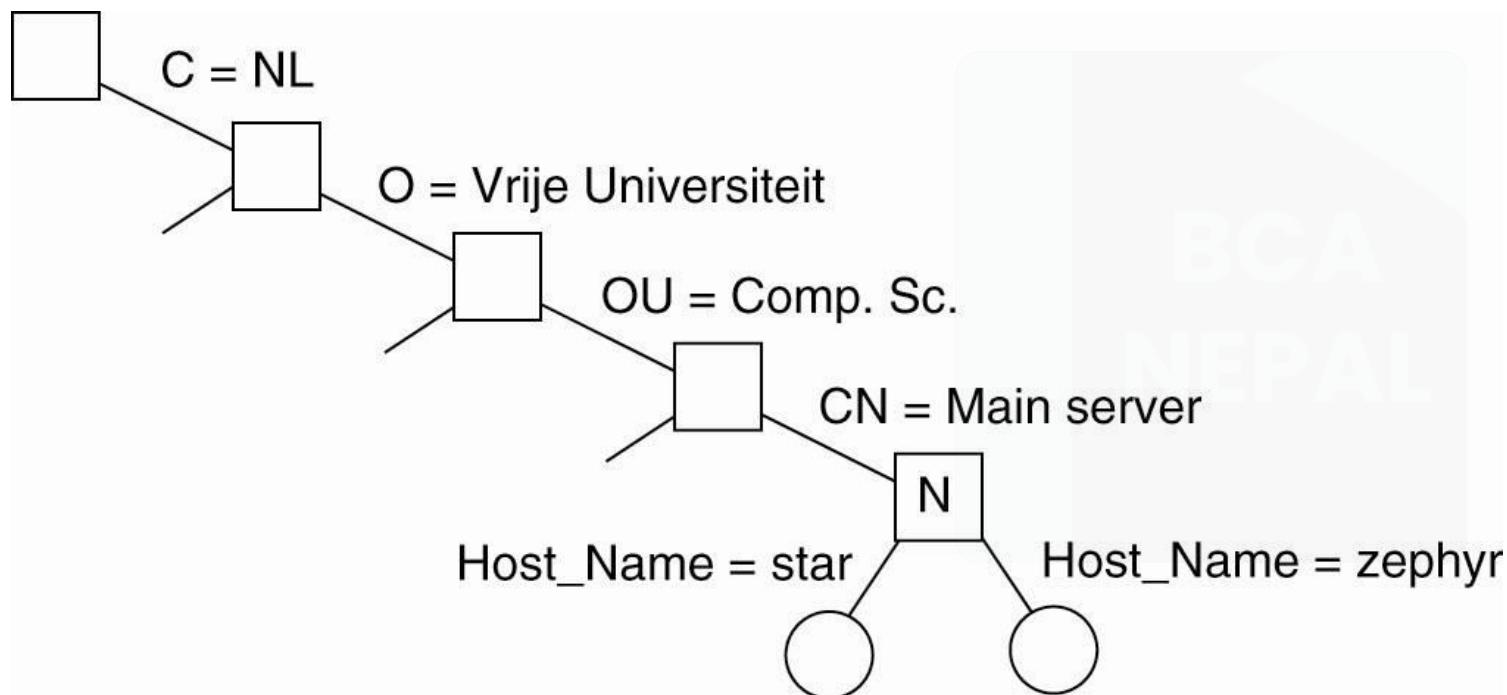
Directory Information Tree in LDAP

- ▶ All the records in the DIB can be organized into a hierarchical tree called Directory Information Tree (DIT)
- ▶ LDAP provides advanced search mechanisms based on attributes by traversing the DIT



Directory Information Tree in LDAP – An Example

```
search("&(C = NL) (O = Vrije Universiteit) (OU = * ) (CN = Main server)")
```



Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Summary

- ▶ Naming and name resolutions enable accessing entities in a distributed system
- ▶ Three types of naming:
 - ➔ Flat Naming
 - Broadcasting, forward pointers, home-based approaches, Distributed Hash Tables (DHTs)
 - ➔ Structured Naming
 - Organizes names into Name Spaces
 - Distributed Name Spaces
 - ➔ Attribute-based Naming
 - Entities are looked up using their attributes

Assignment

- ▶ Case study: The Global Name Service



Distributed System

Course Code: CACS352
Year/Sem: III/VI

Unit:06_Coordination

- 6.1 Clock Synchronization
- 6.2 Logical Clocks
- 6.3 Mutual Exclusion
- 6.4 Election Algorithm
- 6.5 Location System
- 6.6 Distributed Event Matching
- 6.7 Gossip-based coordination



Synchronization



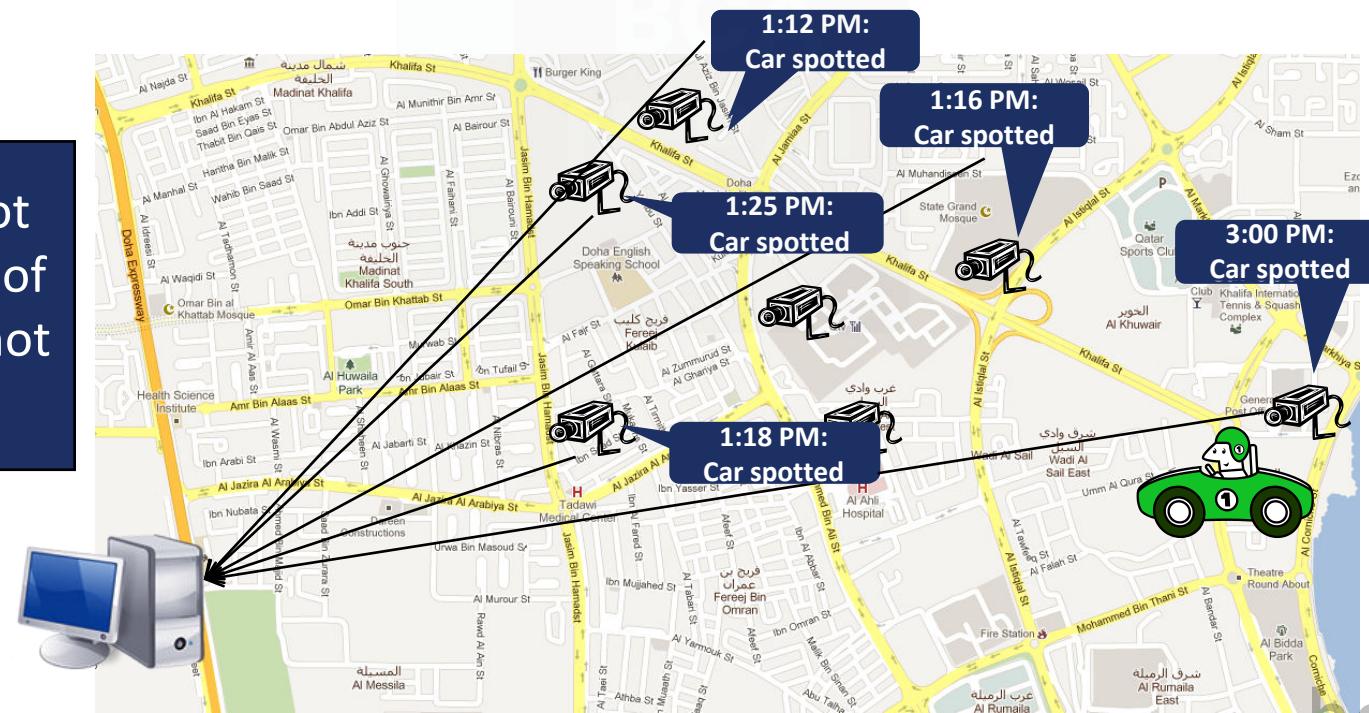
Synchronization

- ▶ Until now, we have looked at:
 - How entities communicate with each other
 - How entities are named and identified
- ▶ In addition to the above requirements, entities in DSs often have to cooperate and synchronize to solve a given problem correctly
 - E.g., In a distributed file system, processes have to synchronize and cooperate such that two processes are not allowed to write to the same part of a file

Need for Synchronization – An Example

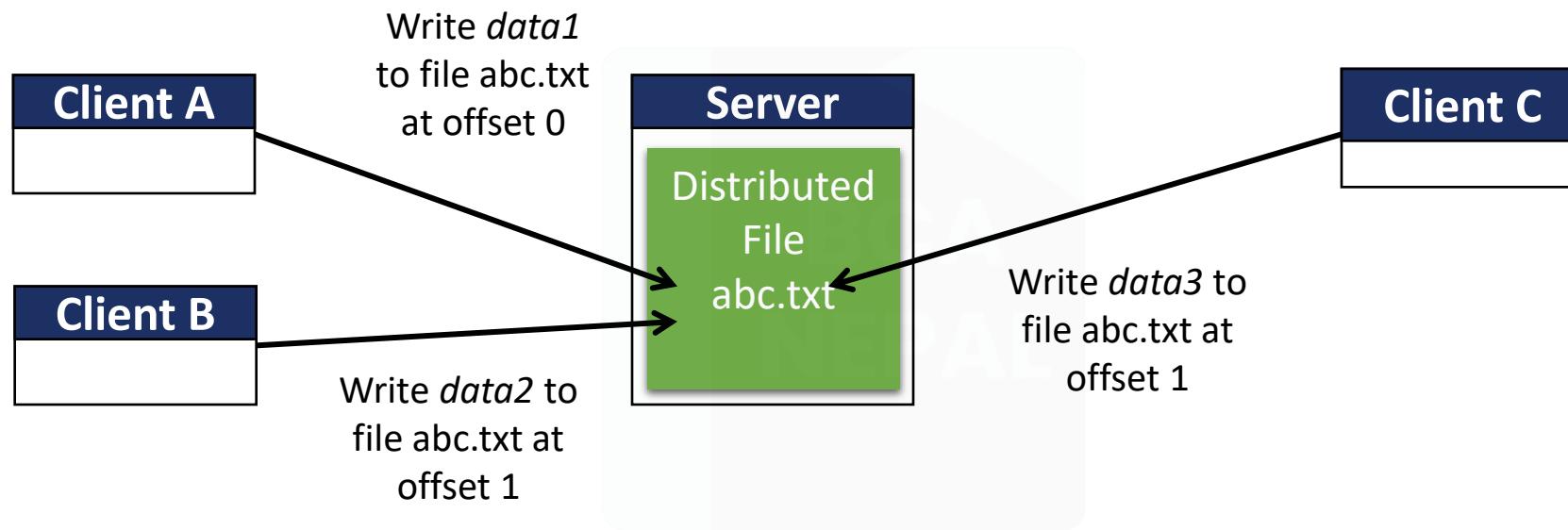
- ▶ Vehicle tracking in a City Surveillance System using a Distributed Sensor Network of Cameras
 - **Objective:** To keep track of suspicious vehicles
 - Camera Sensor Nodes are deployed over the city
 - Each Camera Sensor that detects a vehicle reports the time to a central server
 - Server tracks the movement of the suspicious vehicle

If the sensor nodes do not have a consistent version of the time, the vehicle cannot be reliably tracked



Need for Synchronization – An Example

Writing a file in a Distributed File System



If the distributed clients do not synchronize their write operations to the distributed file, then the data in the file can be corrupted

Clock Synchronization

- ▶ Clock synchronization is a mechanism to synchronize the time of all the computers in a DS
- ▶ It is often important to know when events occurred and in what order they occurred.
 - Need to know when a transaction occurs.
 - Online reservation system.
 - E-mail sorting can be difficult if time stamps are incorrect.
- ▶ In a non-distributed system dealing with time is trivial(less importance) as there is a single shared clock, where all processes see the same time.
- ▶ In a distributed system, on the other hand, each computer has its own clock.
- ▶ Because no clock is perfect each of these clocks has its own skew which causes clocks on different computers to drift and eventually become out of sync.
 - In centralized system, time is unambiguous
 - In distributed system, time is not trivial

Coordinated Universal Time (UTC)

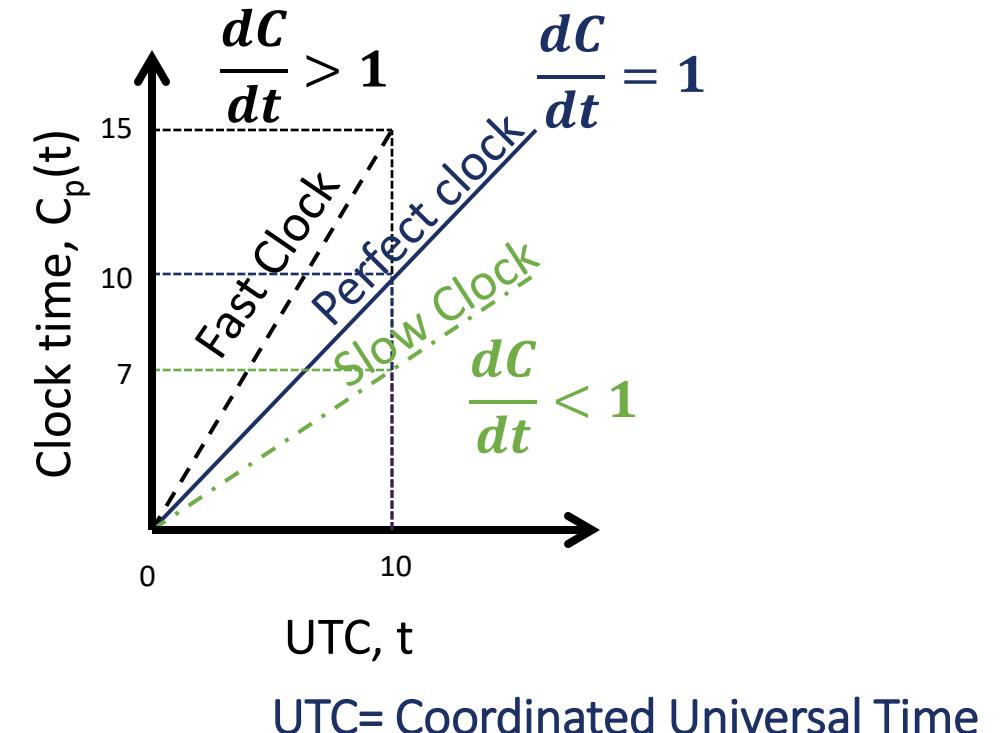
- ▶ All the computers are generally synchronized to a standard time called Coordinated Universal Time (UTC)
 - UTC is the primary time standard by which the world regulates clocks and time
- ▶ UTC is broadcasted via the satellites
 - UTC broadcasting service provides an accuracy of 0.5 msec
- ▶ Computer servers and online services with UTC receivers can be synchronized by satellite broadcasts
 - Many popular synchronization protocols in distributed systems use UTC as a reference time to synchronize clocks of computers

Tracking Time on a Computer

- ▶ How does a computer keep track of its time?
 - Each computer has a hardware timer
 - The timer causes an interrupt ‘H’ times a second
 - The interrupt handler adds 1 to its Software Clock (C)
- ▶ Issues with clocks on a computer
 - In practice, the hardware timer is imprecise
 - It does not interrupt ‘H’ times a second due to material imperfections of the hardware and temperature variations
 - The computer counts the time slower or faster than actual time
 - Loosely speaking, **Clock Skew** is the skew between:
 - The computer clock and the actual time (e.g., UTC)

Clock Skew

- ▶ **Clock Skew:** Difference between two clocks at one point in time.
- ▶ When the UTC time is t , let the clock on the computer have a time $C(t)$
- ▶ Three types of clocks are possible
 - ▶ Perfect clock:
 - The timer ticks 'H' interrupts a second
 $dC/dt = 1$
 - ▶ Fast clock:
 - The timer ticks more than 'H' interrupts a second
 $dC/dt > 1$
 - ▶ Slow clock:
 - The timer ticks less than 'H' interrupts a second
 $dC/dt < 1$



Clock Skew

- ▶ Frequency of the clock is defined as the ratio of the number of seconds counted by the software clock for every UTC second

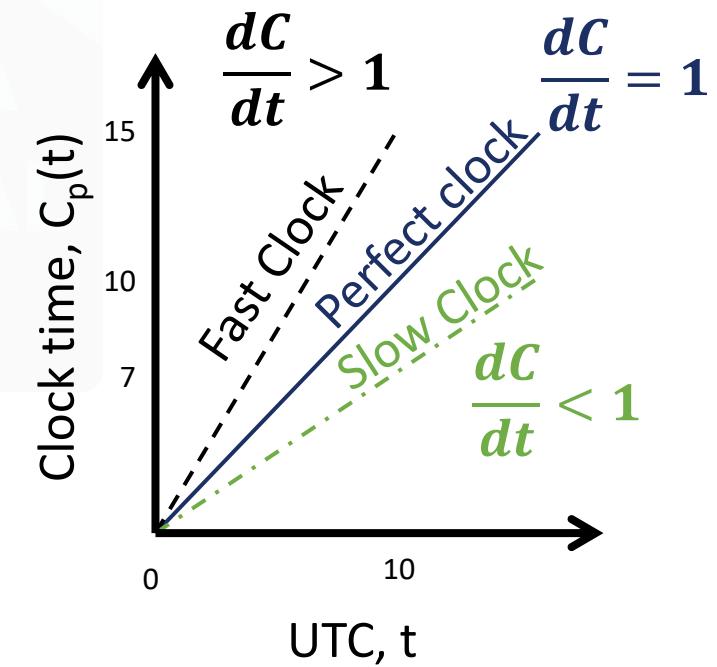
$$\text{Frequency} = dC/dt$$

- ▶ Skew of the clock is defined as the extent to which the frequency differs from that of a perfect clock

$$\text{Skew} = dC/dt - 1$$

- ▶ Hence,

$$\text{Skew} \begin{cases} > 0 & \text{for a fast clock} \\ = 0 & \text{for a perfect clock} \\ < 0 & \text{for a slow clock} \end{cases}$$



Drifting of Clock

► Synchronization techniques

1. **External Synchronization** : Synchronization with real time (external) clocks.
2. **Mutual (Internal) Synchronization** : For consistent view of time across all nodes of the system



8:01:24

Skew = +84 seconds
+84 seconds/35 days
Drift = +2.4 sec/day



Oct 23, 2016
8:00:00



8:01:48

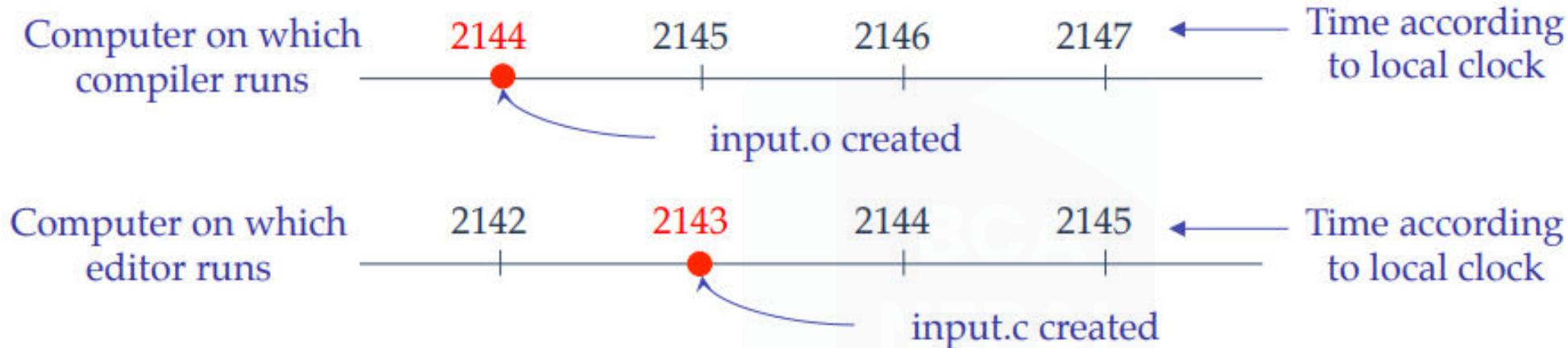
Skew = +108 seconds
+108 seconds/35 days
Drift = +3.1 sec/day

Dealing with Drift

- ▶ Assume we set computer to true time. (It is not good idea to set clock back.)
 - Illusion of time moving backwards can confuse message ordering and software development environments.
- ▶ There should be go for gradual clock correction.
 - **If fast:** Make clock run slower until it synchronizes.
 - **If slow:** Make clock run faster until it synchronizes.
- ▶ Operating System can change rate at which it requests interrupts.
 - if system requests interrupts every 14 msec but clock is too slow: request interrupts at 12 msec.
- ▶ **Software correction:** Redefine the interval.

Dealing with Drift

- In a distributed system, achieving agreement on time is not easy.
- Assume no global agreement on time. Let's see what happens:

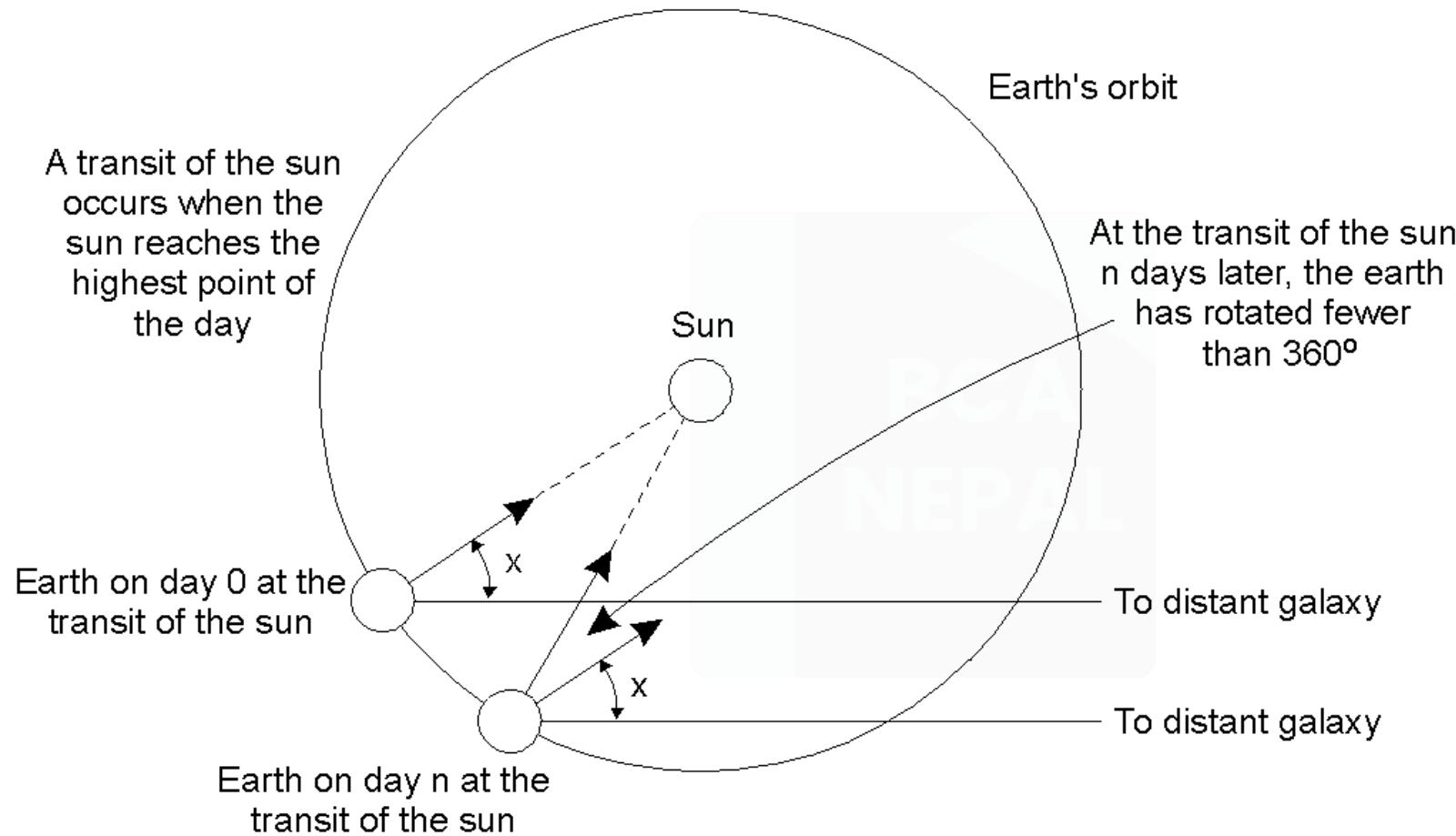


- Assume that the compiler and editor are on different machines
- `output.o` has time 2144 msec
- `output.c` is modified but is assigned time 2143 msec because the clock on its machine is slightly behind.
- `Make` will not call the compiler.
- The resulting executable will have a mixture of object files from old and new sources.

Physical Clocks

- ▶ It is impossible to guarantee that crystals in different computers all run at exactly the same frequency. This difference in time values is **clock skew**.
- ▶ “Exact” time was computed by astronomers
 - The difference between two transits of the sun is termed a solar day.
 - Divide a solar day by $24*60*60$ yields a solar second.
- ▶ However, the earth is slowing! (35 days less in a year over 300 million years)
- ▶ There are also short-term variations caused by turbulence deep in the earth’s core.
 - A large number of days (n) were used to the average day length, then dividing by 86,400 to determine the mean solar second.

Physical Clocks



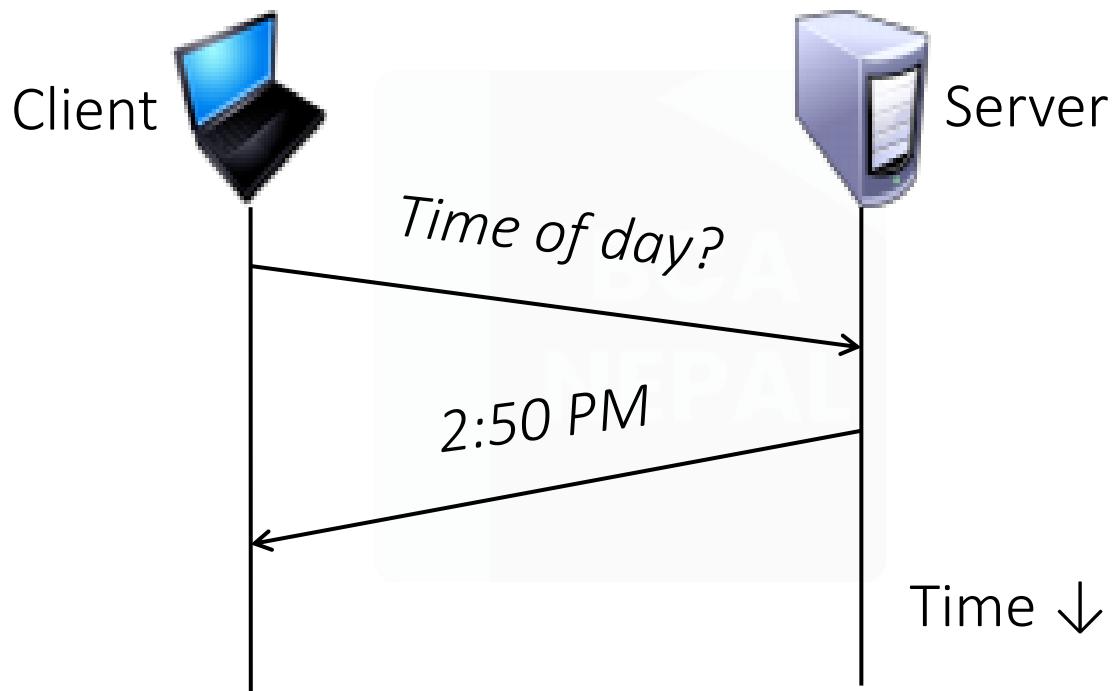
Computation of the mean solar day.

Physical Clocks

- ▶ How Computer Clocks Are Implemented ?
- ▶ A computer clock usually consists of three components
 1. **A quartz crystal** that oscillates at a well-defined frequency,
 2. **A constant register** - used to store a constant value that is decided based on the frequency of oscillation of the quartz
 3. **A counter register** - is used to keep track of the oscillations of the quartz crystal.
- ▶ The value in the counter register is **decremented by 1** for each Oscillation of the quartz crystal. When the value of the counter register **becomes zero**, an **interrupt is generated** and its value is reinitialized to the value in the constant register.
- ▶ Each interrupt is called a **clock tick**.

Synchronization to a time server

- ▶ Suppose a server with an accurate clock (e.g., GPS-receiver)
 - Could simply issue an RPC to obtain the time:



- ▶ But this doesn't account for network latency
 - Message delays will have outdated server's answer

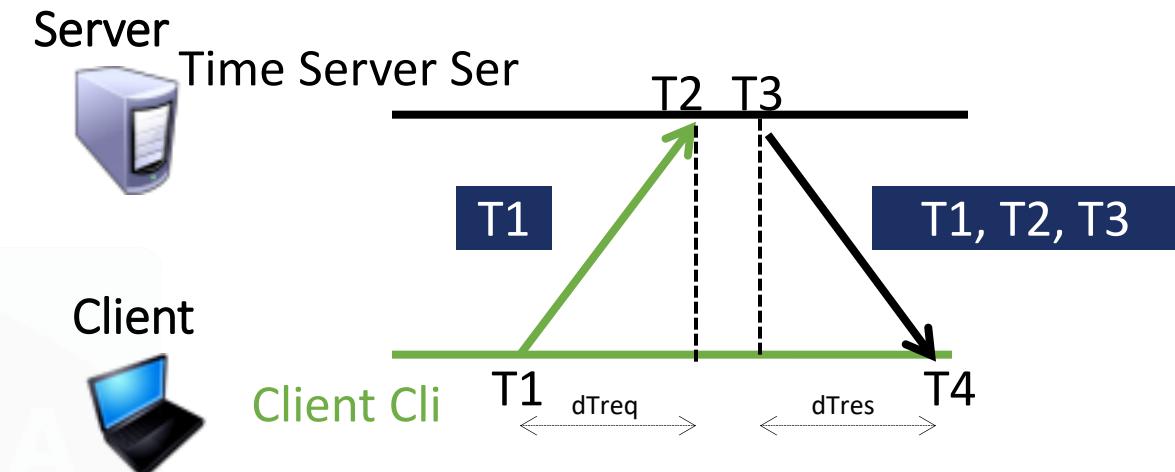
Cristian's Algorithm

- ▶ Flaviu Cristian (in 1989) provided an algorithm to synchronize networked computers with a time server
- ▶ The basic idea:
 - Identify a network time server that has an accurate source for time (e.g., the time server has a UTC receiver)
 - All the clients contact the network time server for synchronization
- ▶ However, the network delays incurred when the client contacts the time server results in outdated time
 - The algorithm estimates the network delays and compensates for it

Cristian's Algorithm

- ▶ Client Cli sends a request to Time Server Ser, time stamped its local clock time T1
- ▶ Ser will record the time of receipt T2 according to its local clock
- ▶ dTreq is network delay for request transmission
- ▶ Ser replies to Cli at its local time T3, piggybacking T1 and T2
- ▶ Cli receives the reply at its local time T4
 - dTres is the network delay for response transmission
- ▶ Now Cli has the information T1, T2, T3 and T4
- ▶ **Assuming that the transmission delay from Cli→Ser and Ser→Cli are the same**

$$T2 - T1 \approx T4 - T3$$



Berkeley Algorithm

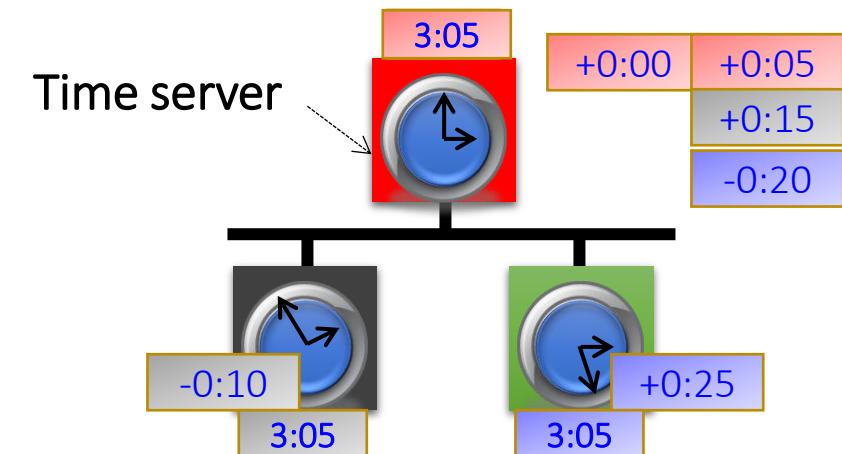
- ▶ A single time server can fail, blocking timekeeping
- ▶ The Berkeley algorithm is a **distributed algorithm** for timekeeping
- ▶ Assumes all machines have equally-accurate local clocks
- ▶ Obtains average from participating computers and synchronizes clocks to that average
- ▶ Time server **periodically sends a message** ("time=?") to all computers in the group.
- ▶ Each computer in the group sends its clock value to the server.
- ▶ Server has prior knowledge of propagation time from node to server.
- ▶ Time server readjusts the clock values of the reply messages using propagation time & then takes **fault tolerant average**.
- ▶ The time server readjusts its own time & **sends the adjustment (positive or negative)** to each node.

Berkeley Algorithm

Approach:

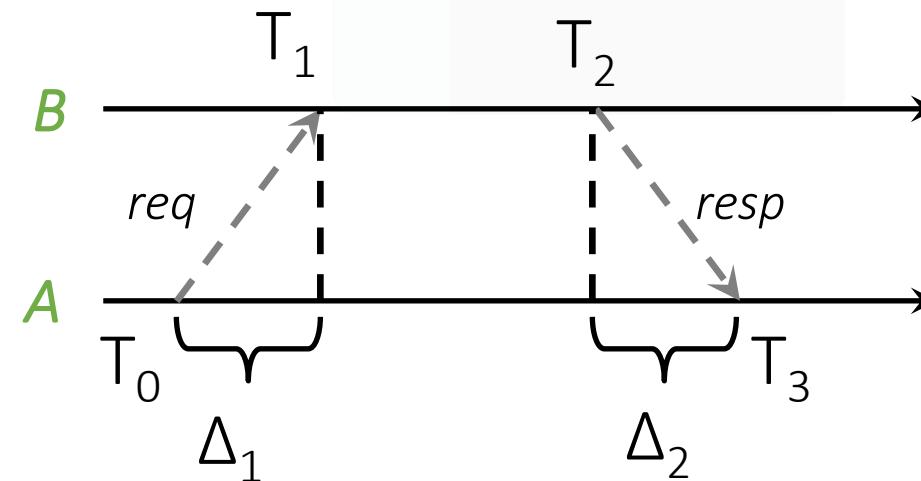
- ▶ A time server periodically (approx. once in 4 minutes) sends its time to all the computers and polls them for the time difference
- ▶ The computers compute the time difference and then reply
- ▶ The server computes an average time difference for each computer
- ▶ The server commands all the computers to update their time (by gradual time synchronization)

1. At 3:00, the time daemon tells the other machines its time and asks for theirs.
2. They respond with how far ahead or behind the time daemon they are.
3. The time daemon computes the average and tells each machine how to adjust its clock



Network Time Protocol (NTP)

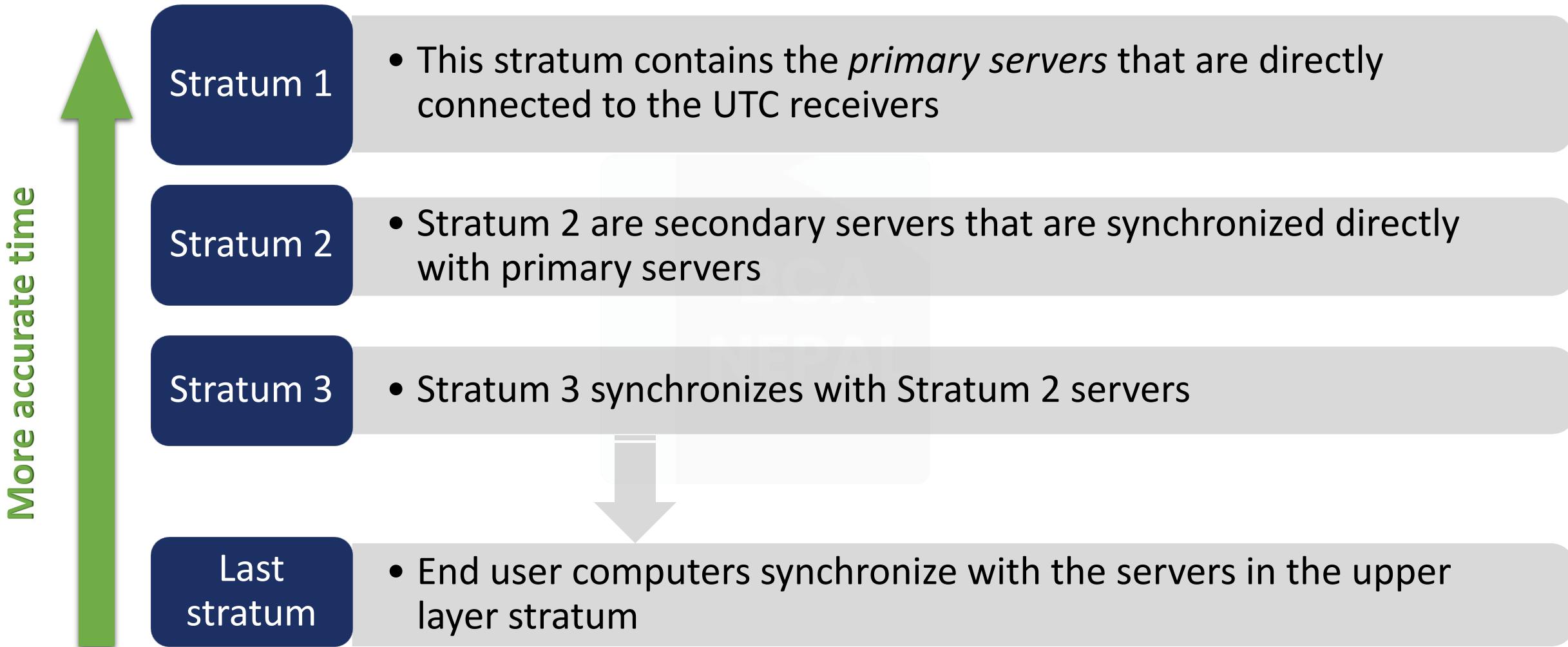
- ▶ NTP is an Application layer protocol.
 - The NTP service is provided by a network of servers located across the Internet.
 - Uses standard UDP Internet transport protocol
 - Adjust system clock as close to UTC as possible over the Internet.
 - Enable sufficiently frequently resynchronizations
 - Primary servers are connected directly to a time source (e.g. a radio clock receiving UTC, GPS) of clients and servers.
 - The servers are connected in a logical hierarchy called a synchronization subnet.



Network Time Protocol (NTP)

- ▶ NTP defines an architecture for a time service and a protocol to distribute time information over the Internet
- ▶ In NTP, servers are connected in a logical hierarchy called *synchronization subnet*
- ▶ The levels of synchronization subnet is called *strata*
 - Stratum 1 servers have most **accurate time information** (connected to a UTC receiver)
 - Servers in each stratum act as **time servers** to the servers in the lower stratum

Hierarchical organization of NTP Servers



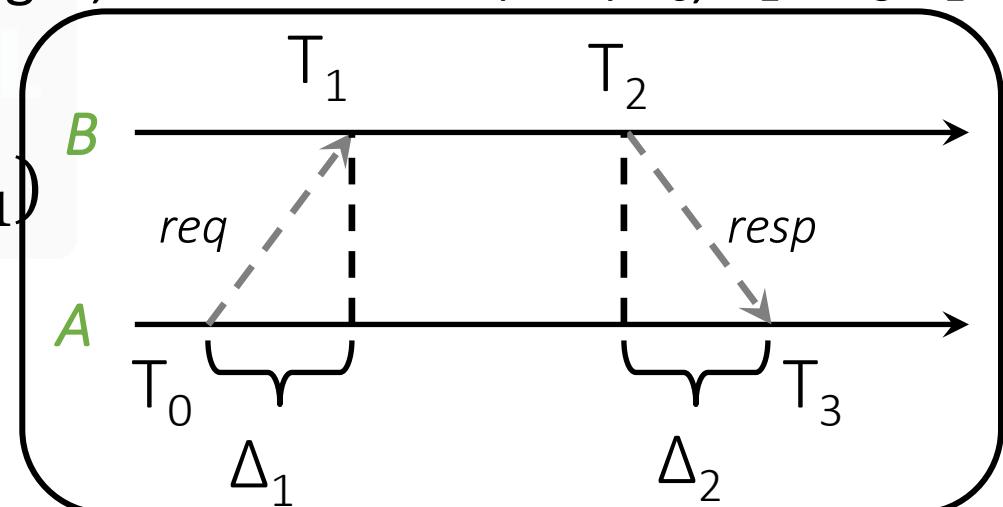
Network Time Protocol (NTP)

▶ Let's consider the following example with 2 machines A and B:

- First, A sends a request timestamped with value T_0 to B.
- B, as the message arrives, records the time of receipt T_1 from its own local clock and responds with a message timestamped with T_2 , piggybacking the previously recorded value T_1 .
- Lastly, A, upon receiving the response from B, records the arrival time T_3 .
- To account for the time delay in delivering messages, we calculate $\Delta_1 = T_1 - T_0$, $\Delta_2 = T_3 - T_2$.
- Now, an estimated offset of A relative to B is:

Roundtrip Delay: $d = (T_3 - T_0) + (T_2 - T_1)$

Time offset: $t = \frac{(T_1 - T_0) + (T_2 - T_3)}{2}$



- Based on t , we can either slow down the clock of A or fasten it so that the two machines can be synchronized with each other.

Logical Clocks

- ▶ If two machines do not interact, there is no need to synchronize them.
 - What usually matters is that processes agree on the order in which events occur rather than the time at which they occurred
- ▶ Many times, it is sufficient if processes agree on the order in which the events have occurred in a DS
 - For example, for a distributed make utility, it is sufficient to know if an input file was modified before or after its object file
- ▶ Logical clocks are used to define an order of events without measuring the physical time at which the events occurred
- ▶ Two types of logical clocks:
 1. Lamport's Logical Clock (or simply, Lamport's Clock)
 2. Vector Clock

Lamport's Logical Clock

- ▶ Lamport advocated maintaining logical clocks at the processes to keep track of the order of events
- ▶ To synchronize logical clocks, Lamport defined a relation called “**happened-before**”
- ▶ The expression **a → b** (reads as “a happened before b”) means that all entities in a DS agree that event a occurred before event b

The Happened-before Relation

- ▶ The happened-before relation can be observed directly in two situations:
 1. If **a** and **b** are events in the same process, and **a** occurs before **b**, then $a \rightarrow b$ is true
 2. If **a** is an event of message **m** being sent by a process, and **b** is the event of **m** (i.e., the same message) being received by another process, then $a \rightarrow b$ is true
- ▶ The happened-before relation is transitive
 - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

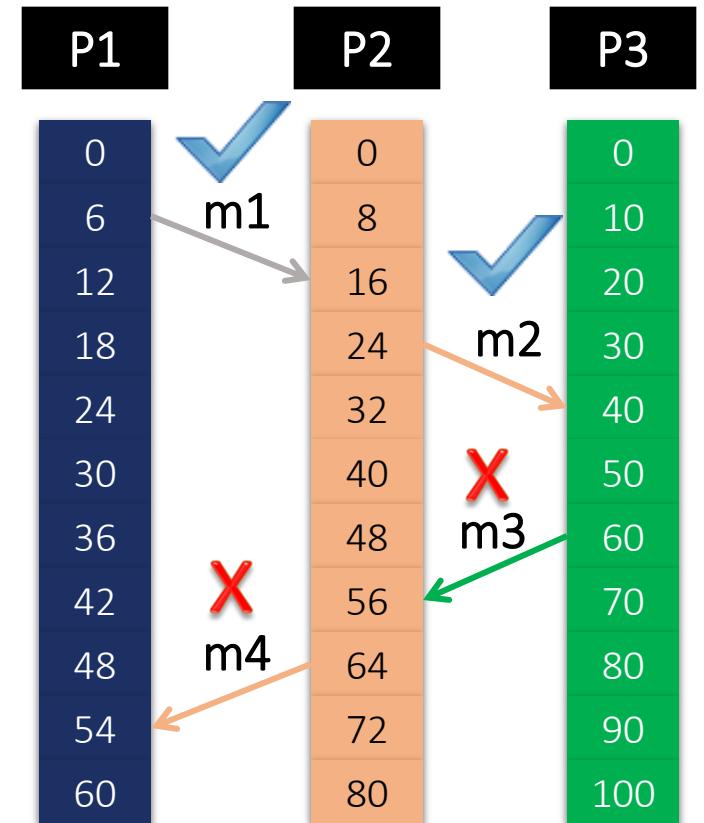
Properties of Logical Clock

From the happened-before relation, we can infer that:

- ▶ If two events a and b occur within the same process and $a \rightarrow b$, then $C(a)$ and $C(b)$ are assigned time values such that $C(a) < C(b)$
- ▶ If a is the event of sending message m from one process (say P1), and b is the event of receiving m (i.e., the same message) at another process (say, P2), then:
 - The time values $C_1(a)$ and $C_2(b)$ are assigned in a way such that the two processes agree that $C_1(a) < C_2(b)$
- ▶ The clock time C must always go forward (increasing), and never backward (decreasing)

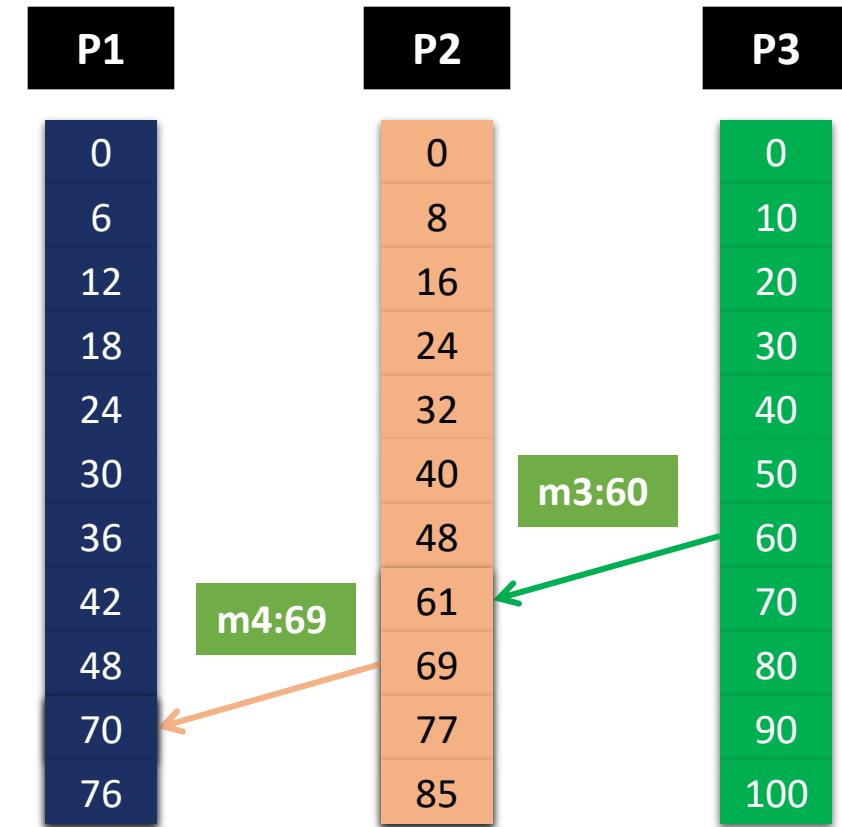
Synchronizing Logical Clocks

- ▶ Three processes P1, P2 and P3 running at different rates
- ▶ If the processes communicate between each other, there might be discrepancies in agreeing on the event ordering
 - Ordering of sending and receiving messages m1 and m2 are correct
 - However, m3 and m4 violate the happened-before relationship



Lamport's Clock Algorithm

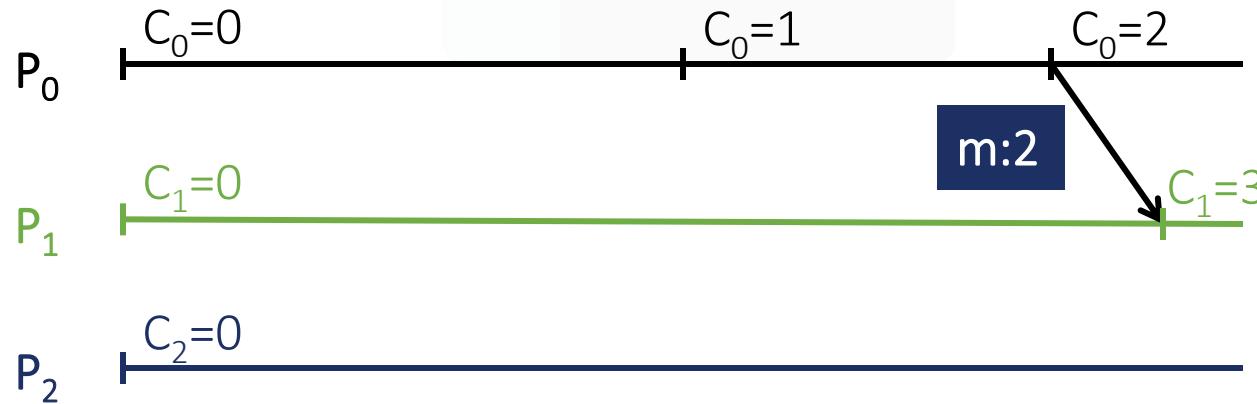
- ▶ When a message is being sent:
 - Each message carries a timestamp according to the sender's logical clock
- ▶ When a message is received:
 - If the receiver logical clock is less than the message sending time in the packet, then adjust the receiver's clock such that:
currentTime = timestamp + 1



Implementation of Lamport's Clock

- ▶ Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

1. For any two successive events that take place within P_i , C_i is incremented by 1
2. Each time a message m is sent by process P_i , m is assigned a timestamp $ts(m) = C_i$
3. Whenever a message m is received by a process P_j , P_j adjusts its local counter C_j to $\max(C_j, ts(m)) + 1$



Vector Clocks

- ▶ Vector clock was proposed to overcome the limitation of Lamport's clock
 - The property of *inferring* that **a** occurred before **b** is known as the **causality** property
- ▶ A vector clock for a system of **N** processes is an array of **N** integers
- ▶ Every process P_i stores its own vector clock VC_i
 - Lamport's time values for events are stored in VC_i
 - $VC_i(a)$ is assigned to an event **a**
- ▶ If $VC_i(a) < VC_i(b)$, then we can infer that $a \rightarrow b$ (or more precisely, that event **a** *causally preceded* event **b**)

Updating Vector Clocks

- ▶ Vector clocks are constructed as follows:

1. $VC_i[i]$ is the number of events that have occurred at process P_i so far

- $VC_i[i]$ is the local logical clock at process P_i

Increment VC_i whenever a new event occurs

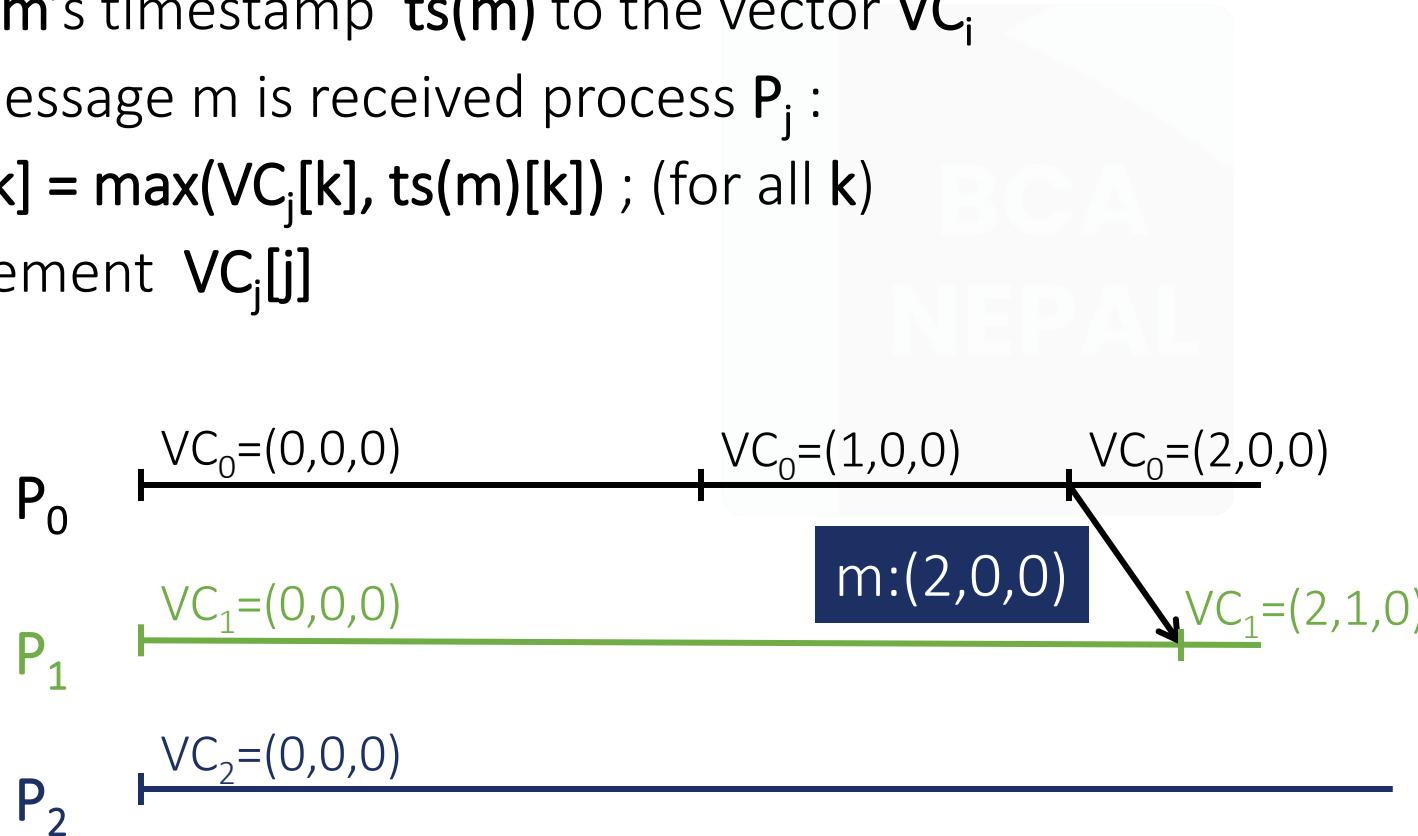
2. If $VC_i[j] = k$, then P_i knows that k events have occurred at P_j

- $VC_i[j]$ is P_i 's knowledge of the local time at P_j

Pass VC_j along with the message

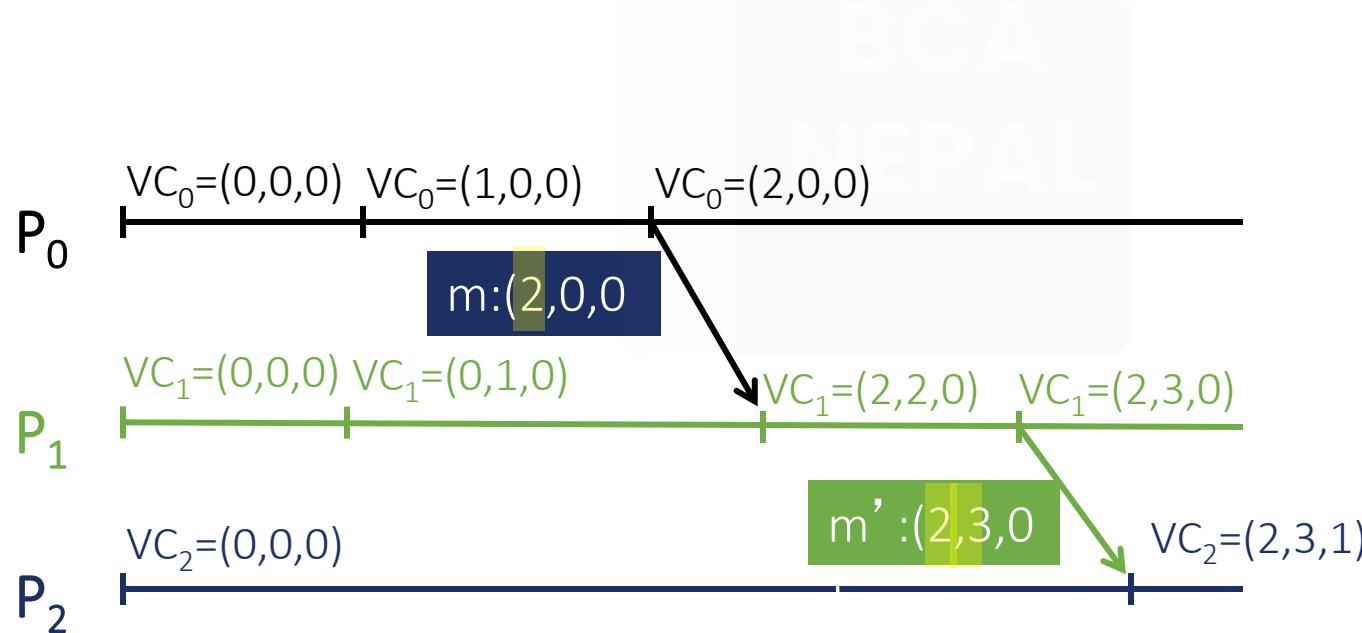
Vector Clock Update Algorithm

- ▶ Whenever there is a new event at P_i , increment $VC_i[i]$
- ▶ When a process P_i sends a message m to P_j :
 - Increment $VC_i[i]$
 - Set m 's timestamp $ts(m)$ to the vector VC_i
- ▶ When message m is received process P_j :
 - $VC_j[k] = \max(VC_j[k], ts(m)[k])$; (for all k)
 - Increment $VC_j[j]$



Inferring Events with Vector Clocks

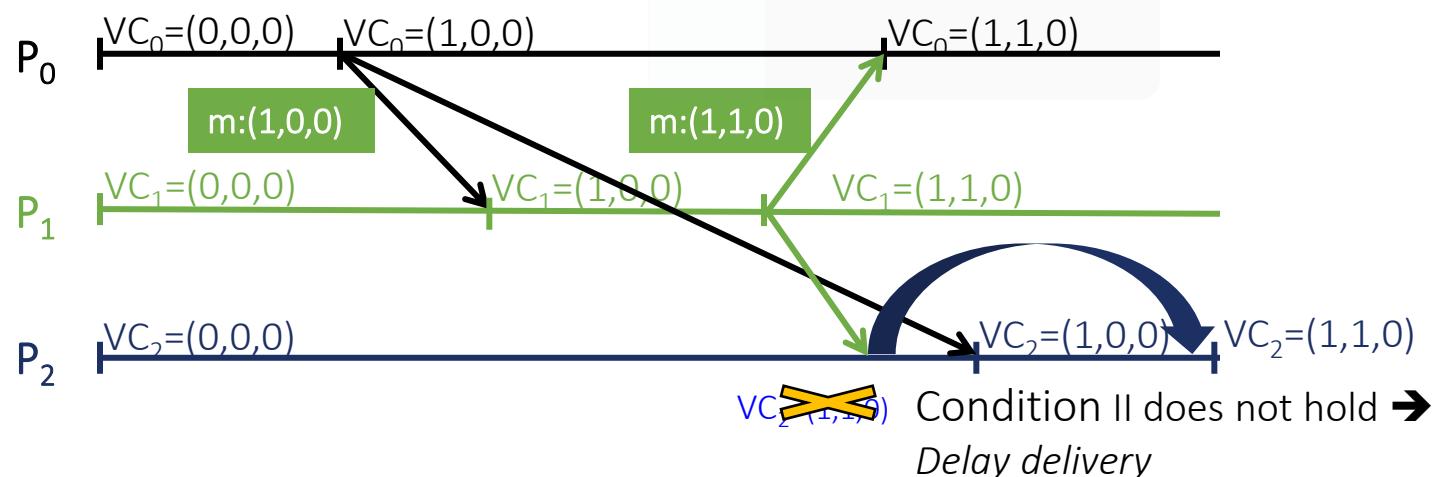
- Let a process P_i send a message m to P_j with timestamp $ts(m)$, then:
 - P_j knows the number of events at the sender P_i that causally precede m
 - $(ts(m)[i] - 1)$ denotes the number of events at P_i
 - P_j also knows the minimum number of events at other processes P_k that causally precede m
 - $(ts(m)[k] - 1)$ denotes the minimum number of events at P_k



Enforcing Causal Communication

- ▶ Assume that messages are *multicast* within a group of processes, P_0 , P_1 and P_2
- ▶ To enforce **causally-ordered multicasting**, the delivery of a message m sent from P_i to P_j can be delayed until the following two conditions are met:
 - $ts(m)[i] = VC_j[i] + 1$ (Condition I)
 - $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$ (Condition II)

Assuming that P_i only increments $VC_i[i]$ upon sending m and adjusts $VC_i[k]$ to $\max\{VC_i[k], ts(m)[k]\}$ for each k upon receiving a message m'



Mutual Exclusion

- ▶ In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs.
- ▶ In distributed systems, since there is no shared memory, these methods cannot be used.
- ▶ Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
- ▶ Only one process is allowed to execute the critical section (CS) at any given time.
- ▶ In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- ▶ Message passing is the sole means for implementing distributed mutual exclusion

Critical Section

- ▶ The sections of a program that need exclusive access to shared resources are referred to as critical sections.
- ▶ Mutual Exclusion : Prevent processes from executing concurrently with their associated critical section.

Requirement :

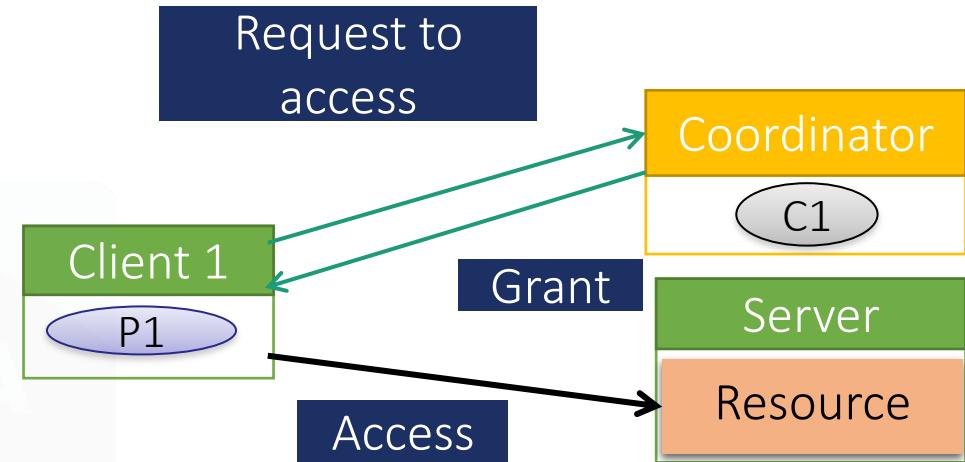
- ▶ **Mutual exclusion:** At anytime only one process should access the resource.
 - A process can get another resource first releasing its own.
- ▶ **No starvation:** if every process that is granted the resource eventually release it, every request must be eventually granted.

Types of Distributed Mutual Exclusion

Mutual exclusion algorithms are classified into two categories

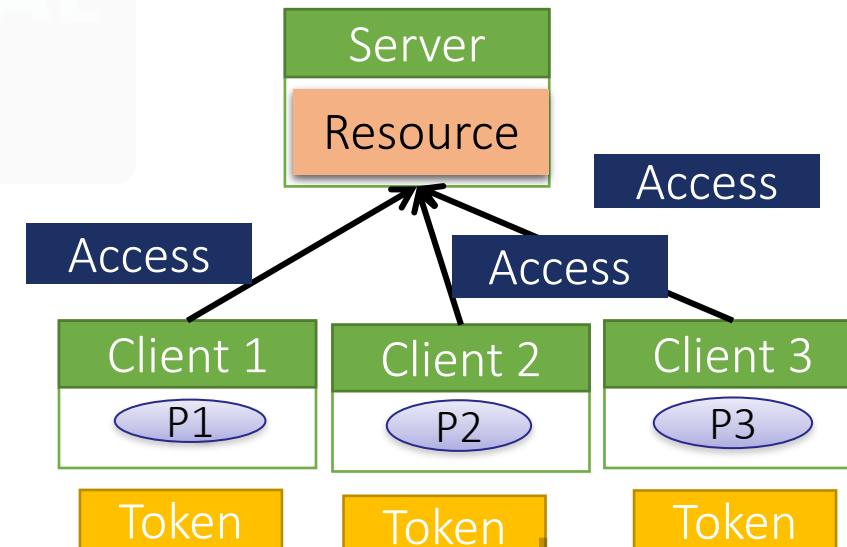
1. Permission-based Approaches

- ▶ A process, which wants to access a shared resource, requests the permission from one or more coordinators



2. Token-based Approaches

- ▶ Each shared resource has a token
- ▶ Token is circulated among all the processes
- ▶ A process can access the resource if it has the token



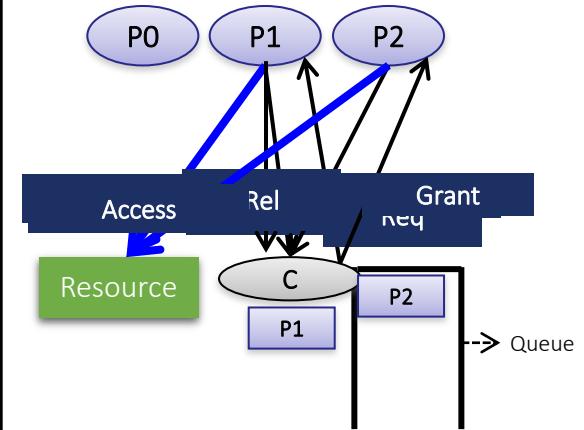
Permission-based Approaches

- ▶ There are two types of permission-based mutual exclusion algorithms
 1. **Centralized Algorithms**
 2. **Decentralized Algorithms**



Centralized Mutual Exclusion Algorithm

- ▶ One process is *elected* as a coordinator (**C**) for a shared resource
- ▶ Coordinator maintains a **Queue** of access requests
- ▶ Whenever a process wants to access the resource, it sends a request message to the coordinator to access the resource
- ▶ When the coordinator receives the request:
 - If no other process is currently accessing the resource, it grants the permission to the process by sending a “grant” message
 - If another process is accessing the resource, the coordinator queues the request, and does not reply to the request
- ▶ The process in action releases the exclusive access after accessing the resource
- ▶ Afterwards, the coordinator sends the “grant” message to the next process in the queue

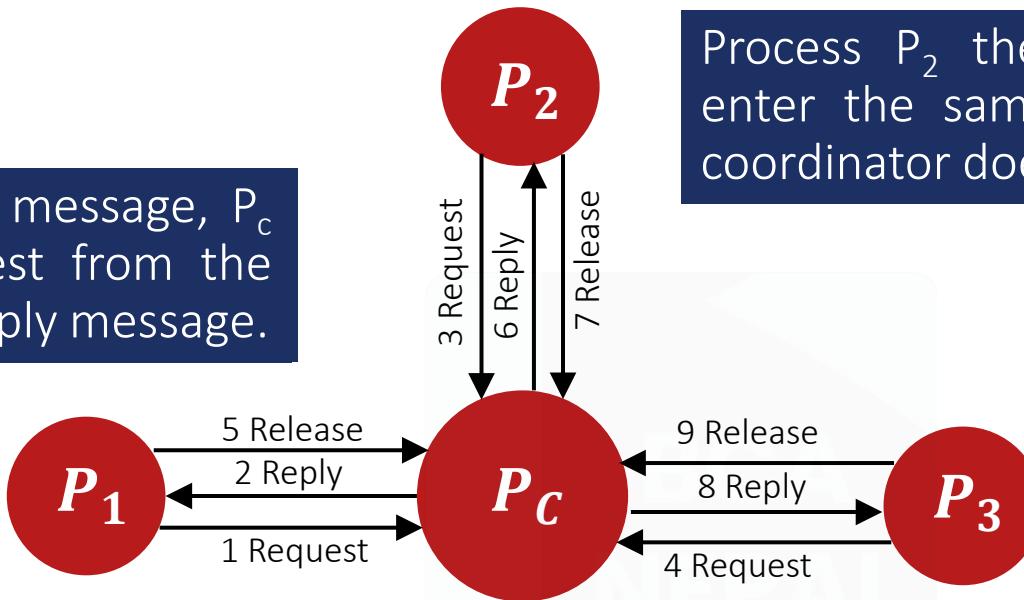


Centralized Mutual Exclusion Algorithm

- ▶ Coordinator coordinates entry to critical section.
- ▶ Allows only one process to enter critical section.
- ▶ Ensures no starvation as uses first come, first served policy.
- ▶ Simple implementation
- ▶ Three messages per critical section
 1. Request
 2. Reply
 3. Release
- ▶ Single point of failure & performance bottleneck.

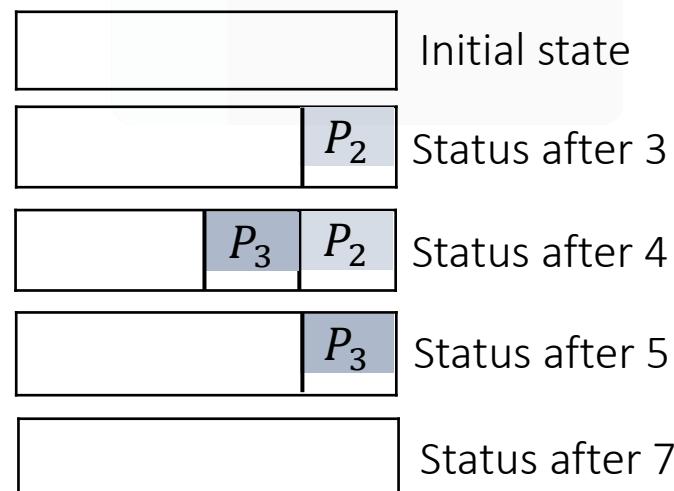
Centralized Mutual Exclusion Algorithm

On receiving release message, P_c takes the first request from the queue and sends a reply message.



Process P_2 then asks permission to enter the same critical region. The coordinator does not reply.

Process P_1 wants to enter a critical section for which it sends a request to P_c



A Centralized Algorithm – Advantages and Disadvantages

Advantages

- ▶ **Flexibility** : Blocking versus non-blocking requests
 - The coordinator can block the requesting process until the resource is free
 - Or, the coordinator can send a “permission-denied” message back to the process
- ▶ **Simplicity** : The algorithm guarantees mutual exclusion, and is simple to implement

Disadvantages

- ▶ **Fault-Tolerance Deficiency** : Centralized algorithm is vulnerable to a single-point of failure (at coordinator)
 - If processes normally block after making a request, they cannot distinguish a dead coordinator from “permission denied” since no message comes back.
- ▶ **Performance Bottleneck** : In a large-scale system, single coordinator can be overwhelmed with requests

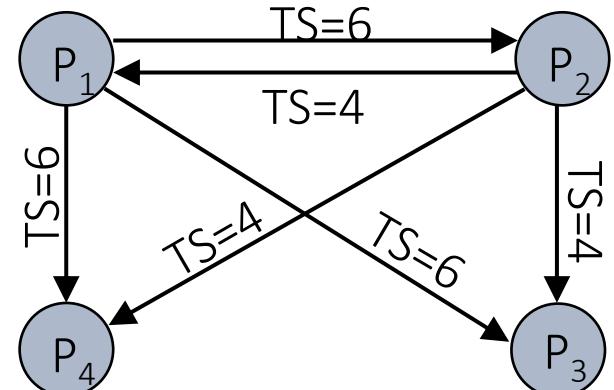
Distributed Mutual Exclusion Algorithm

- ▶ When a process wants to enter the **CS (critical section)** , it sends a request message to all other processes, and when it receives reply from all processes, then only it is allowed to enter the CS.
- ▶ The request message contains following information:
 - Process identifier
 - Name of CS
 - Unique time stamp generated by process for request message

Distributed Mutual Exclusion Algorithm

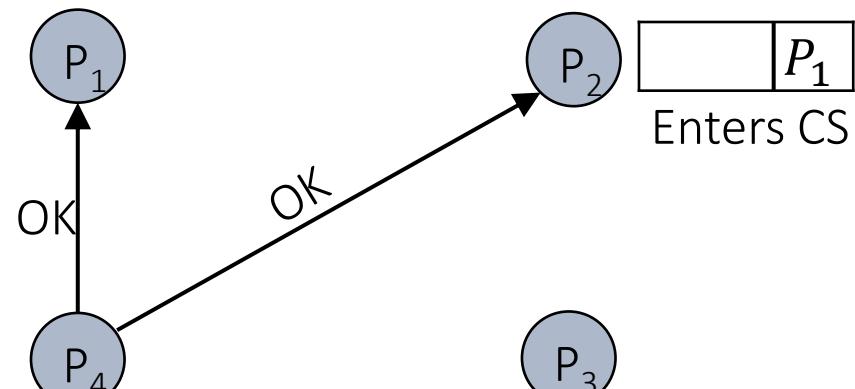
- ▶ The decision whether receiving process replies immediately to a request message or defers its reply is based on three cases:
 1. If **receiver** process is in its **critical section**, then it **defers its reply**.
 2. If **receiver** process does **not want to enter its critical section**, then it **immediately sends a reply**.
 3. If **receiver** process itself is **waiting to enter critical section**, then it compares its own **request timestamp** with the **timestamp in request message**
 - If its **own request timestamp** is **greater** than **timestamp in request message**, then it **sends a reply immediately**.
 - Otherwise, the **reply is deferred**.

Distributed Mutual Exclusion Algorithm



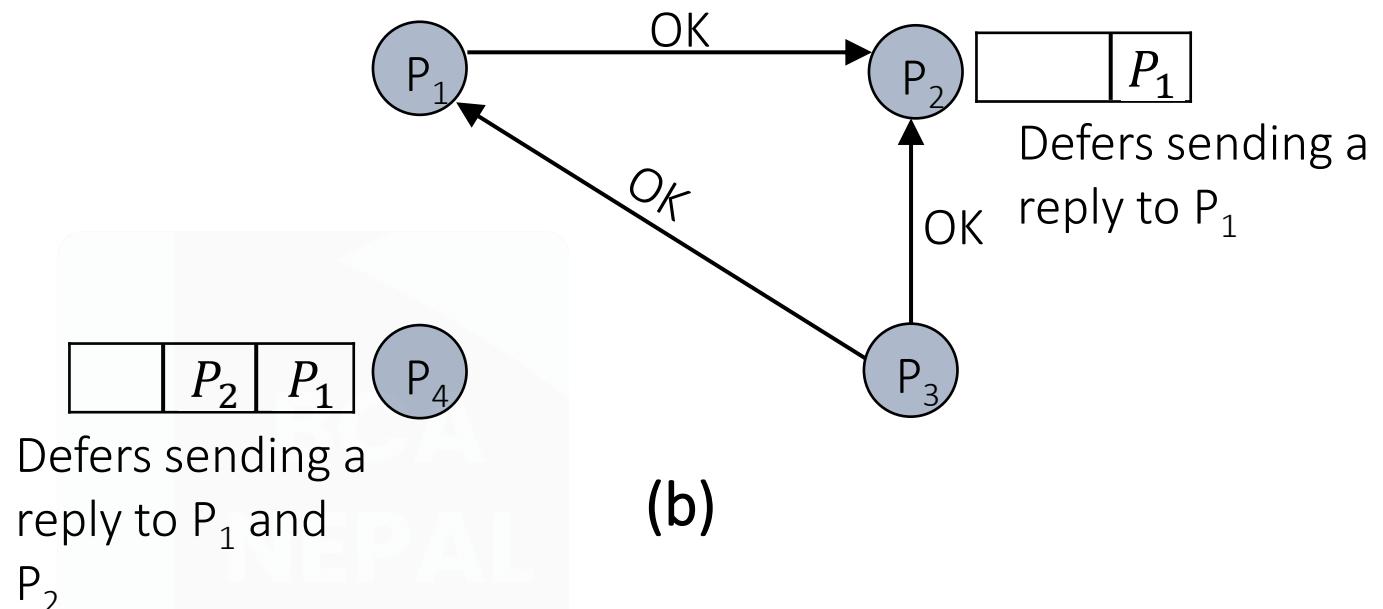
Already in CS

(a)

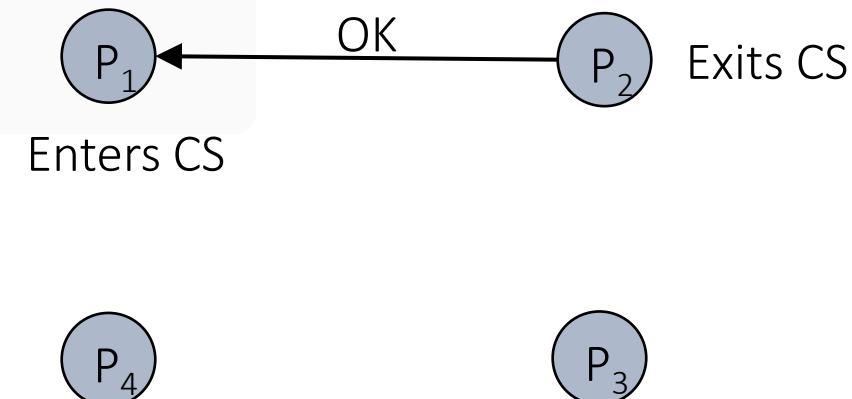


Enters CS

(c)



(b)



(d)

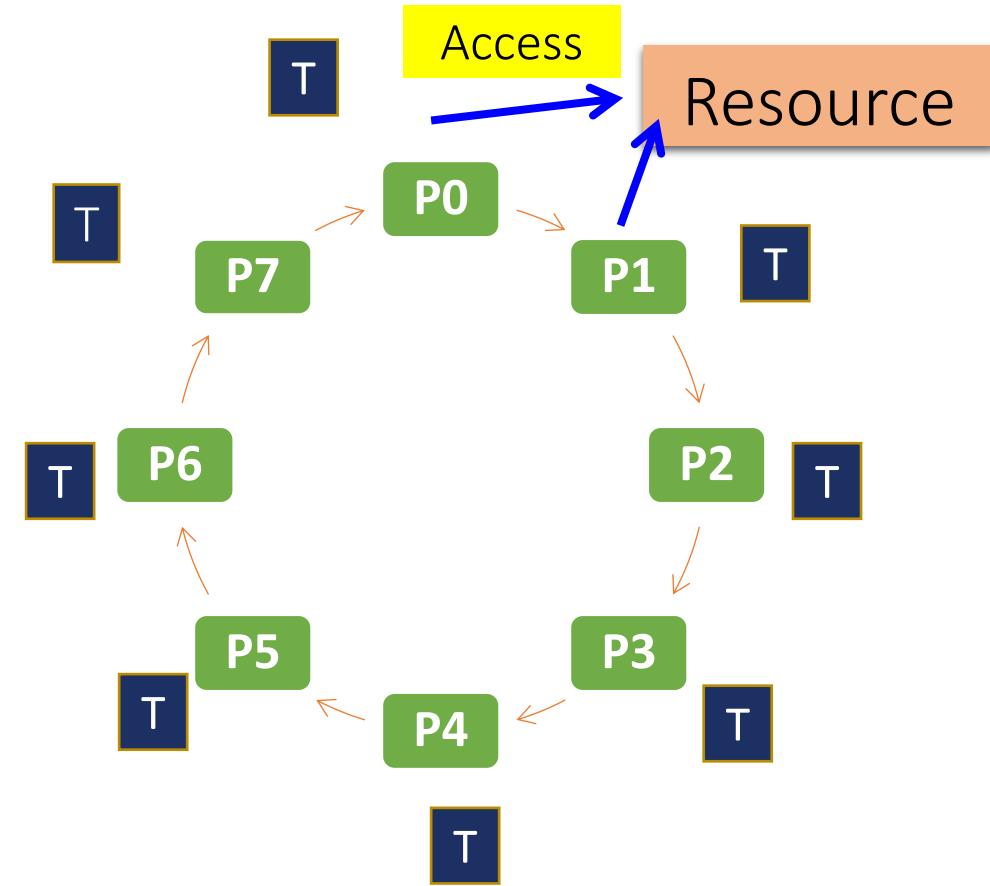
A Token Ring Algorithm

With a token ring algorithm:

- Each resource is associated with a token
- The token is circulated among the processes
- The process with the token can access the resource

How is the token circulated among processes?

- All processes form a logical ring where each process knows its next process
- One process is given the token to access the resource
- The process with the token has the right to access the resource
- If the process has finished accessing the resource OR does not want to access the resource:
 - It passes the token to the next process in the ring



Failure in Token Ring Algorithm

Two types of failure can occur:

1. Process failure
2. Lost token



Failure in Token Ring Algorithm

► **Process failure:** A process failure in the system causes the logical ring to break.

- Requires detection of failed process & dynamic reconfiguration of logical ring.
- Process receiving token sends back acknowledgement to neighbor.
- When a process detects failure of neighbor, it removes failed process by skipping it & passing token to process after it.
- When a process becomes alive after recovery, It informs the neighbour previous to it so that it gets the token during the next round of circulation.

► **Lost token:** If the token is lost, a new token must be generated.

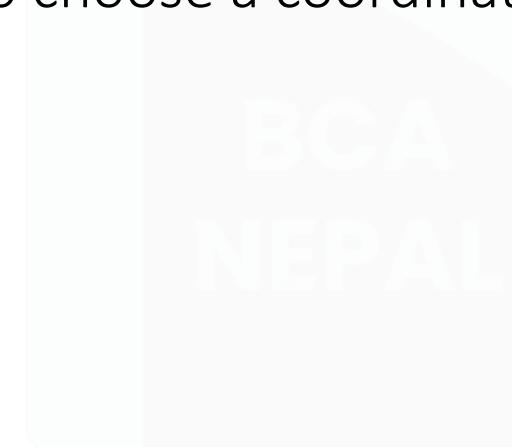
- Must have mechanism to detect & regenerate token.
- Designate process as monitor. Monitor periodically circulates “who has token”.
- Owner of token writes its process identifier in message & passes on.
- On receipt of message, monitor checks process identifier field. If empty generate new token & passes it.
- Multiple monitors can be used.

Comparison of Mutual Exclusion Algorithms

Algorithm	Number of messages required for a process to access and release the shared resource	Delay before a process can access the resource (in message times)	Problems
Centralized	3	2	<ul style="list-style-type: none">Coordinator crashes
Decentralized	$2(n-1)$	$2(n-1)$	<ul style="list-style-type: none">Crash of any processLarge number of messages
TokenRing	1 to ∞	0 to $(n-1)$	<ul style="list-style-type: none">Token may be lostRing can cease to exist since processes crash

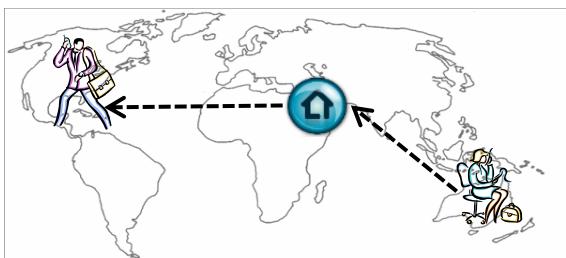
Need for a Coordinator

- ▶ Many algorithms used in distributed systems require a coordinator
 - For example, see the centralized mutual exclusion algorithm.
- ▶ In general, all processes in the distributed system are equally suitable for the role
- ▶ Election algorithms are designed to choose a coordinator.

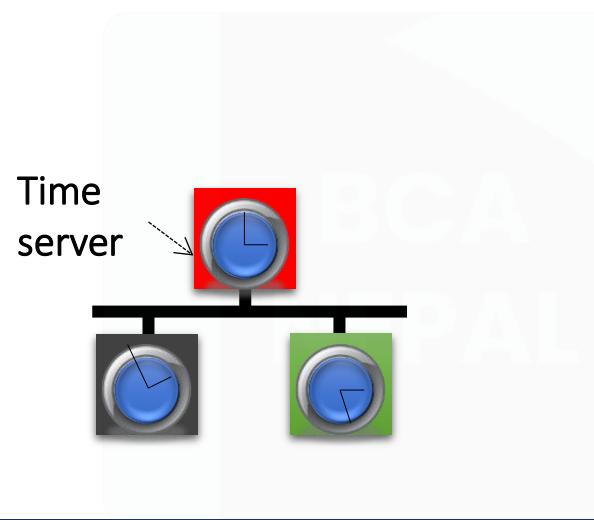


Election in Distributed Systems

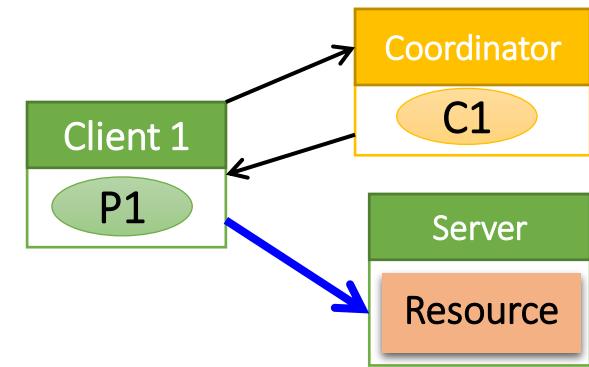
- ▶ Many distributed algorithms require one process to act as a coordinator
- ▶ Typically, it does not matter which process is elected as the coordinator



Home Node
Selection in Naming



Berkeley Clock
Synchronization Algorithm



A Centralized Mutual
Exclusion Algorithm

Election Algorithms

- ▶ What is Election?
- ▶ In a group of processes, elect a leader to undertake special tasks.
- ▶ What happens when a **leader** fails (crashes)
 - Some process detects this (how?)
 - Then what?
- ▶ Any process can **serve as coordinator**
- ▶ Any process can “call an election” (initiate the algorithm to choose a **new coordinator**).
 - There is no harm (other than extra message traffic) in having multiple concurrent elections.
- ▶ Elections may be needed when the system is initialized, or if the coordinator crashes or retires.
- ▶ An algorithm for choosing a unique process to play a particular role.



Assumptions

- ▶ Any process can call for an election.
- ▶ A process can call for at most one election at a time.
- ▶ Every process/site has a unique ID; e.g.
 - The network address
 - A process number
- ▶ Every process in the system should know the values in the set of ID numbers, although not which processors are up or down.
- ▶ The process with the highest ID number will be the new coordinator.
- ▶ Multiple processes can call an election simultaneously.
 - All of them together must yield a single leader only
 - The result of an election should not depend on which process calls for it.
- ▶ Messages are eventually delivered.
- ▶ Process groups (as with ISIS toolkit or MPI) satisfy these requirements.

Election Algorithms

1. Bully Algorithm
2. Ring Algorithm



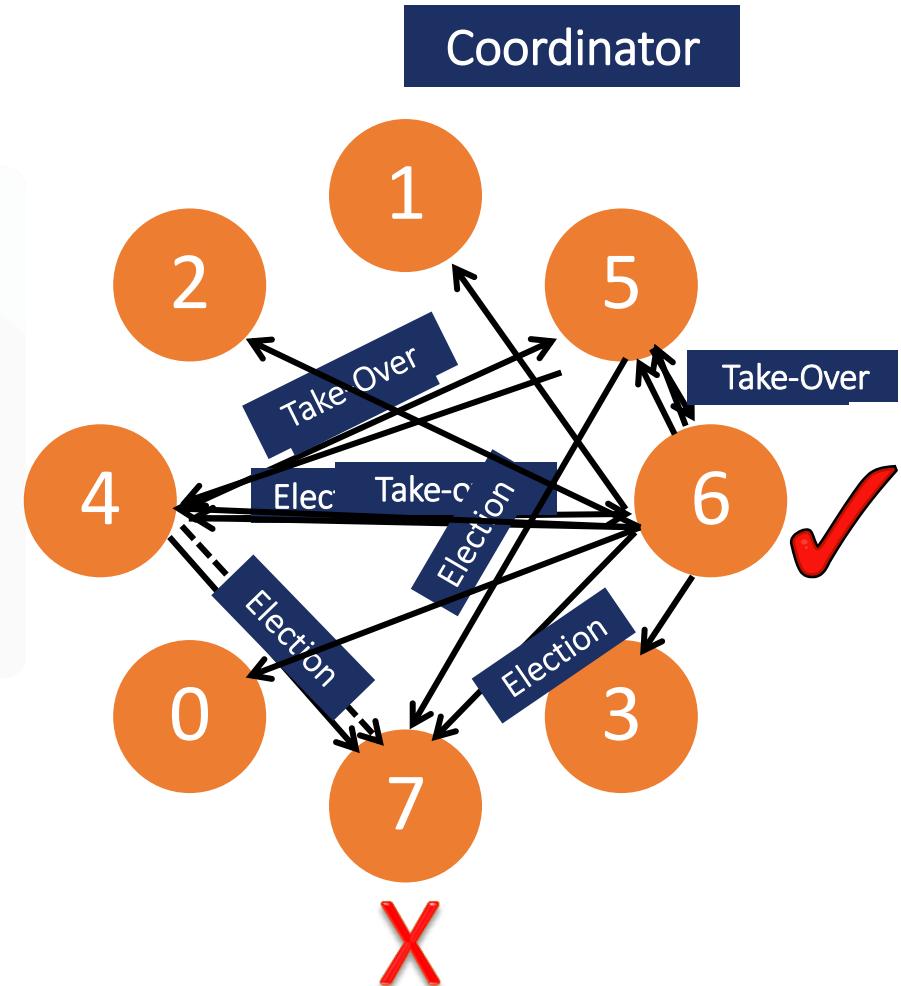
Bully Algorithm

- ▶ Bully algorithm specifies the process with the **highest identifier** will be the **coordinator** of the group. It works as follows:
- ▶ When a process p detects that the coordinator is not responding to requests, it initiates an election:
 - p sends an **election message** to all processes with **higher numbers**.
 - If **nobody responds**, then p **wins and takes over**.
 - If **one** of the processes **answers**, then p's job is done.
- ▶ If a process receives an election message from a lower-numbered process at any time, it:
 - sends an **OK** message back.
 - holds an **election** (unless its already holding one).
- ▶ A process announces its victory by sending all processes a message telling them that it is the new coordinator.
- ▶ If a process that has been down recovers, it holds an election.

Bully Algorithm

- ▶ A process (say, P_i) initiates the election algorithm when it notices that the existing coordinator is not responding
- ▶ Process P_i calls for an election as follows:

1. P_i sends an “**Election**” message to all processes with higher process IDs
2. When process P_j with $j > i$ receives the message, it responds with a “**Take-over**” message. P_i no more contests in the election
 - Process P_j re-initiates another call for election. Steps 1 and 2 continue
3. If no one responds, P_i wins the election. P_i sends “**Coordinator**” message to every process



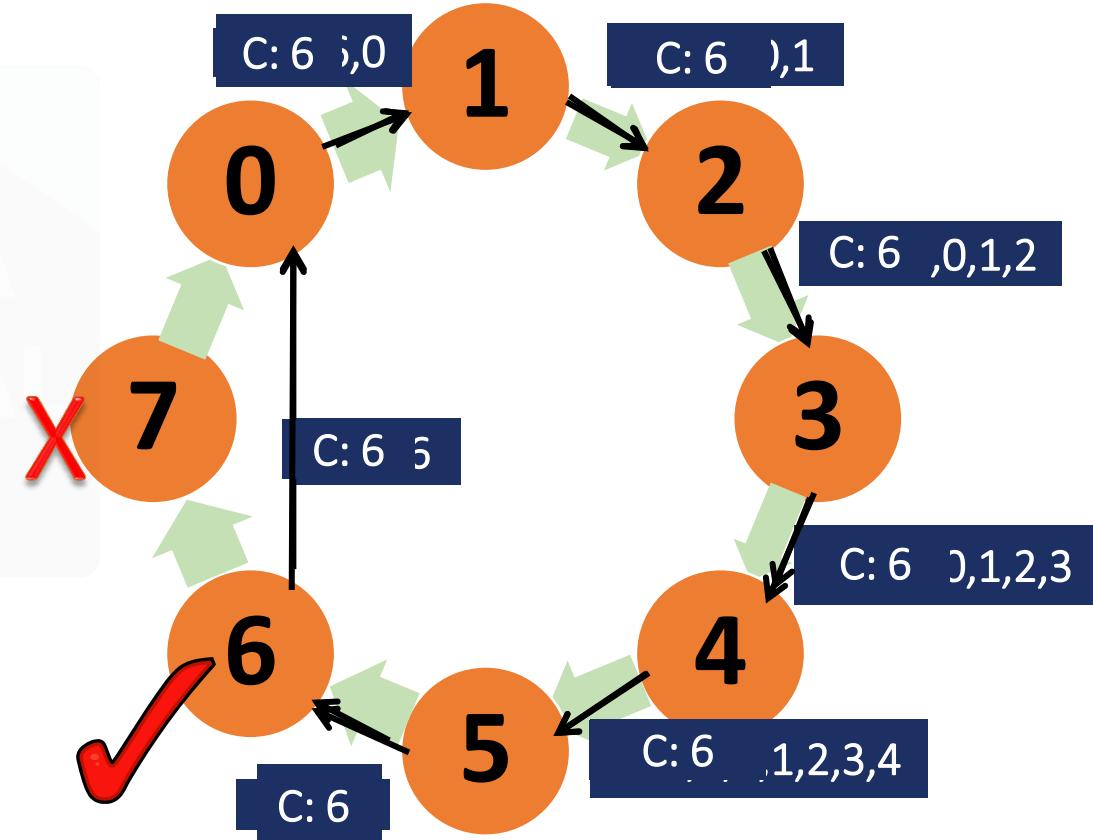
Ring Algorithm

- ▶ The ring algorithm assumes that the processes are arranged in a logical ring and each process is knowing the order of the ring of processes.
- ▶ If any process detects failure, it constructs an election message with its process ID and sends it to its neighbor.
- ▶ If the neighbor is down, the process skips over it and sends the message to the next process in the ring.
- ▶ This process is repeated until a running process is located.
- ▶ At each step, the process adds its own process ID to the list in the message and sends the message to its living neighbor.
- ▶ Eventually, the election message comes back to the process that started it.
- ▶ The process then picks either the highest or lowest process ID in the list and sends out a message to the group informing them of the new coordinator.

Ring Algorithm

- ▶ This algorithm is generally used in a ring topology
- ▶ When a process P_i detects that the coordinator has crashed, it initiates the election algorithm

1. P_i builds an “**Election**” message (E), and sends it to its next node. It inserts its ID into the Election message
2. When process P_j receives the message, it appends its ID and forwards the message
 - If the next node has crashed, P_j finds the next alive node
3. When the message gets back to P_i :
 - P_i elects the process with the **highest ID** as coordinator
 - P_i changes the message type to a “**Coordination**” message (C) and triggers its circulation in the ring



Comparison of Election Algorithms

Algorithm	Number of Messages for Electing a Coordinator	Problems
Bully Algorithm	$O(n^2)$	Large message overhead
Ring Algorithm	$2n$	An overlay ring topology is necessary

Assume that: n = Number of processes in the distributed system

Summary of Election Algorithms

- ▶ Election algorithms are used for choosing a *unique* process that will coordinate certain activities
- ▶ At the end of an election algorithm, all nodes should uniquely identify the coordinator
- ▶ We studied two algorithms for performing elections:
 - **Bully algorithm**
 - Processes communicate in a distributed manner to elect a coordinator
 - **Ring algorithm**
 - Processes in a ring topology circulate election messages to choose a coordinator

Decentralized Algorithm

- ▶ To avoid the drawbacks of the centralized algorithm, Lin et al. (2005) advocated a decentralized mutual exclusion algorithm

Assumptions:

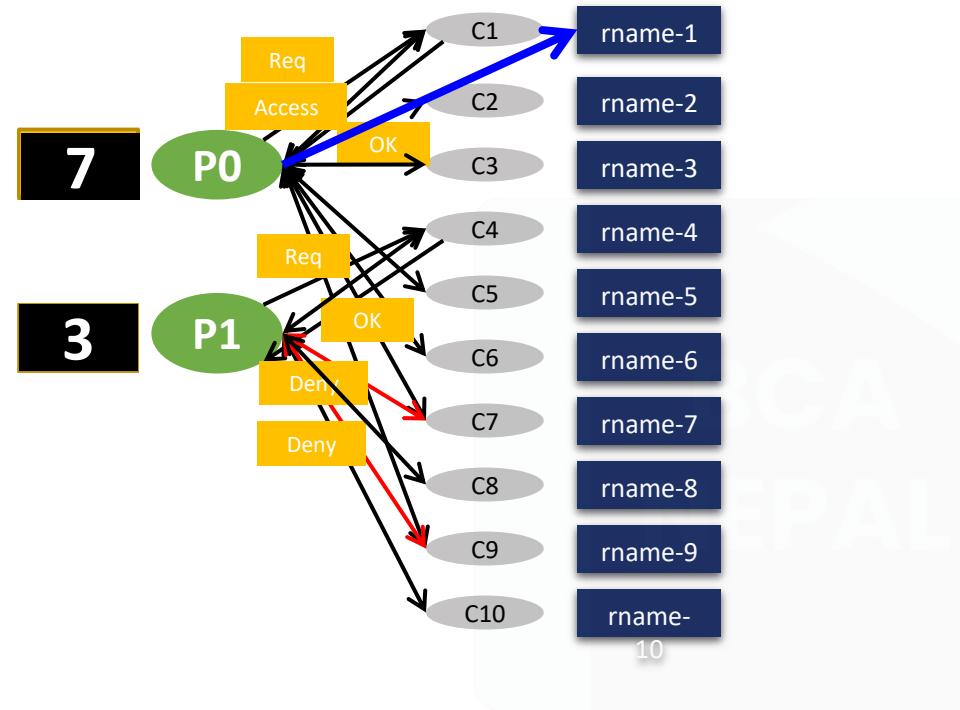
- ▶ Distributed processes are in a Distributed Hash Table (DHT) based system
- ▶ Each resource is replicated n times
 - The i^{th} replica of a resource **rname** is named as **rname-i**
- ▶ Every replica has its own coordinator for controlling access
 - The coordinator for **rname-i** is determined by using a hash function

Approach:

- ▶ Whenever a process wants to access the resource, it will have to get a majority vote from $m > n/2$ coordinators
- ▶ If a coordinator does not want to vote for a process
 - It will send a “permission-denied” message to the process

Decentralized Algorithm

- If $n=10$ and $m=7$, then a process needs at-least 7 votes to access the resource



rname-x = xth replica of a resource rname

Cj = Coordinator j

Pi = Process i

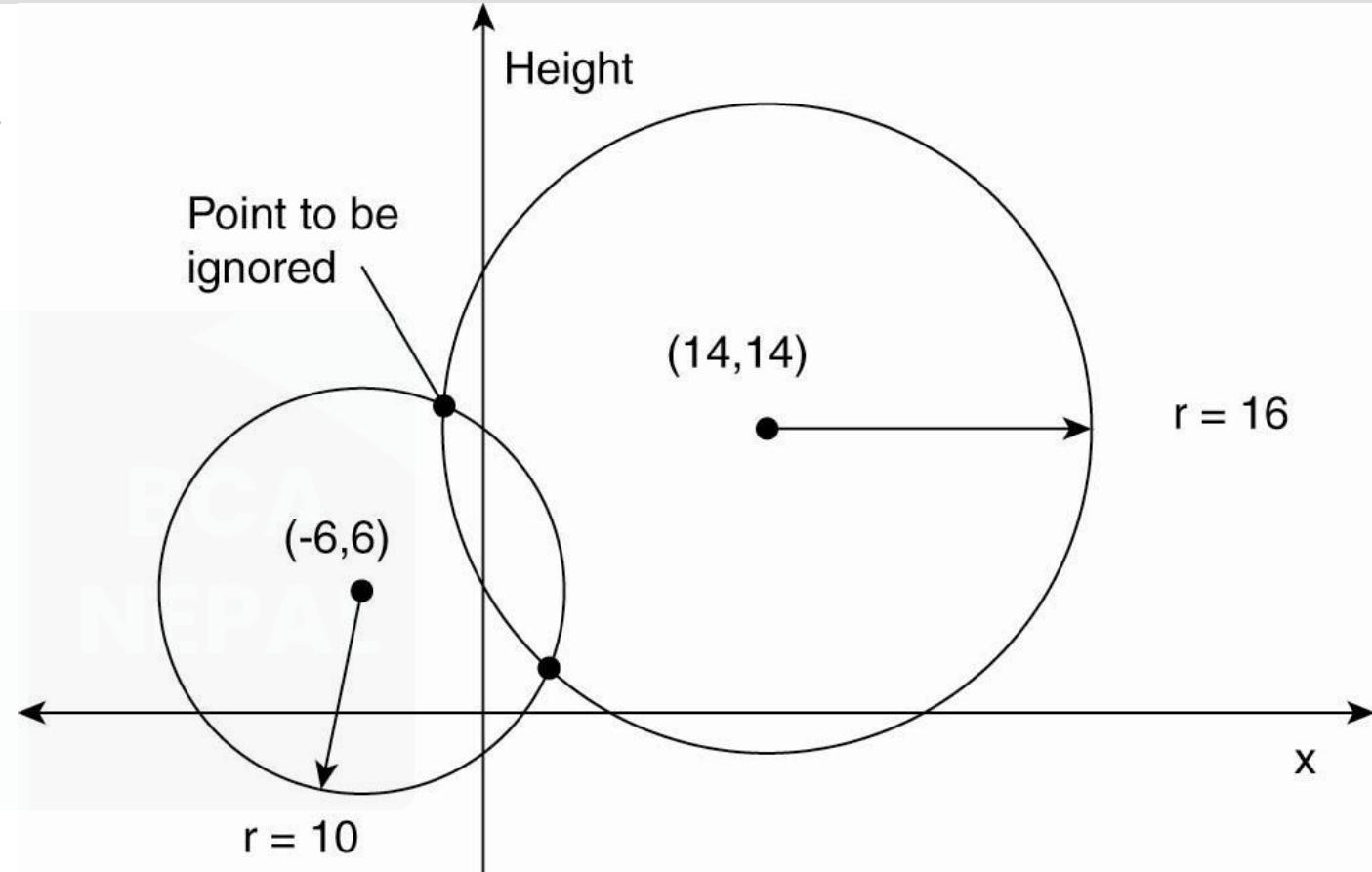
n = Number of votes gained

Decentralized Algorithm

- ▶ Now every replica has a coordinator that controls access
- ▶ Coordinators respond to requests at once: Yes or No
- ▶ For a process to use the resource it must receive permission from $m > n/2$ coordinators.
 - If the requester gets fewer than m votes it will wait for a random time and then ask again.
- ▶ If a request is denied, or when the CS is completed, notify the coordinators who have sent OK messages, so they can respond again to another request. (Why is this important?)
- ▶ More robust than the central coordinator approach. If one coordinator goes down others are available.
 - If a coordinator fails and resets then it will not remember having granted access to one requestor, and may then give access to another. According to the authors, it is highly unlikely that this will lead to a violation of mutual exclusion.

Global Positioning System

- ▶ Basic idea: You can get an accurate account of time as a side-effect of GPS.
- ▶ Real world facts that complicate GPS
 1. It takes a while before data on a satellite's position reaches the receiver.
 2. The receiver's clock is generally not in synch with that of a satellite.



Computing a position in a two-dimensional space

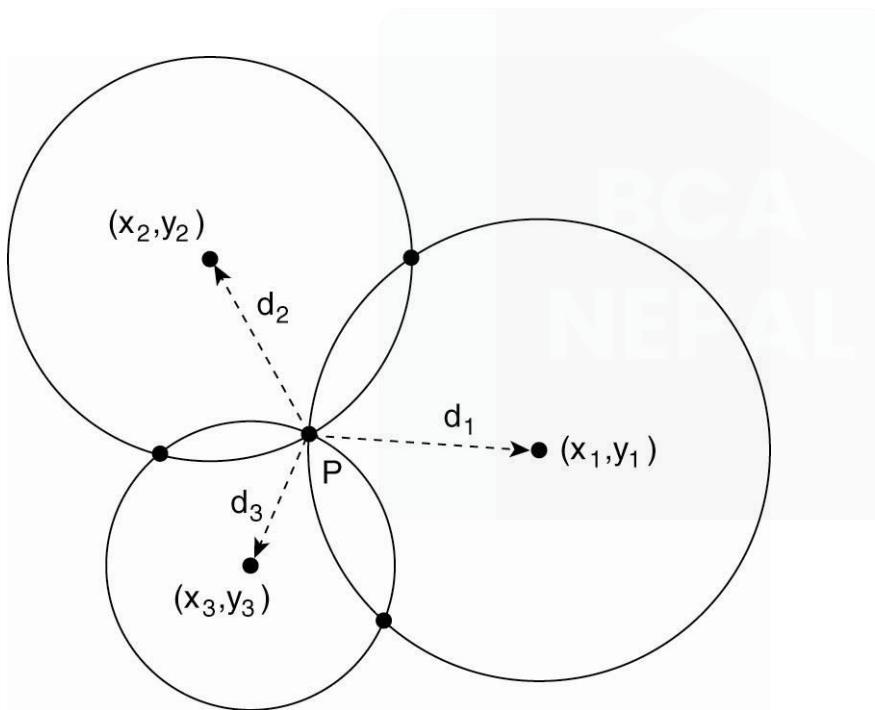
Global Positioning of Nodes

- ▶ When the number of nodes in a distributed system grows, it becomes increasingly difficult for any node to keep track of the others.
- ▶ Such knowledge may be important for executing distributed algorithms such as routing, multicasting, data placement, searching, and so on.
- ▶ There are many applications of geometric overlay networks.
- ▶ Consider the situation where a Web site at server O has been replicated to multiple servers on the Internet.
 - When a client C requests a page from O, the latter may decide to redirect that request to the server closest to C, give the best response time.
 - If the geometric location of C is known, as well as those of each replica server, O can then simply pick that server (i.e.) S, for which $d(C,S)$ is minimal.
 - There is no need to sample all the latencies between C and each of the replica servers

Global Positioning of Nodes

► Observation:

- A node P needs $k+1$ landmarks to compute its own position in a d -dimensional space.
Consider two-dimensional case.



Computing a node's position in a two-dimensional space.

Global Positioning of Nodes

Solution

- ▶ P needs to solve three equations in two unknowns

(x_p, y_p) :

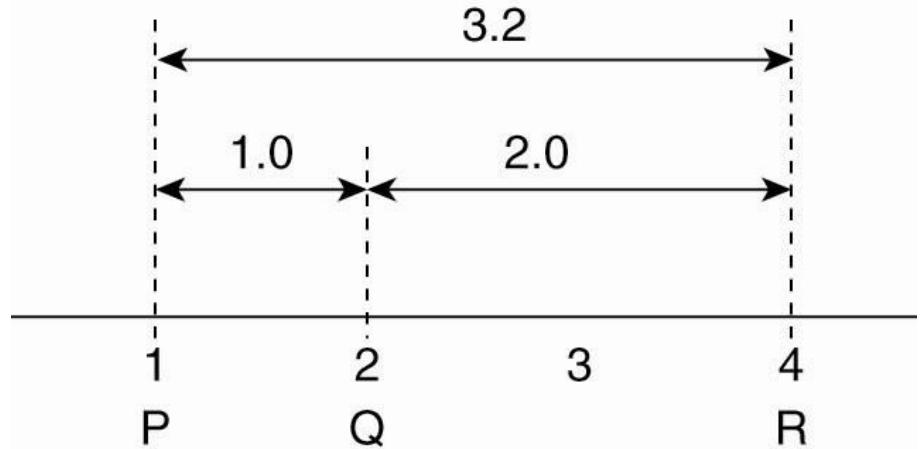
$$d = \sqrt{(x_i - xp)^2 + (y_i - yp)^2}$$

Computing position

- ▶ Problems :

- Measured latencies to landmarks fluctuate
- Computed distances will not even be consistent:
- Let the L landmarks measure their pairwise latencies \dots node P minimize

$$\sum_{i=1}^L \left[\frac{d(b_i, P) - \hat{d}(b_i, P)}{d(b_i, P)} \right]^2$$



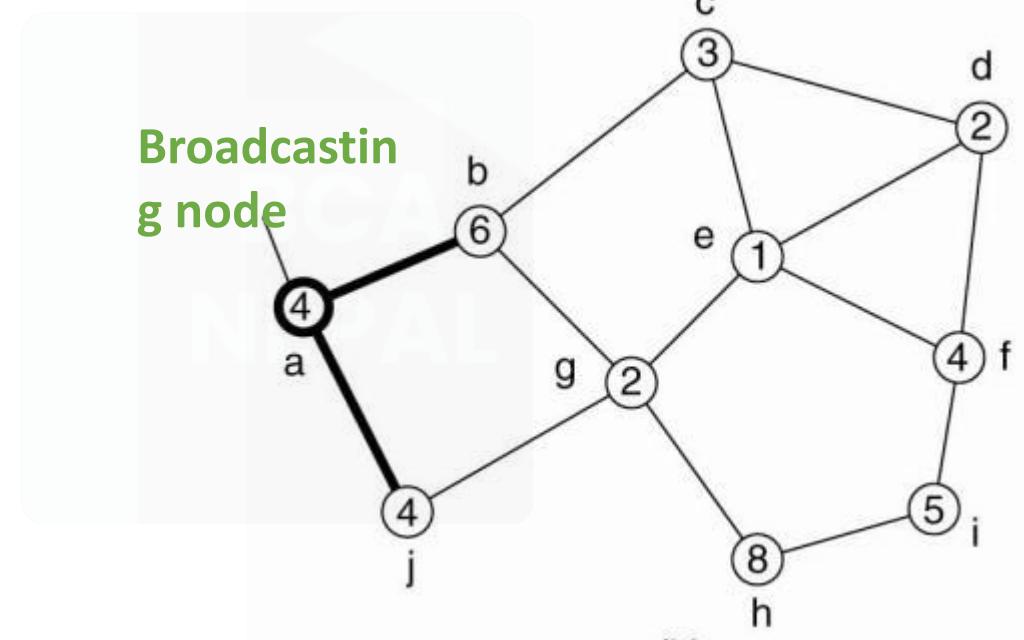
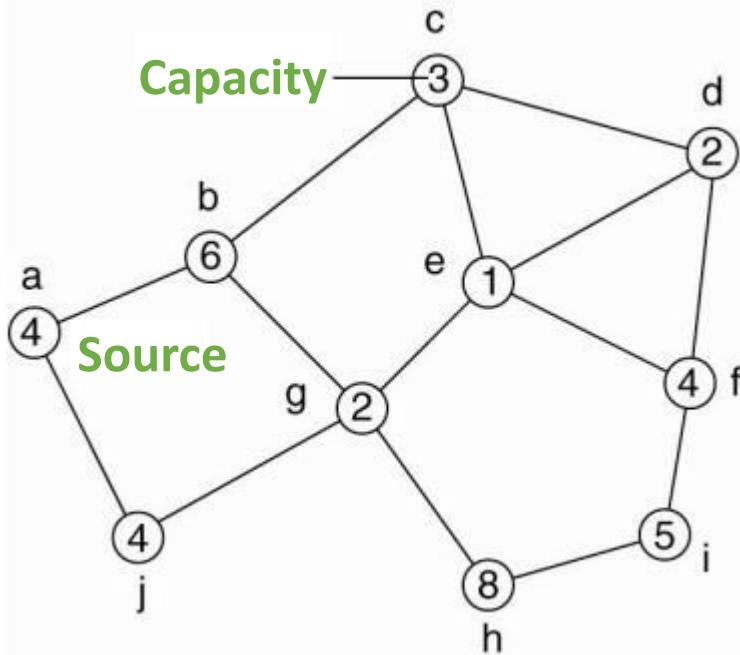
Inconsistent distance measurements in a one-dimensional space.

Elections in Wireless Environments

- ▶ Traditional algorithms aren't appropriate.
 - Can't assume reliable message passing or stable network configuration
- ▶ This discussion focuses on ad hoc wireless networks but ignores node mobility.
 - Nodes connect directly, no common access point, connection is short term
 - Often used in multiplayer gaming, on-the-fly file transfers, etc.
- ▶ Assumptions
 - Wireless algorithms try to find the **best node** to be coordinator; traditional algorithms are satisfied with any node.
 - Any node (the source) can initiate an election by sending an **ELECTION** message to its neighbors – nodes within range.
 - When a node receives its first ELECTION message the sender becomes its **parent node**.

Elections in Wireless Environments

- ▶ Node a initiates an ELECTION message by broadcasting to nodes b and j

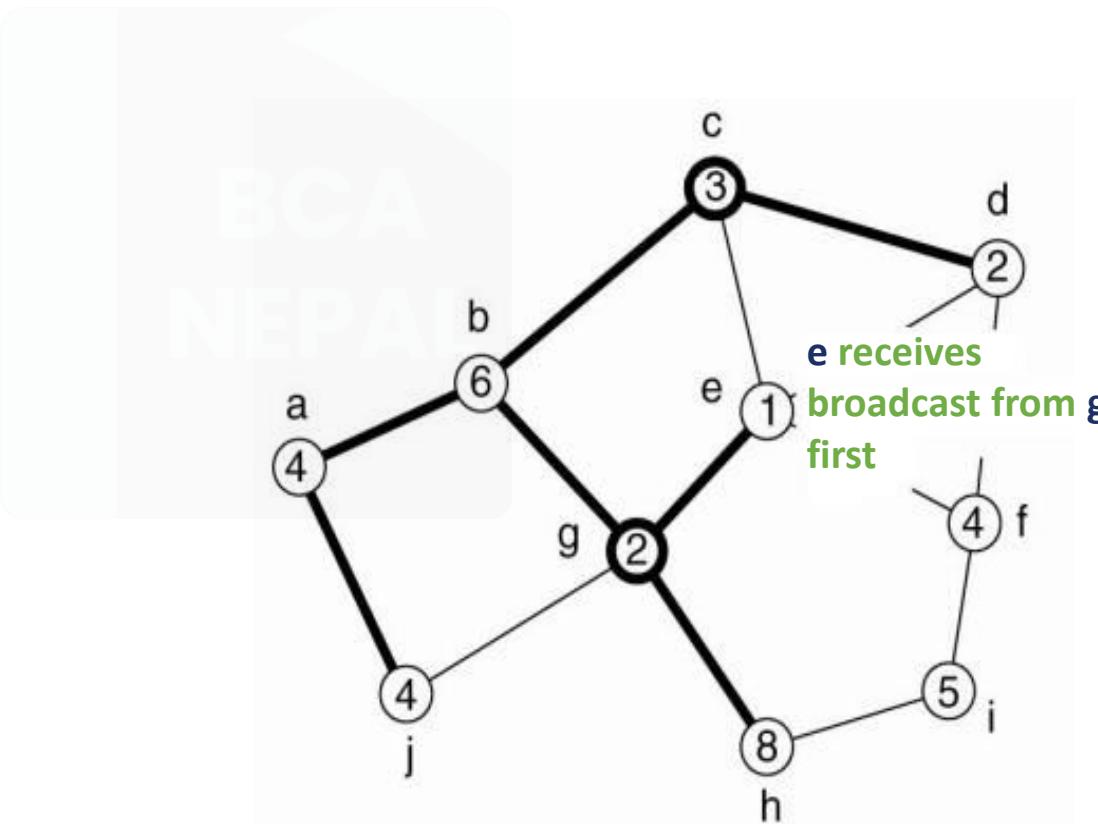
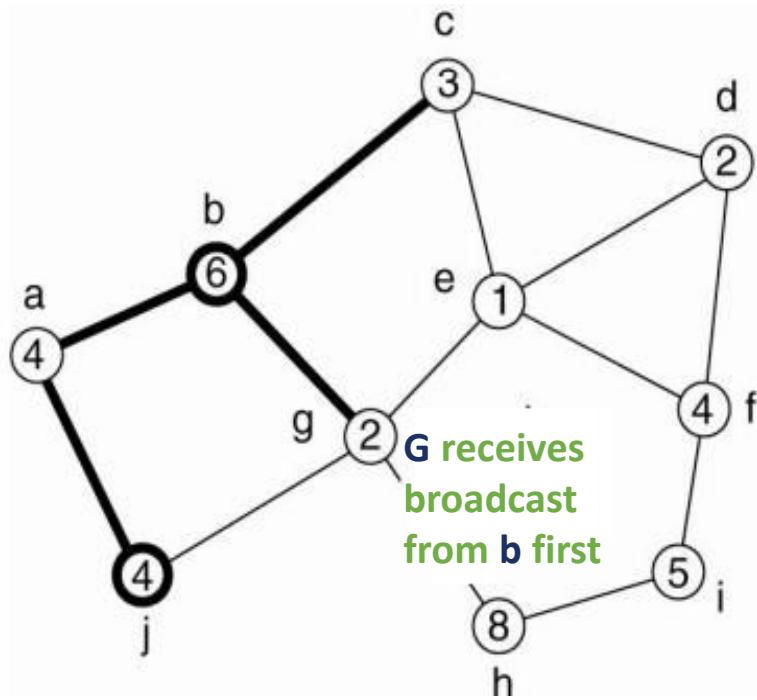


Election algorithm in a wireless network,
with node a as the source: Initial
network

Election algorithm in a wireless network,
with node a as the source: The build-tree
phase

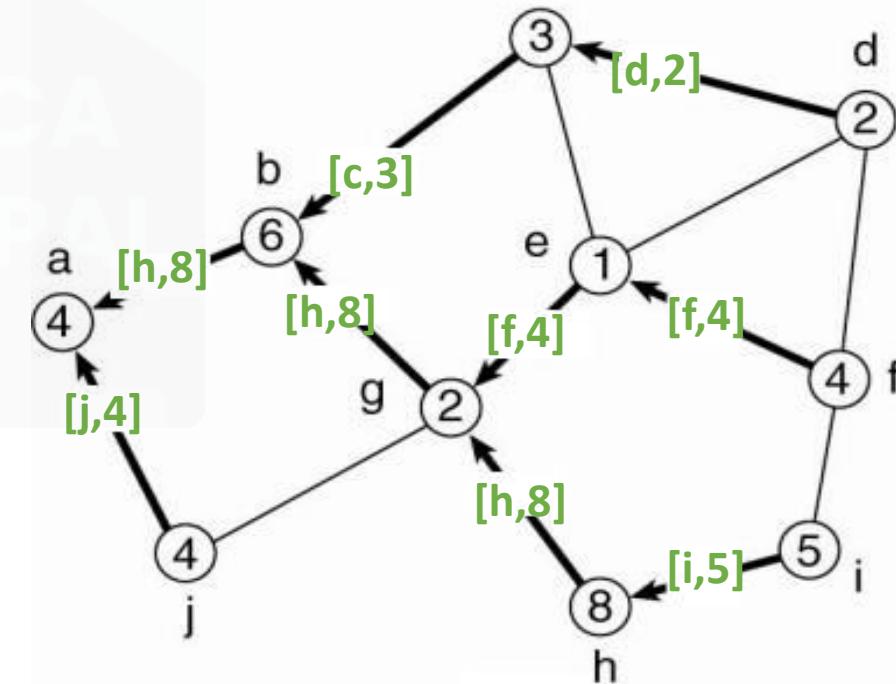
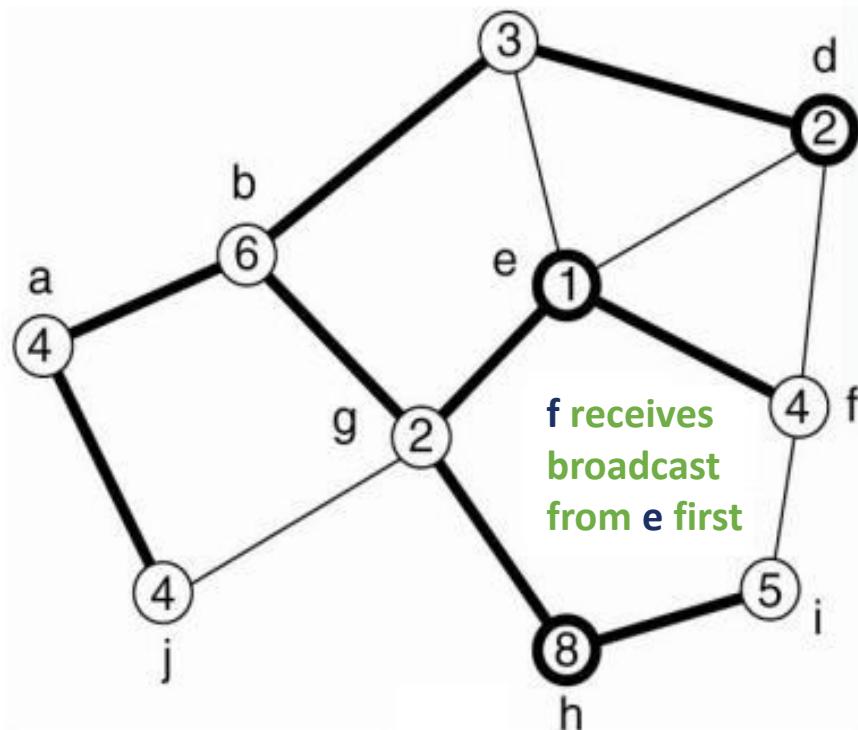
Elections in Wireless Environments

- ▶ The Node b marks a as parent and passes the ELECTION message to node g
- ▶ Election message it transmitted till the leaf nodes
- ▶ Each nodes then transmit back the best available option to its parent node



Elections in Wireless Environments

- ▶ Node g has node e and h as child nodes but it will propagate only the best node i.e [h,8] back to its own parent.
- ▶ When multiple elections are initiated, nodes tend to join the ones having highest identifier in the source tag



Elections in Large-Scale Systems

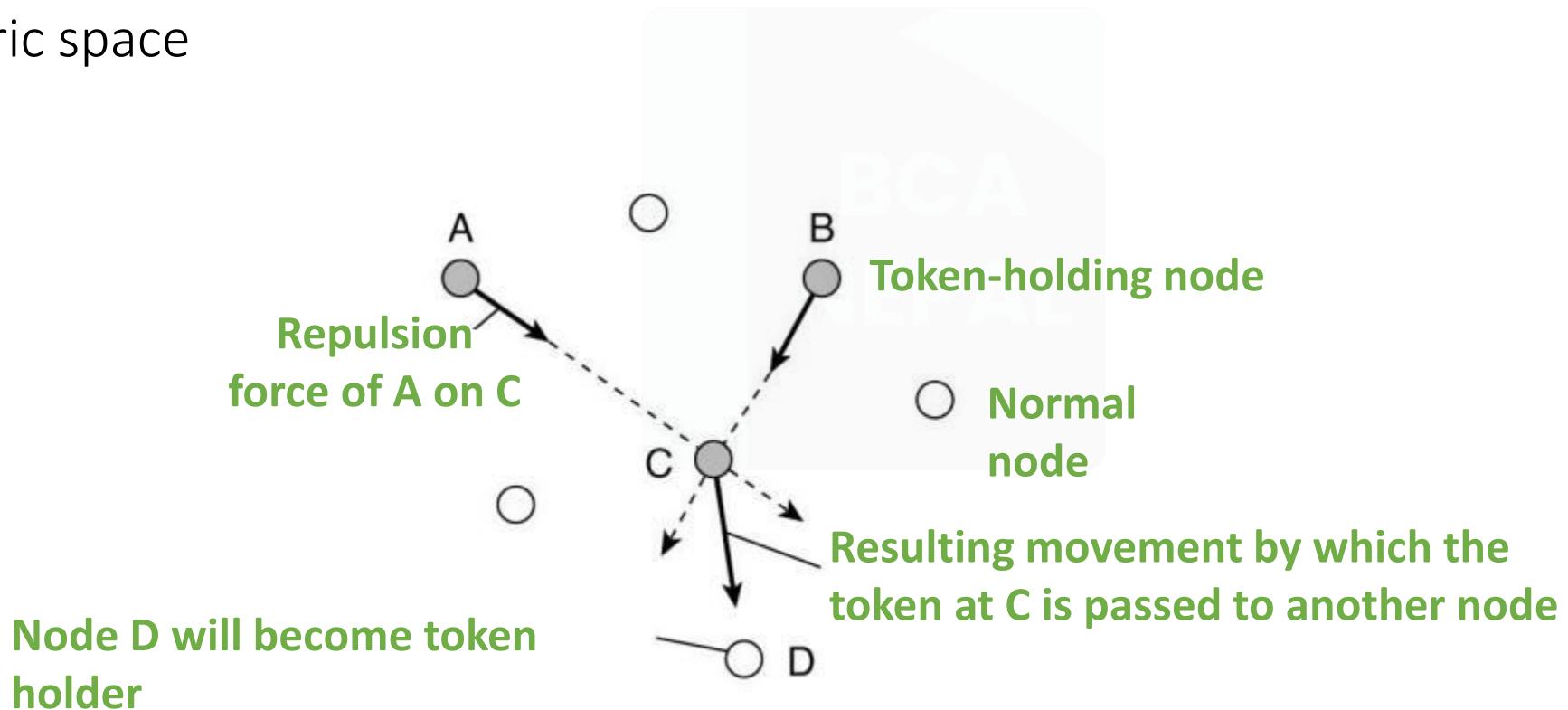
- ▶ Above discussed situations deal with selecting one coordinator as they are small distributed systems
- ▶ For a larger network, a situation arises where we need to select super peers in a peer- to-peer network.
- ▶ Requirements for superpeer selection:
 - Normal nodes should have low-latency access to superpeers.
 - Superpeers should be evenly distributed across the overlay network.
 - There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
 - Each superpeer should not need to serve more than a fixed number of normal nodes.

Elections in Large-Scale Systems

- ▶ Requirements are relatively easy to meet in most peer-to-peer systems as the overlay network is either DHT based or randomly unstructured.
- ▶ In DHT networks,
 - Idea is to reserve a fraction of the identifier space for Superpeers
 - Each node receives a m -bit identifier and first k bits are reserved to identify the Superpeers
 - For N superpeers, first $\log_2(N)$ bits of any key can be used to identify the Superpeers

Elections in Large-Scale Systems

- ▶ Total of N tokens are spread across N randomly-chosen nodes
- ▶ Each token represents a repelling force by which another token is inclined to move away If all tokens exert the same repulsion force, they will move away and spread evenly in the geometric space



Distributed System

Course Code: CACS352
Year/Sem: III/VI

Unit 7. Consistency and Replication

5 Hrs.

7.1 Introduction

7.2 Data-centric consistency models

7.3 Client-centric consistency models

7.4 Replica management

7.5 Consistency protocols

7.6 Caching and Replication in Web

...

Why Replication?

- ▶ *Replication is the process of maintaining the data at multiple computers.*
- ▶ Replication is necessary for:
 - ▶ **Improving performance** : A client can access nearby replicated copies and save latency
 - ▶ **Increasing the availability of services** : Replication can mask failures such as server crashes and network disconnection
 - ▶ **Enhancing the scalability of systems** : Requests to data can be distributed across many servers, which contain replicated copies of the data
 - ▶ **Securing against malicious attacks** : Even if some replicas are malicious, security of data can be guaranteed by relying on replicated copies at non-compromised servers

Why Replication?

- ▶ **Data replication:** common technique in distributed systems
- ▶ **Reliability :** If one replica is unavailable or crashes, use another
 - Protect against corrupted data
- ▶ **Performance :** Scale with size of the distributed system (replicated web servers)
 - Scale in geographically distributed systems (web proxies)

Key issue:

need to maintain consistency of replicated data

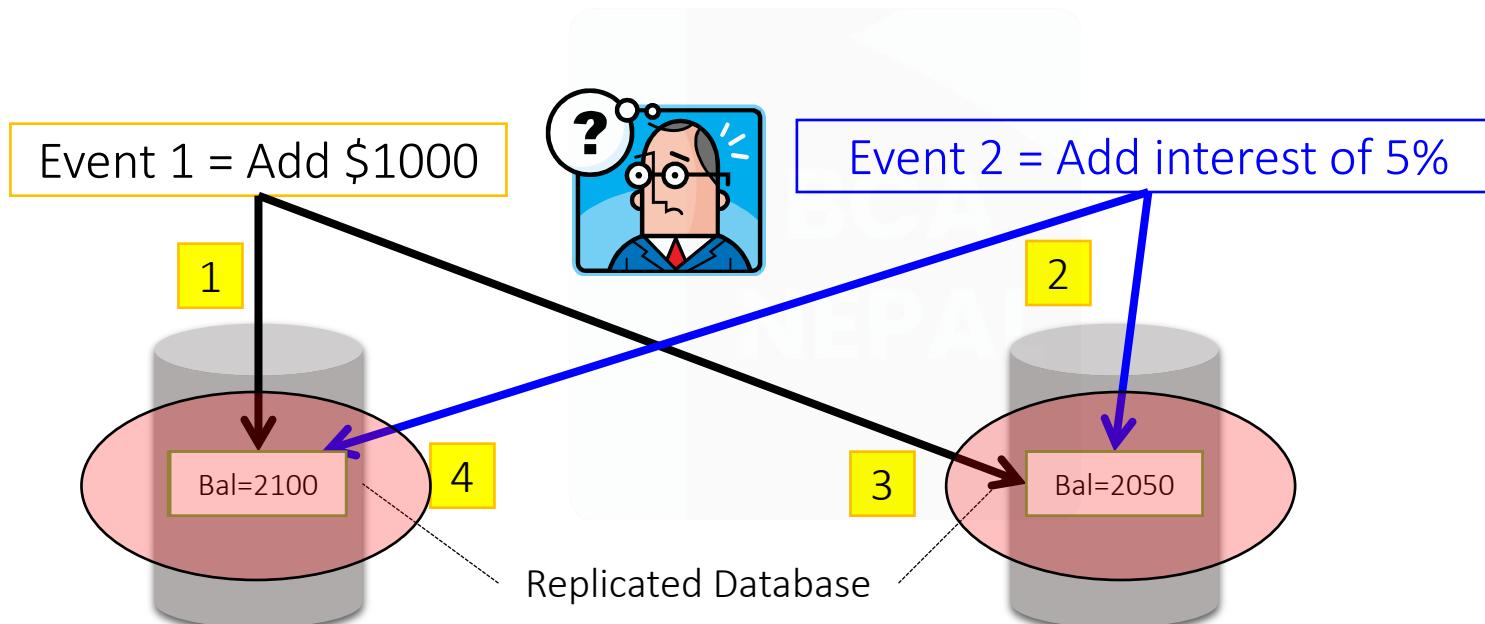
- ▶ If one copy is modified, others become inconsistent
- ▶ When and how modifications need to be carried out, determines the price of replication??

Performance and scalability

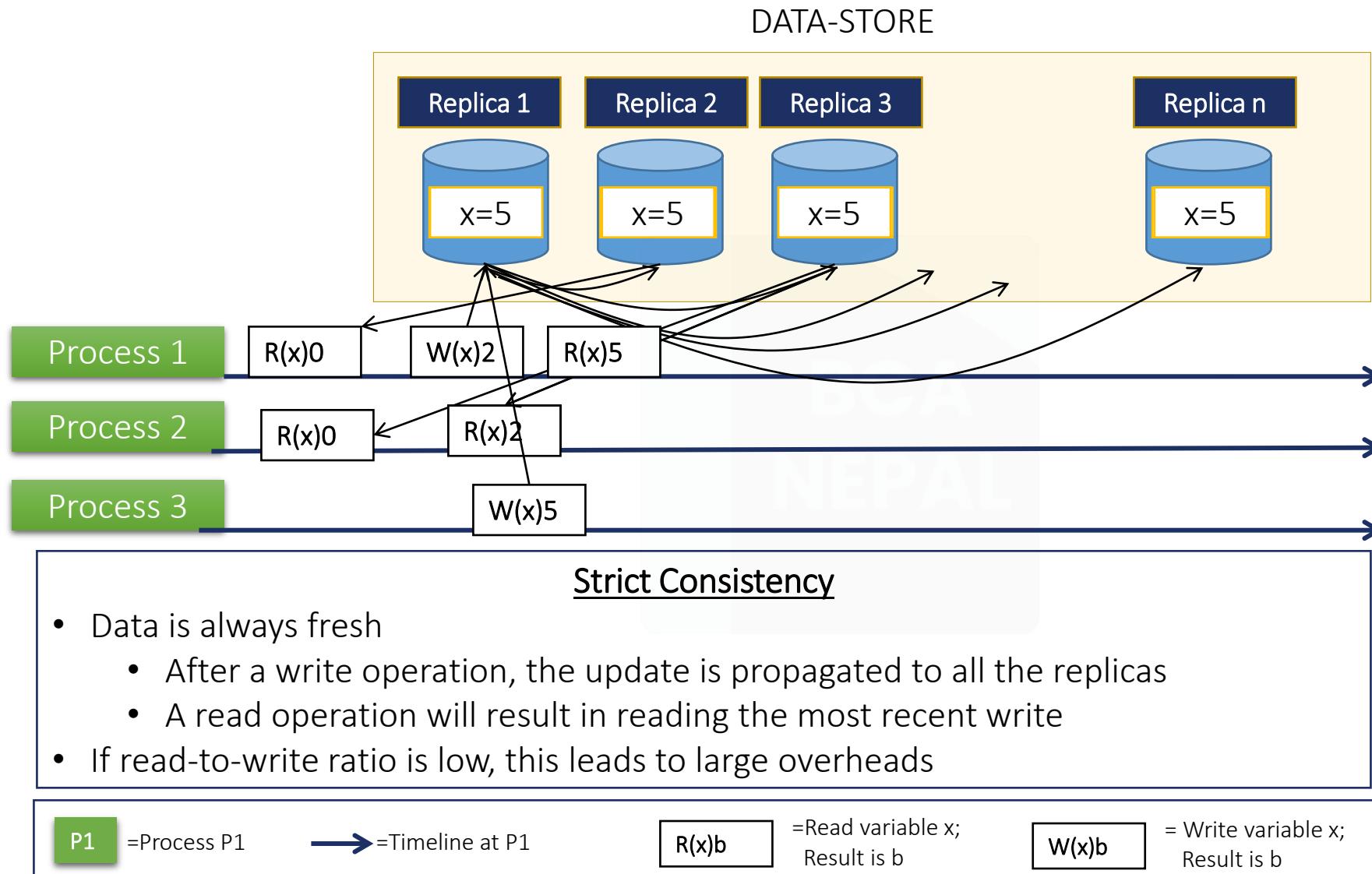
- ▶ **Main issue:** To keep replicas consistent, we generally need to ensure that all **conflicting** operations are done in the same order everywhere.
- ▶ **Conflicting operations:** From the world of transactions:
 - Read–write conflict: a read operation and a write operation act concurrently
 - Write–write conflict: two concurrent write operations
- ▶ Issue: Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability
 - **Solution:** weaken consistency requirements so that hopefully global synchronization can be avoided

Why Consistency?

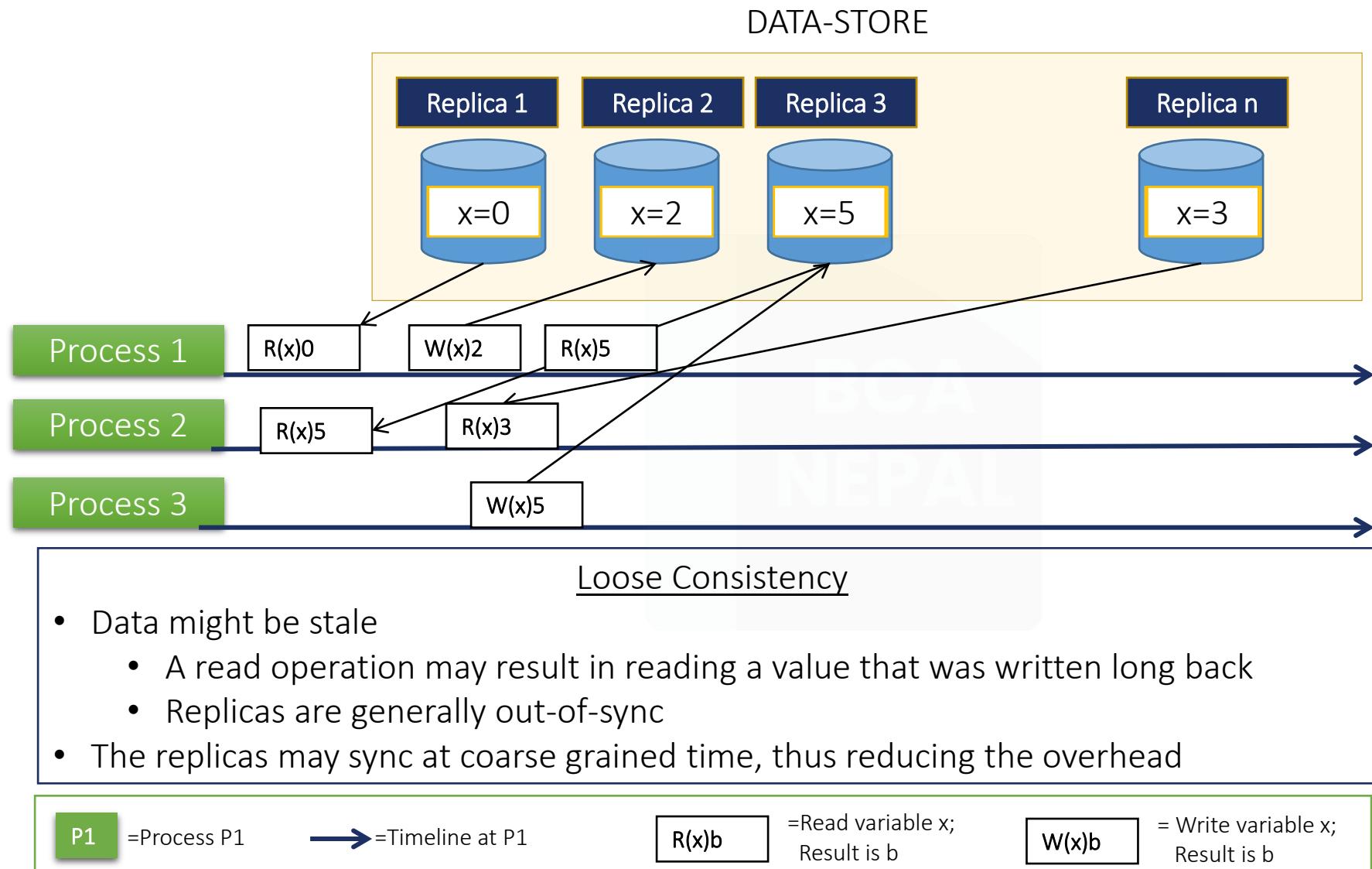
- ▶ But (server-side) replication comes with a cost, which is the necessity for maintaining consistency (or more precisely consistent ordering of updates)
- ▶ Example: A Bank Database



Maintaining Consistency of Replicated Data

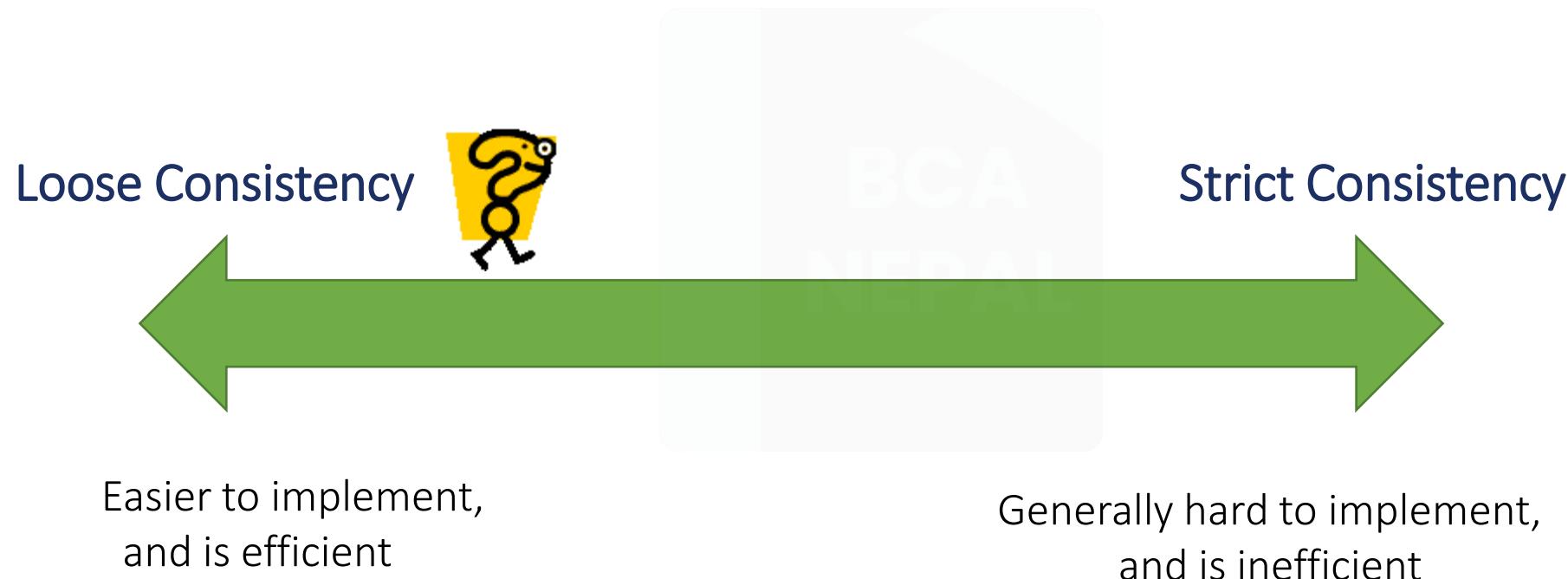


Maintaining Consistency of Replicated Data



Trade-offs in Maintaining Consistency

- ▶ Maintaining consistency should balance between the strictness of consistency versus efficiency (or performance)
 - Good-enough consistency depends on your application



Consistency Model

- ▶ A consistency model is a contract between:
 - The process that wants to use the data
 - and the data-store
- ▶ A consistency model states the level (or degree) of consistency provided by the data-store to the processes while reading and writing data

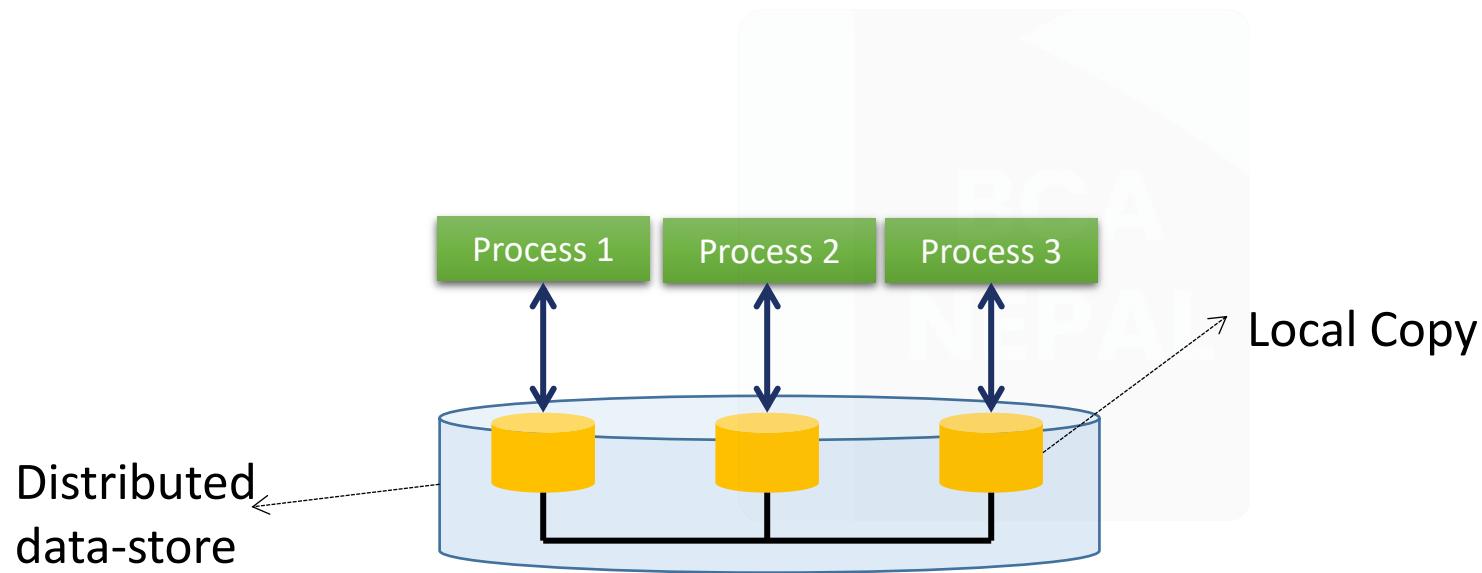
Types of Consistency Models

Consistency models can be divided into two types:

1. **Data-Centric Consistency Models** : These models define how updates are propagated across the replicas to keep them consistent
2. **Client-Centric Consistency Models** : These models assume that clients connect to different replicas at different times.
 - They ensure that whenever a client connects to a replica, the replica is brought up to date with the replica that the client accessed previously

Data-centric consistency models

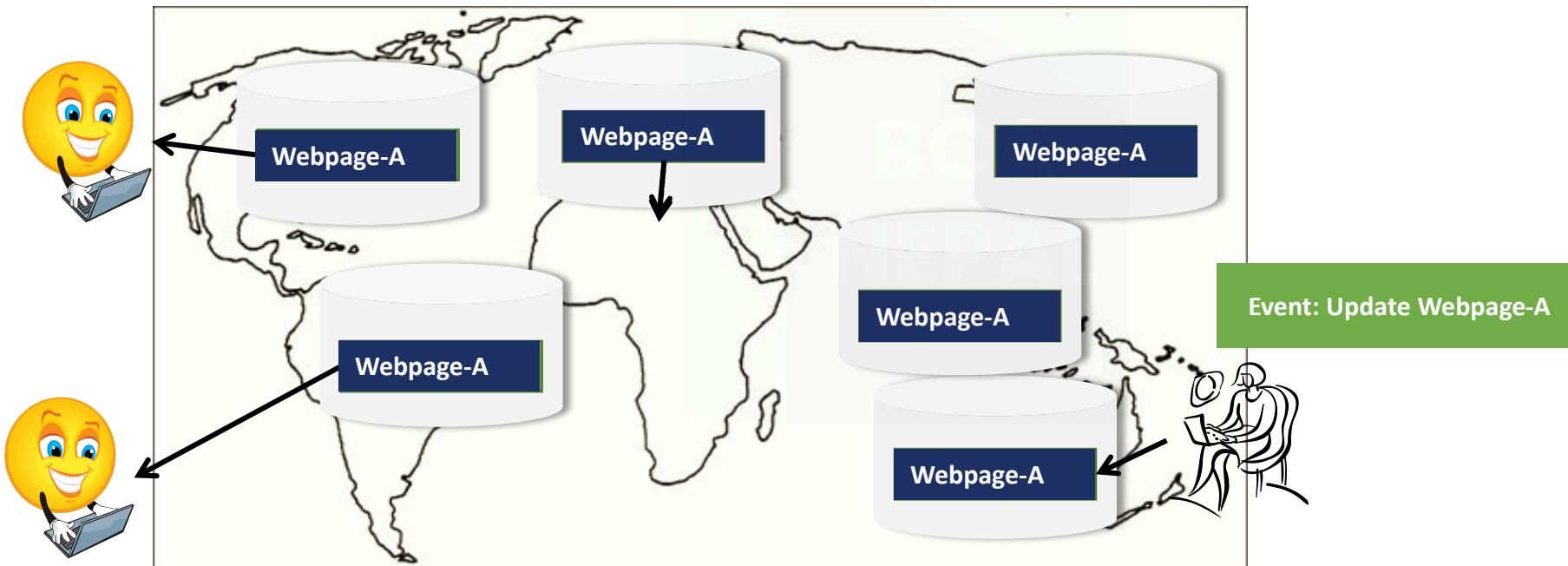
- ▶ A data-store can be read from or written to by any process in a distributed system.
- ▶ A local copy of the data-store (replica) can support “fast reads”.
- ▶ However, a **write** to a local replica needs to be propagated to all remote replicas.



The general organization of a logical data store, physically distributed and replicated across multiple processes.

Applications that can use Data-centric Models

- ▶ Data-centric models are applicable when many processes are concurrently updating the data-store
- ▶ But, do all applications need all replicas to be consistent?



Data-Centric Consistency Model is too strict when

- One client process updates the data
- Other processes read the data, and are OK with reasonably stale data

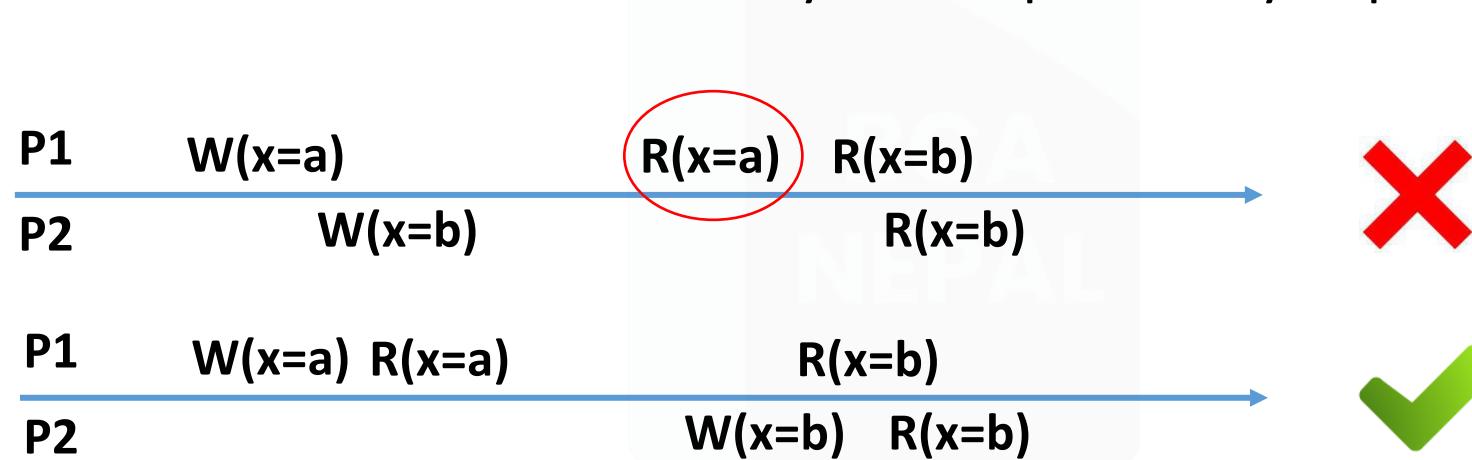
Data-Centric Consistency Models

- ▶ **Strong consistency models:** Operations on shared data are synchronized
 - ↳ Strict consistency (related to time)
 - ↳ Sequential consistency (what we are used to)
 - ↳ Causal consistency (maintains only causal relations)
 - ↳ FIFO consistency (maintains only individual ordering)

- ▶ **Weak consistency models:** Synchronization occurs only when shared data is locked and unlocked
 - ↳ General weak consistency,
 - ↳ Release consistency,
 - ↳ Entry consistency.

Strict Consistency (related to time)

- Value returned by a read operation on a memory address is always same as the most recent write operation to that address.
- All writes instantaneously become visible to all processes.
- Implementation of this model for a DSM system is practically impossible.



- Practically impossible because absolute synchronization of clock of all the nodes of a distributed system is not possible.

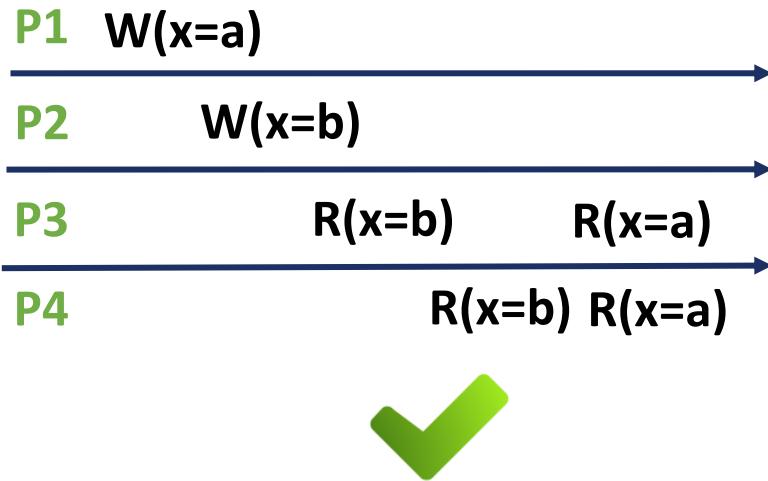
Strict Consistency

- ▶ Practically impossible because absolute synchronization of clock of all the nodes of a distributed system is not possible.
- ▶ In a **single processor** system strict consistency is for free, it's the behavior of main memory with atomic reads and writes
- ▶ However, in a **DSM** without the notion of a global time it is hard to determine what is the most recent write

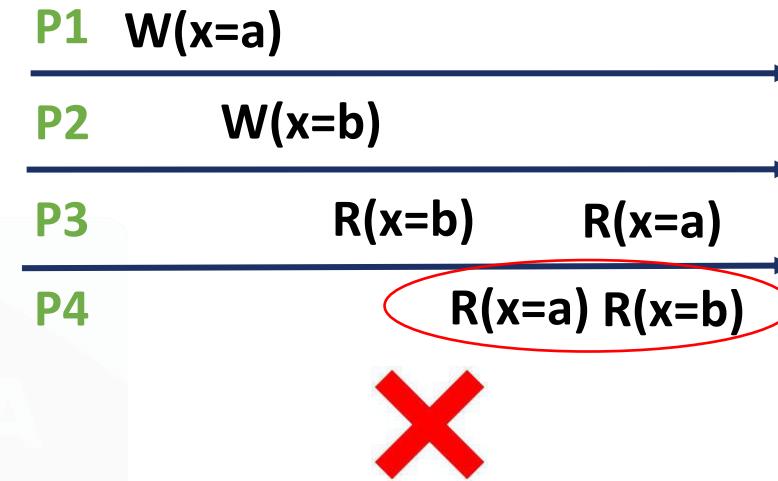
Sequential Consistency (what we are used to)

- ▶ A **shared memory system** is said to support the **sequential consistency** model if all processes see the same order.
- ▶ Exact order of access operations are interleaved does not matter.
- ▶ The consistency requirement of the sequential consistency model is **weaker** than that of the strict consistency model.
- ▶ It is **Time independent process**.
- ▶ reads and writes of an individual process occur in their **program order**
- ▶ **reads and writes** of different processes occur in some **sequential order** as long as interleaving of concurrent accesses is valid.
- ▶ Sequential consistency is weaker than strict consistency

Sequential Consistency



1. P1 performs $W(x=a)$.
2. Later (in absolute time), P2 performs $W(x=b)$.
3. Both P3 and P4 first read value b and later value a.
4. Write operation of process P2 appears to have taken place before that of P1.

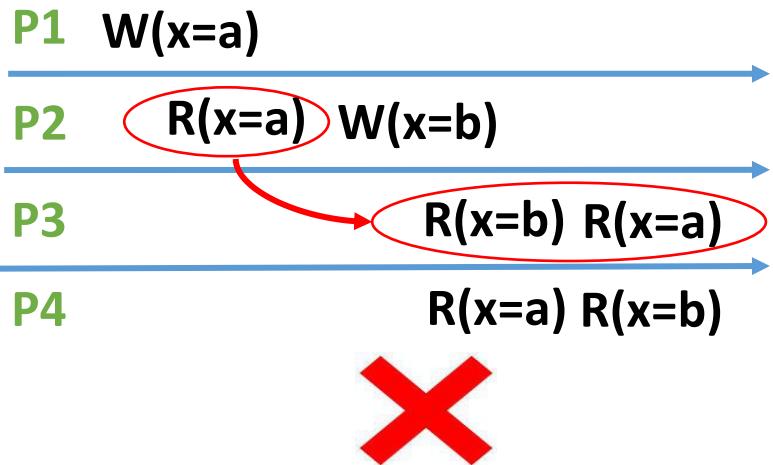


1. Violates sequential consistency - not all processes see the same interleaving of write operations.
2. To process P3, it appears as if the data item has first been changed to b and later to a.
3. BUT, P4 will conclude that the final value is b.

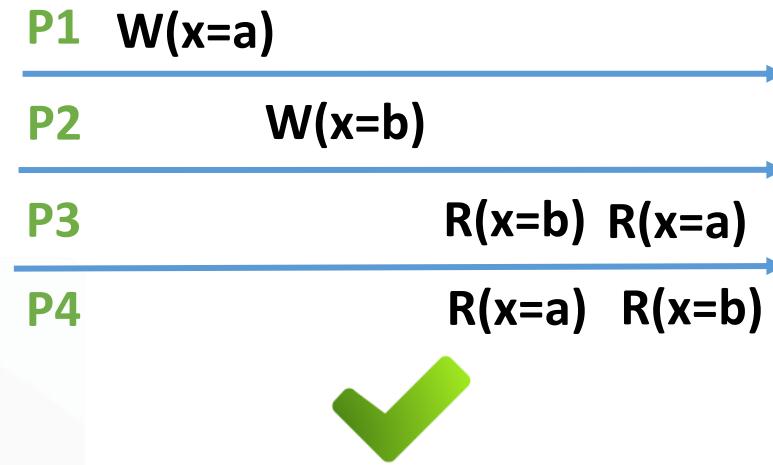
Causal Consistency (maintains only causal relations)

- ▶ All **write operations** that are potentially causally related are seen by all **processes** in the **same(correct) order**.
- ▶ Write operations that are **not potentially causally** related may be seen by **different processes** in **different orders**.
- ▶ If a **write operation (w2)** is **causally** related to another **write operation (w1)**, the acceptable **order** is **(w1, w2)** because the value written by w2 might have been influenced in some way by the value written by w1.
- ▶ Therefore, **(w2, w1)** is **not an acceptable order**.
- ▶ Conversely, if two processes **spontaneously** and **simultaneously** write two different data items, these **are not causally related**.
- ▶ Operations that are **not causally related** are said to be **concurrent**.

Causal Consistency (W – W order)



- $W(x=b)$ potentially depending on $W(x=a)$ because b may result from a computation involving the value read by $R(x=a)$.
- The two writes are causally related, so all processes must see them in the same order.
- It is incorrect.



- ▶ Read has been removed, so $W(x=a)$ and $W(x=b)$ are now concurrent writes.
- ▶ A causally consistent store does not require concurrent writes to be globally ordered.
- ▶ It is correct.

FIFO Consistency (maintains only individual ordering)

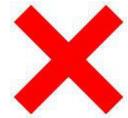
- ▶ This is also called “PRAM Consistency” – Pipelined RAM.
- ▶ Program order must be respected
- ▶ In FIFO consistency, writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.
- ▶ It is weaker than causal consistency.
- ▶ This model is simple and easy to implement having good performance because processes are ready in the pipeline.
- ▶ Implementation is done by sequencing write operations performed at each node independently of the operations performed on other nodes.
- ▶ Example: If (w11) and (w12) are write operations performed by p1 in that order and (w21),(w22) by p2. A process p3 can see them as [(w11,w12),(w21,w22)] while p4 can view them as [(w21,w22),(w11,w12)].

FIFO Consistency

P1	W(x=a)	W(x=c)
P2	R(x=a)	W(x=b)
P3		R(x=a) R(x=c) R(x=b)
P4		R(x=b) R(x=a) R(x=c)



P1	W(x=a)	W(x=c)
P2	R(x=a)	W(x=b)
P3		R(x=a) R(x=c) R(x=b)
P4		R(x=b) R(x=c) R(x=a)



Data centric model- Summary

Consistency models not using synchronization operations.

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters .
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used . Writes from different processes may not always be seen in that order

Data centric model- Summary

Consistency models with synchronization operations

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered

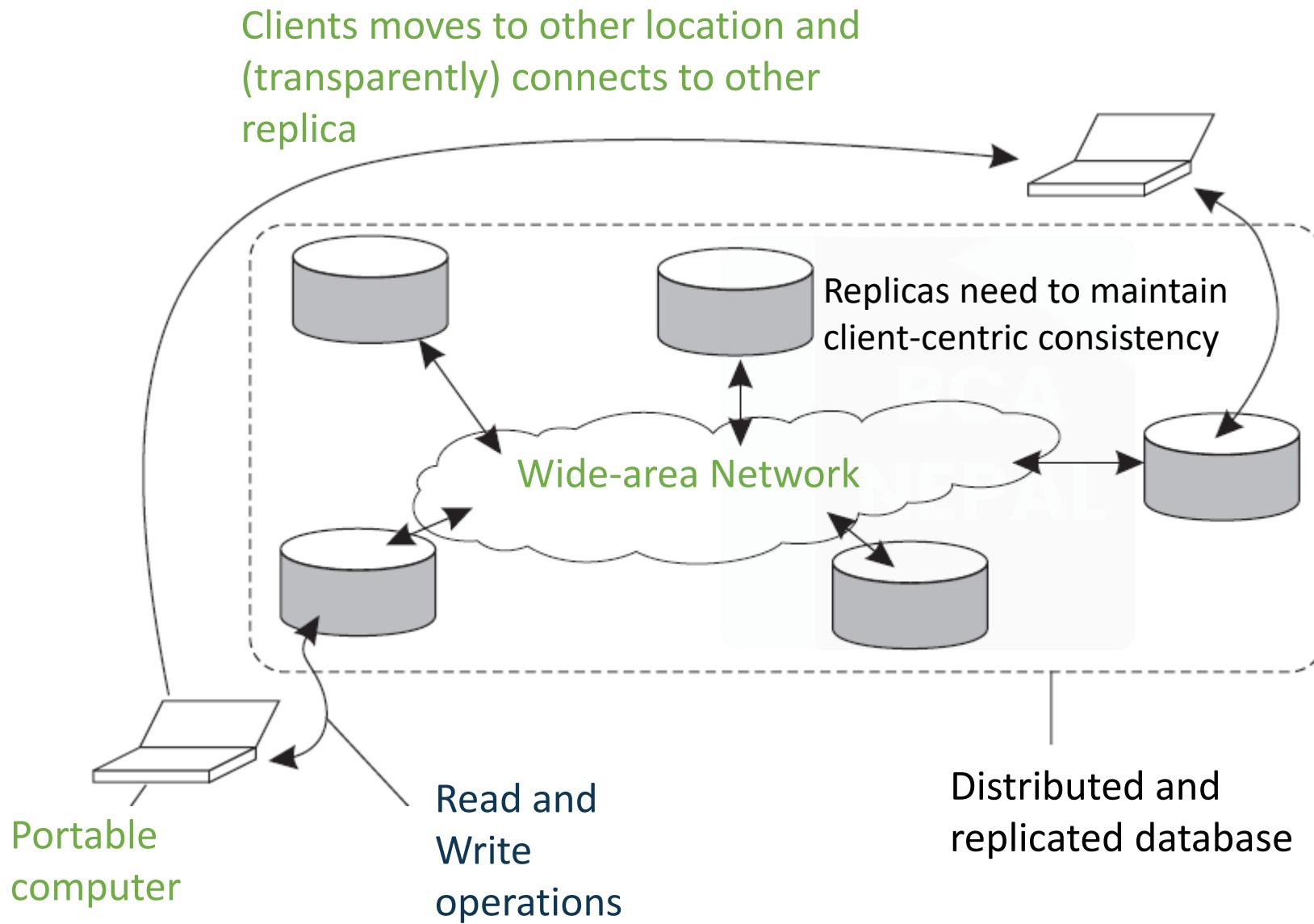
Client-Centric Consistency Models

- ▶ The previously studied consistency models concern themselves with maintaining a consistent (**globally accessible**) data-store in the presence of **concurrent read/write operations**
- ▶ Another class of distributed data-store is that which is characterized by the **lack of simultaneous updates**. Here, the emphasis is more on **maintaining a consistent view of things** for the individual client process that is **currently operating on the data-store**.
- ▶ How fast should updates (writes) be made available to read-only processes?
 - Think of most **database systems**: **mainly read**.
 - **DNS**: **write-write conflicts do no occur, only read-write conflicts**.
 - **WWW**: as with DNS, except that heavy use of **client-side caching is present**: even the return of stale pages is acceptable to most users.
- ▶ These systems all exhibit a high degree of acceptable inconsistency ... with the replicas gradually becoming consistent over time.

Eventual Consistency (at the end)

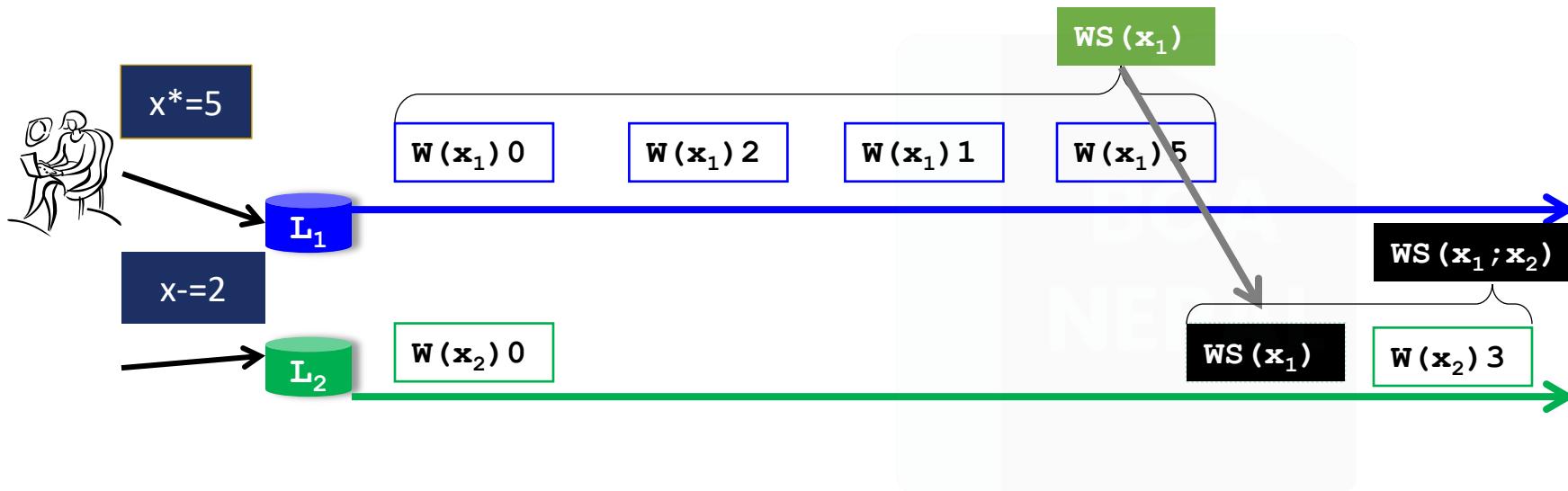
- ▶ In Systems that tolerate high degree of inconsistency, if no updates take place for a long time all replicas will gradually and eventually become consistent. This form of consistency is called **eventual consistency**.
- ▶ Eventual consistency only requires those updates that **guarantee propagation** to all replicas.
- ▶ Eventual consistent **data stores work fine** as long as clients always access the **same replica**.
- ▶ **Write conflicts** are often relatively easy to solve when assuming that **only a small group of processes can perform updates**. Eventual consistency is therefore often **cheap to implement**.
- ▶ Eventual consistency for replicated data is fine if clients always access the same replica
Client centric consistency provides consistency guarantees for a single client with respect to the data stored by that client

Eventual Consistency



Client Consistency Guarantees

- ▶ Client-centric consistency provides guarantees for a single client for its accesses to a data-store
- ▶ Example: Providing consistency guarantee to a client process for data \mathbf{x} replicated on two replicas. Let \mathbf{x}_i be the local copy of a data \mathbf{x} at replica L_i .



$WS(x_1)$ = Write Set for x_1 = Series of ops being done at some replica that reflects how L_1 updated x_1 till this time

$WS(x_1; x_2)$ = Write Set for x_1 and x_2 = Series of ops being done at some replica that reflects how L_1 updated x_1 and, later on, how x_2 is updated on L_2

L_i = Replica i

$R(x_i)b$ = Read variable x at replica i ; Result is b

$W(x)b$ = Write variable x at replica i ; Result is b

$WS(x_i)$ = Write Set

Client Consistency models

Four types of client-centric consistency models

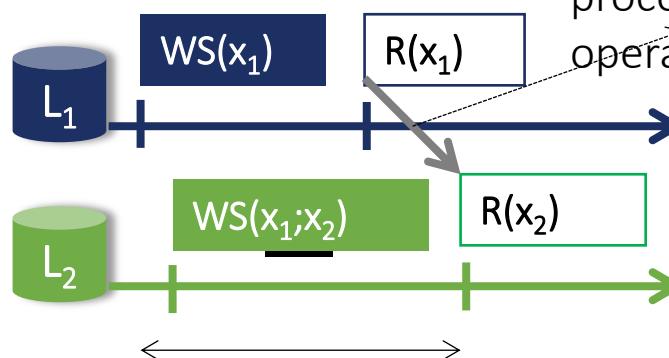
1. Monotonic Reads
2. Monotonic Writes
3. Read Your Writes
4. Write Follow Reads



Monotonic Reads Consistency

- ▶ A data store is said to provide monotonic-read consistency if a process reads the value of a data item x , any successive read operation on x by that process will always **return that same value or a more recent value.**
- ▶ A process has seen a **value of x at time t** , it will **never see an older version of x at a later time.**
- ▶ Example:
 - Automatically reading your **personal calendar updates** from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.
 - Reading (not modifying) **incoming mail** while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

Monotonic Reads Consistency

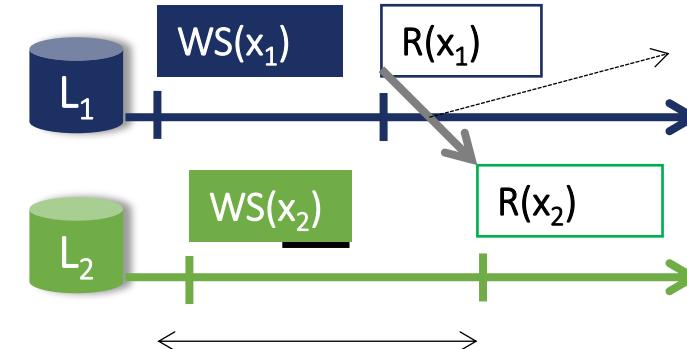


Order in which client process carries out the operations

A monotonic-read consistent data store.

Return of R(x₂) should at least as recent as R(x₁)

1. Process P first performs a read operation on x at L₁, returning the value of x₁ (at that time). This value results from the write operations in WS (x₁) performed at L₁.
2. Later, P performs a read operation on x at L₂, shown as R (x₂).
3. To guarantee monotonic-read consistency, all operations in WS (x₁) should have been propagated to L₂ before the second read operation takes place



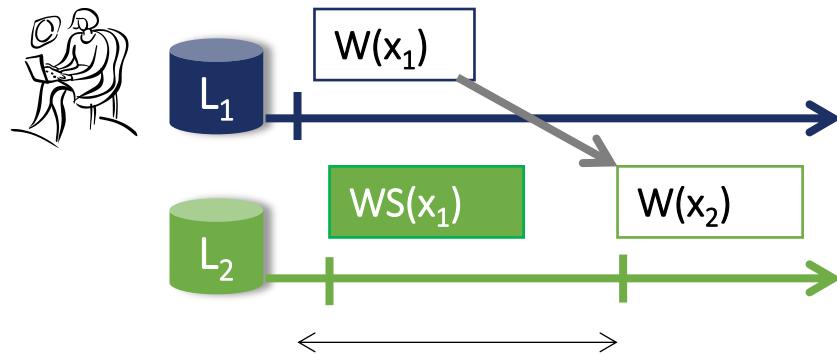
A data store that does not provide monotonic reads.

1. Situation in which monotonic-read consistency is not guaranteed.
2. After process P has read x₁ at L₁, it later performs the operation R (x₂) at L₂ .
3. But, only the write operations in WS (x₂) have been performed at L₂ .
4. No guarantees are given that this set also contains all operations contained in WS (x₁).

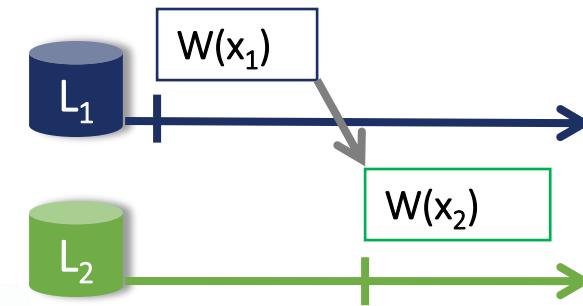
Monotonic Writes

- ▶ A data store is said to be monotonic-write consistent if a write operation by a process on a data item x is completed before any successive write operation on X by the same process.
- ▶ A write operation on a copy of data item x is performed only if that copy has been brought up to date by means of any preceding write operations, which may have taken place on other copies of x .
- ▶ Example: Monotonic-write consistency guarantees that if an update is performed on a copy of Server S , all preceding updates will be performed first. The resulting server will then indeed become the most recent version and will include all updates that have led to previous versions of the server.

Monotonic Writes



W(x₂) operation should be performed only after the result of W(x₁) has been updated at L₂



The data-store does not provide monotonic write consistency

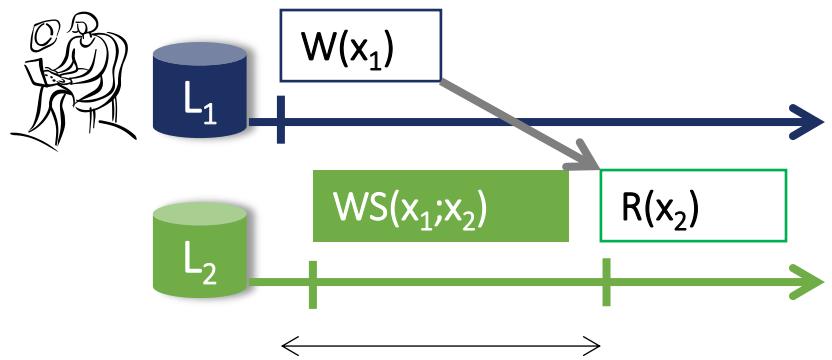
1. Process P performs a write operation on x at local copy L₁, presented as the operation W(x₁).
2. Later, P performs another write operation on x, but this time at L₂, shown as W(x₂).
3. To ensure monotonic-write consistency, the previous write operation at L₁ must have been propagated to L₂.
4. This explains operation W(x₁) at L₂, and why it takes place before W(x₂).

1. Situation in which monotonic-write consistency is not guaranteed.
2. Missing is the propagation of W(x₁) to copy L₂.
3. No guarantees can be given that the copy of x on which the second write is being performed has the same or more recent value at the time W(x₁) completed at L₁.

Read Your Writes

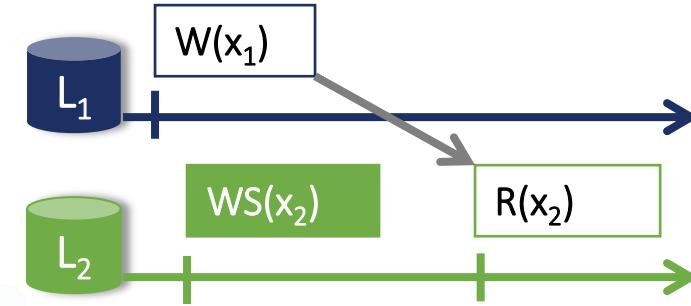
- ▶ A data store is said to provide read-your-writes consistency if the effect of a write operation by a process on data item x will always be a successive read operation on x by the same process.
- ▶ A write operation is always completed before a successive read operation by the same process no matter where that read operation takes place.
- ▶ Example: Updating a Web page and guaranteeing that the Web browser shows the newest version instead of its cached copy.

Read Your Writes



R(x₂) operation should be performed only after the updating the Write Set WS(x₁) at L₂

1. Process P performed a write operation W(x₁) and later a read operation at a different local copy.
2. Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation.
3. This is expressed by WS (x₁; x₂), which states that W (x₁) is part of WS (x₂).



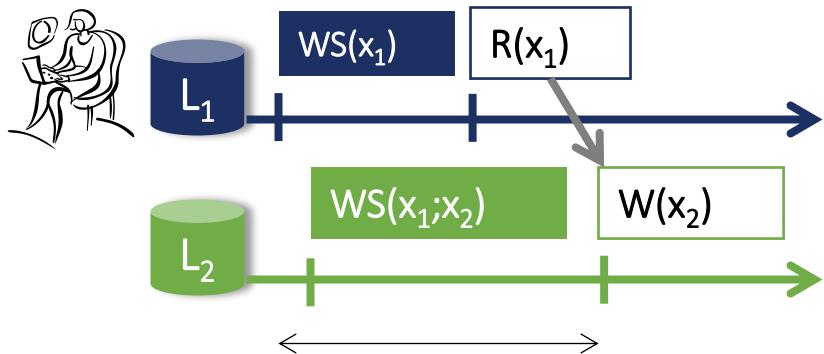
A data-store that does not provide *Read Your Write* consistency

W (x₁) has been left out of WS (x₂), meaning that the effects of the previous write operation by process P have not been propagated to L₂.

Writes Follow Reads

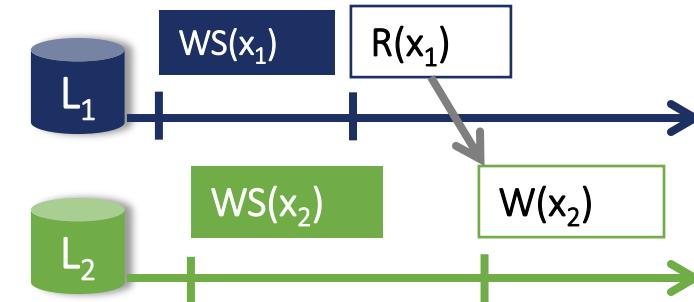
- ▶ A data store is said to provide writes-follow-reads consistency if a process has write operation on a data item x following a previous read operation on x then it is guaranteed to take place on the same or a more recent value of x that was read.
- ▶ Any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process.
- ▶ Example: Suppose a user first reads an article A then posts a response B. By requiring writes-follow-reads consistency, B will be written to any copy only after A has been written.

Writes Follow Reads



W(x₂) operation should be performed only after the all previous writes have been seen

1. Process P performs a write operation on x at local copy L₁, presented as the operation W(x₁).
2. Later, P performs another write operation on x, but this time at L₂, shown as W (x₂).
3. To ensure monotonic-write consistency, the previous write operation at L₁ must have been propagated to L₂.
4. This explains operation W (x₁) at L₂, and why it takes place before W (x₂).



A data-store that does not guarantee Write Follow Read Consistency Model

1. Situation in which monotonic-write consistency is not guaranteed.
2. Missing is the propagation of W(x₁) to copy L₂.
3. No guarantees can be given that the copy of x on which the second write is being performed has the same or more recent value at the time W(x₁) completed at L₁.

Client centric model-Summary

Consistency	Description
Monotonic read	If a process reads the value of a data item x , any successive read operation on x by that process will always return that same value or a more recent value.
Monotonic write	A write operation by a process on a data item x is completed before any successive write operation on x by the same process.
Read your writes	The effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process.
Writes follow reads	A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or more recent values of x than was read.

Replica Management

- ▶ Replica management describes **where, when and by whom** replicas should be placed
- ▶ Two problems under replica management
 1. **Replica-Server Placement** : Decides the best locations to place the replica server that can host data-stores
 2. **Content Replication and Placement** : Finds the best server for placing the contents

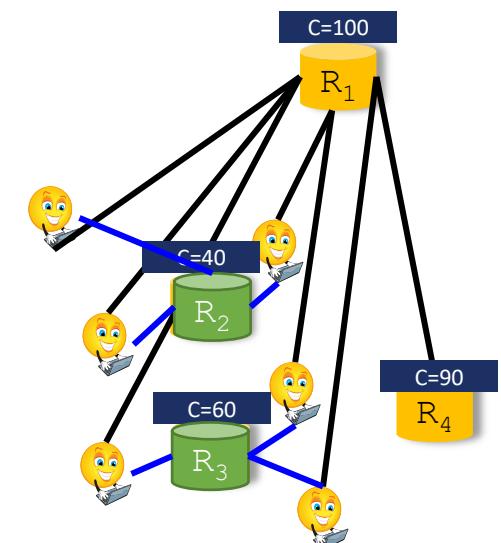
Replica Server Placement

Factors that affect placement of replica servers:

- ▶ What are the possible locations where servers can be placed?
 - Should we place replica servers **close-by** or **distribute** it uniformly?
- ▶ How many replica servers can be placed?
 - What are the trade-offs between placing many **replica servers vs. few**?
- ▶ How many clients are accessing the data from a location?
 - More replicas at locations where most clients access **improves performance and fault-tolerance**
- ▶ If K replicas have to be placed out of N possible locations, find the best K out of N locations($K < N$)

Replica Server Placement – An Example Approach

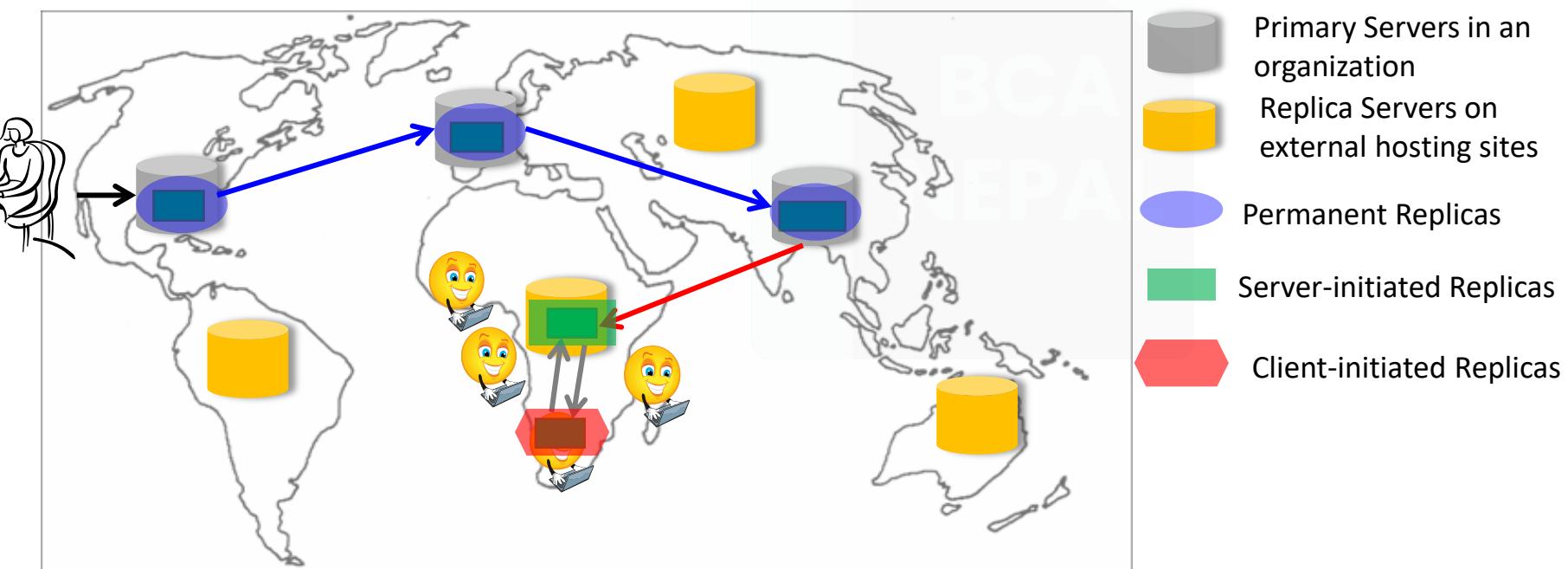
- ▶ Problem: **K replica servers** should be placed on some of the **N possible replica sites** such that
 - ▶ Clients have low-latency/high-bandwidth connections
 - ▶ Suggested a Greedy Approach
1. Evaluate the cost of placing a replica on each of the **N** potential sites
 - Examining the cost of **C** clients connecting to the replica
 - Cost of a link can be **1/bandwidth or latency**
 2. Choose the **lowest-cost site**
 3. In the second iteration, search for **a second replica site** which, in conjunction with the **already selected site**, yields the **lowest cost**
 4. Iterate steps 2,3 and 4 until **K** replicas are chosen



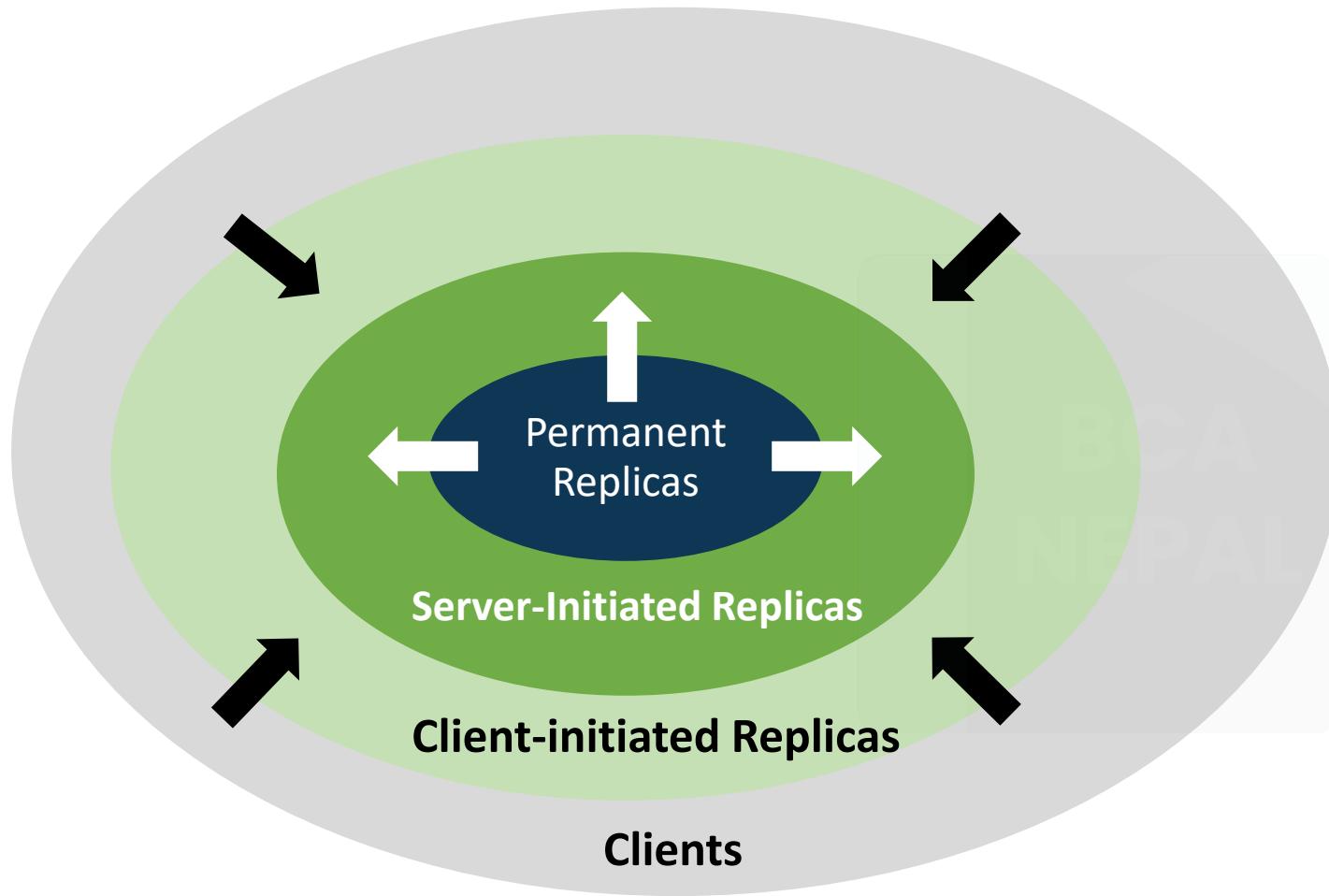
Content Replication and Placement

- ▶ In addition to the server placement, it is important:
 - how, when and by whom different data items (contents) are placed on possible replica servers

- ▶ Identify how webpage replicas are replicated:



Logical Organization of Replicas



→ Server-initiated Replication

→ Client-initiated Replication

Permanent replicas: Process/machine always having a replica

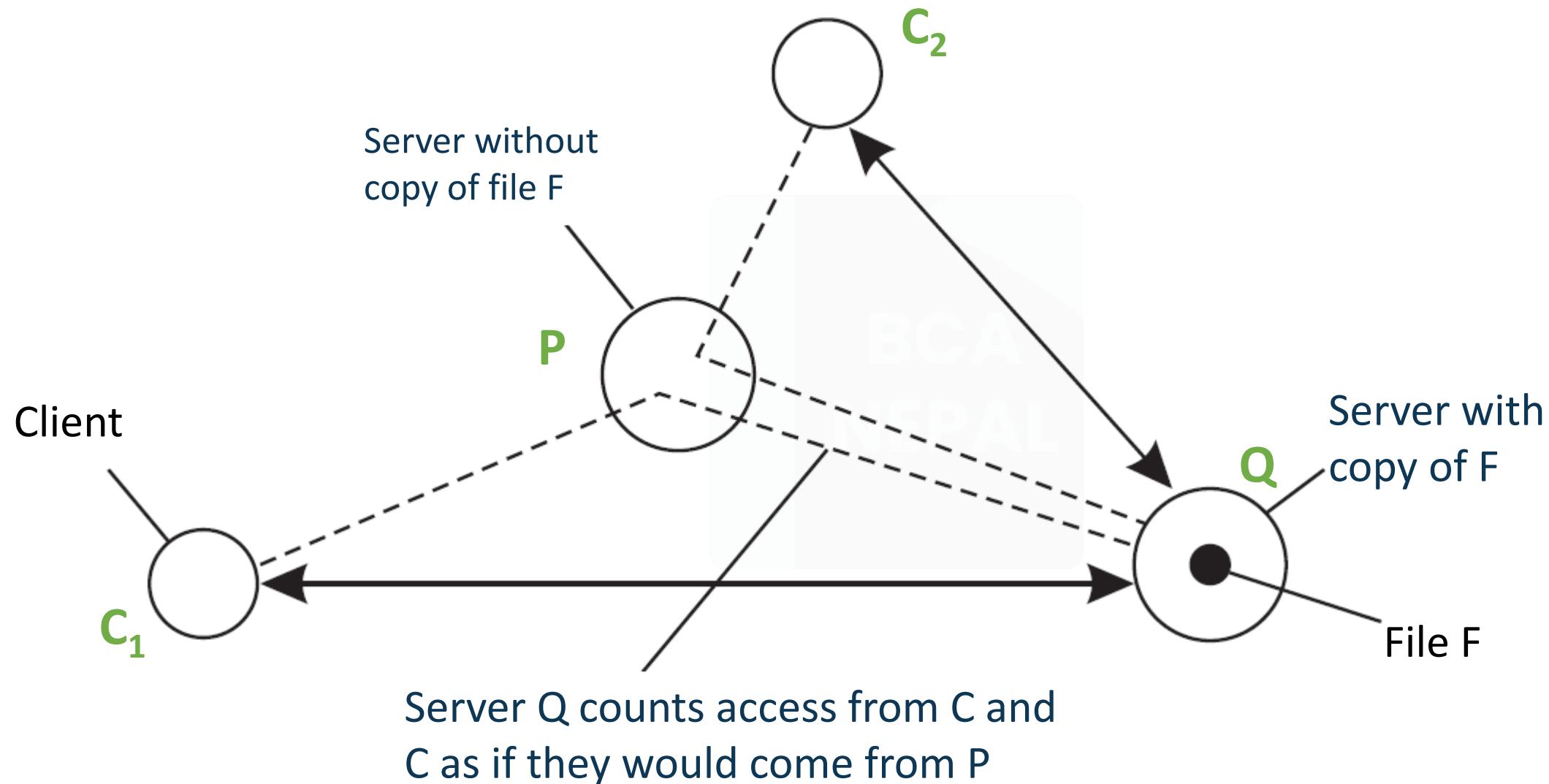
Server-initiated replica: Process that can dynamically host a replica on request of another server in the data store

Client-initiated replica: Process that can dynamically host a replica on request of a client (client cache)

Permanent Replicas

- ▶ They are created by the **data store owner** and function as **permanent storage** for the data.
- ▶ Tend to be small in number (Often is a **single server**), organized as **COWs** (Clusters of Workstations) or **mirrored systems** (cluster or group of mirrors)
- ▶ Permanent replicas are the initial set of replicas that constitute a **distributed data-store**
- ▶ Typically, small in number
- ▶ There can be two types of permanent replicas:
 1. Primary servers
 - One or more servers in an organization
 - Whenever a **request** arrives, it is **forwarded** into one of the **primary servers**
 2. Mirror sites
 - Geographically spread, and replicas are generally statically configured
 - Clients pick one of the **mirror sites** to download the data

Server-initiated Replicas



Server-initiated Replicas

- ▶ They are replicas created in order to enhance the performance of the system at the initiation of the owner of the data-store.
- ▶ Placed on servers maintained by others and close to large concentrations of clients.
- ▶ Typically used by web hosting companies to geographically locate replicas close to where they are needed most. (Often referred to as “push caches”).
- ▶ A third party (provider) owns the secondary replica servers, and they provide hosting service
 - The provider has a collection of servers across the Internet
 - The hosting service dynamically replicates files on different servers
 - Based on the popularity of the file in a region
- ▶ The permanent server chooses to host the data item on different secondary replica servers
- ▶ The scheme is efficient when updates are rare
- ▶ Examples of Server-initiated Replicas :
 - Replicas in Content Delivery Networks (CDNs)

Client-initiated Replicas

- ▶ They are temporary copies created by clients to improve their access to the data (client caches)
- ▶ Examples: Web browser caches and proxy caches
- ▶ Works well assuming, of course, that the cached data does not go stale too soon.
- ▶ Client-initiated replicas are known as client caches
- ▶ Client caches are used only to reduce the access latency of data
 - e.g., Browser caching a web-page locally
- ▶ Typically, managing a cache is entirely the responsibility of a client
 - Occasionally, data-store may inform client when the replica has become stale

Content distribution

- ▶ Replication management also includes propagation of updated content delivery to the replica server (how to update).
- ▶ Information type to be propagated : There are three possibilities for information to be actually propagated.
 1. Propagate only updates notifications
 2. Transmit update data from one copy to another
 3. Propagate update operations to other replicas

Update Propagation: Design Issues

▶ Updates:

- Clients initiate
- They are subsequently forwarded to one of the copies
- From there, the update should be propagated to other Copies, while ensuring the consistency at the same time

▶ What to propagate?

- Only a notification of an update (e.g. invalidation used for caches). Useful when read-to-update ratio is low
- Transfer the modified data from one copy to another. Useful when read-to-update ratio is high.
- Propagate the update operation to other copies. The replicas execute the update operation

Pull versus Push Protocols

▶ Pushing updates:

- Server-based approach; in which updates are propagated regardless whether target asks for it.
- Often used between permanent servers and server initiated replicas (can be used for client caches).
- Used when a high degree of consistency is required and where read-to-update ratio is relatively high

▶ Pulling updates:

- Client-based approach; in which client polls the server to check whether an update is needed.
- Non-shared client caches
- Efficient when read-to-update ratio is low.
- High response time in case of a cache miss

Pull versus Push Protocols

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch update time)	Fetch update time

A comparison between push-based and pull-based protocols in the case of multiple client, single server systems.

Consistency Protocols

- ▶ A consistency protocol describes the implementation of a specific consistency model
- ▶ Three types of consistency protocols:
 - ➔ Primary-based protocols : One primary coordinator is elected to control replication across multiple replicas
 - ➔ Replicated-write protocols : Multiple replicas coordinate to provide consistency guarantees
 - ➔ Cache-coherence protocols : A special case of client-controlled replication

Primary-based protocols

- ▶ In Primary-based protocols, a simple centralized design is used to implement consistency models
 - Each data-item x has an associated “Primary Replica”
 - Primary replica is responsible for coordinating write operations
- ▶ There are two types of Primary-Based Protocol:
 1. Remote-Write.
 2. Local-Write.
- ▶ Example of Primary-based protocols that implement Sequential Consistency Model
 - Remote-Write Protocol

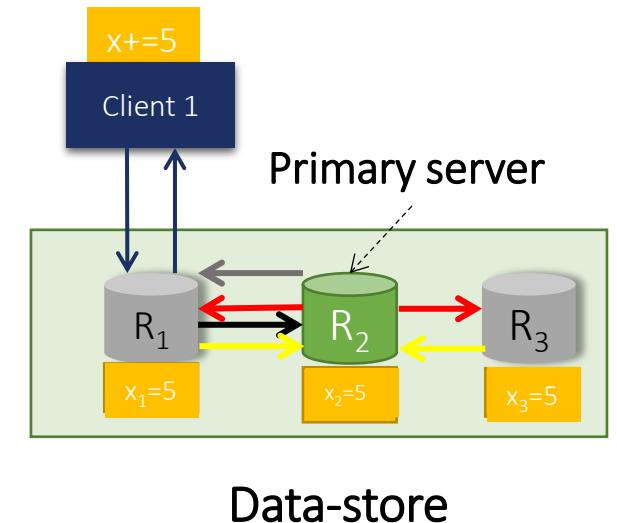
Remote-Write Protocol

► Rules:

- All write operations are forwarded to the primary replica
- Read operations are carried out locally at each replica

► Approach for write ops: (Budhiraja *et al.* [4])

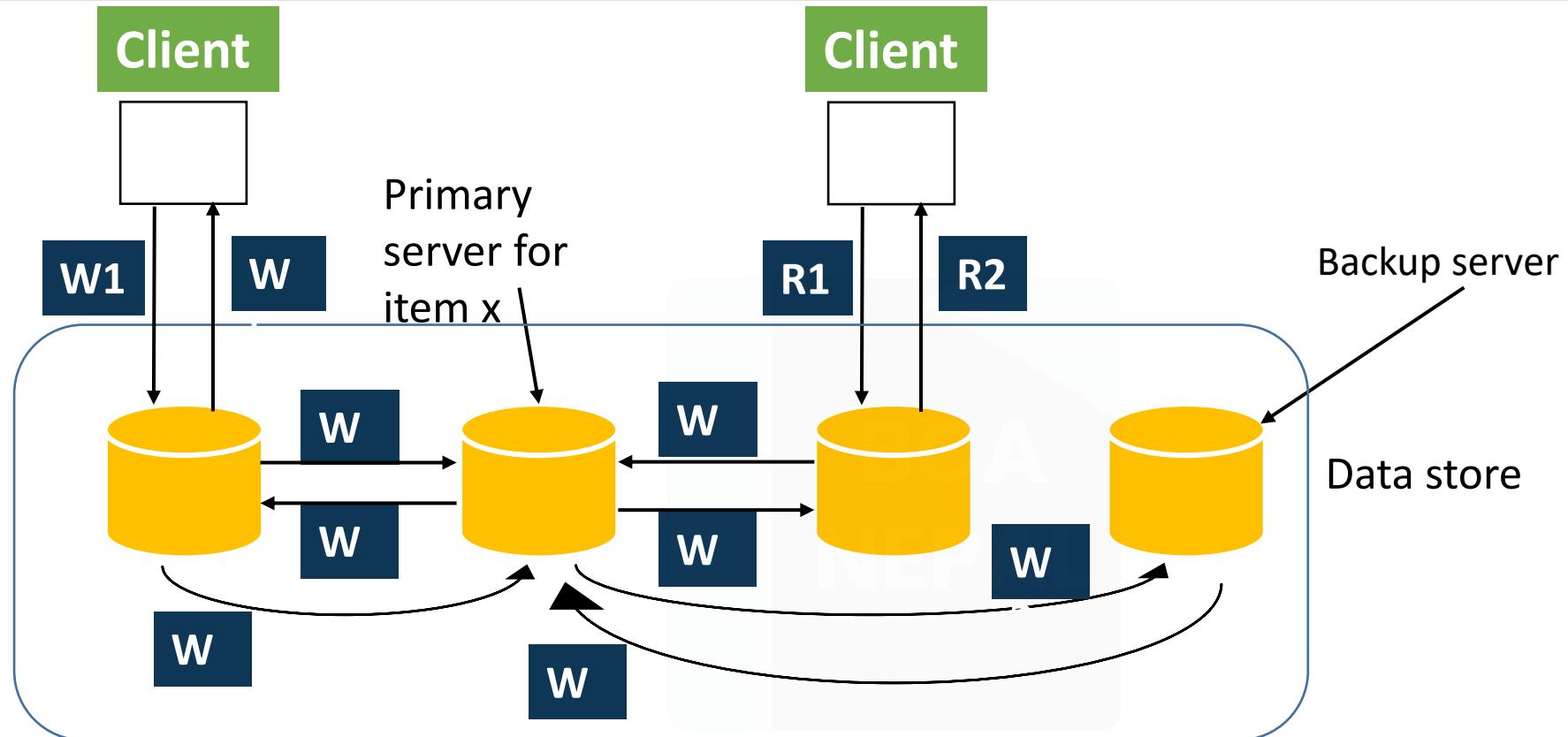
- Client connects to some replica R_C
- If the client issues write operation to R_C :
 - R_C forwards the request to the primary replica R_P
 - R_P updates its local value
 - R_P forwards the update to other replicas R_i
 - Other replicas R_i update, and send an ACK back to R_P
- After R_P receives all ACKs, it informs the R_C that write operation is successful
- R_C acknowledges to the client that write operation was successful



Remote-Write Protocol

- ▶ All write operations need to be forwarded to a fixed single server
- ▶ Read operations can be carried out locally
- ▶ Also called (primary-backup protocol)
- ▶ **Disadvantage:** It may take a relatively long time before the process that initiated the update is allowed to continue, an update is implemented as a blocking operation
- ▶ Alternative: **Non-blocking approach**
- ▶ No-blocking approach:
 - As soon as the primary has updated its local copy of x , it returns an acknowledgment. After that, it tells the backup servers to perform the update as well
 - Advantage: write operations may speed up considerably
 - Disadvantage: fault tolerance, updates may not be backed up by other servers

Remote-Write Protocol



W1. Write request

W2. Forward request to primary

W3. Tell backups to update

W4. Acknowledge update

W5. Acknowledge write completed

R1. Read request

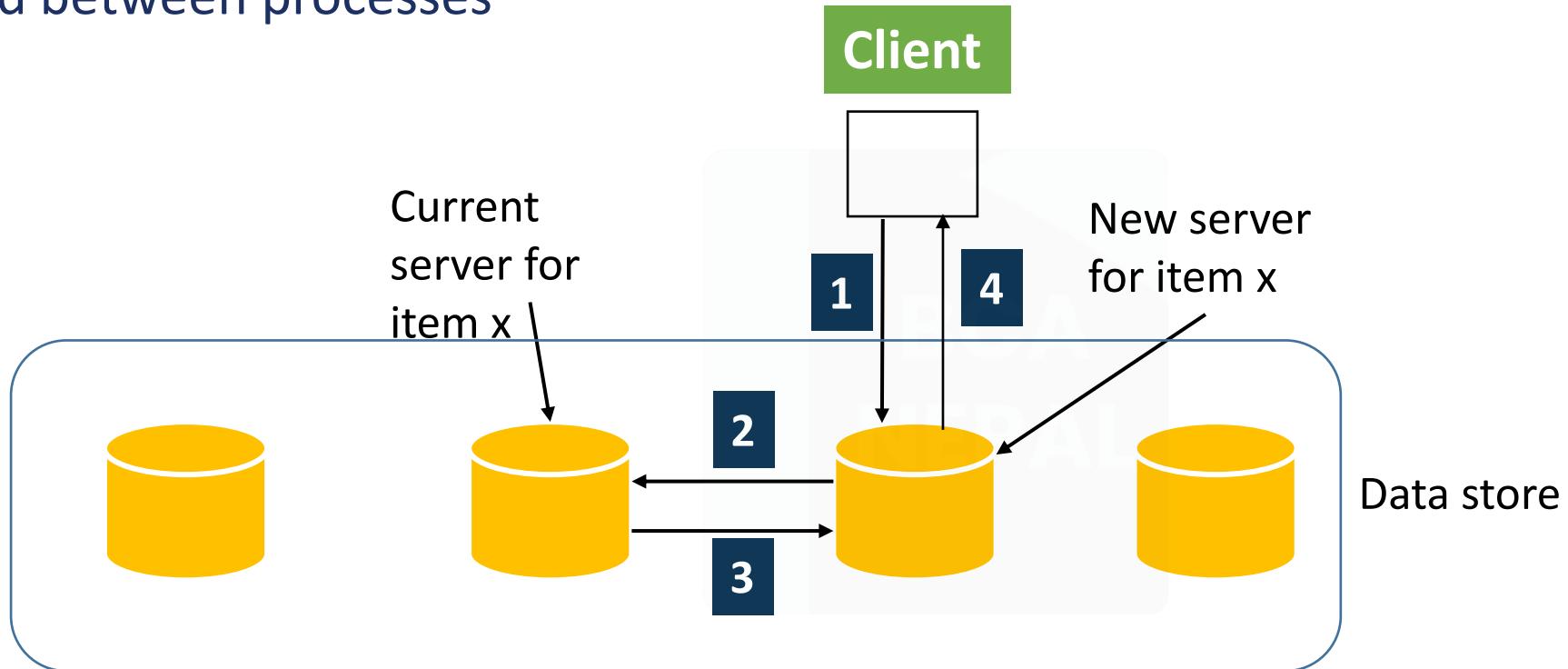
R2. Response to read

Local-Write Protocols

- ▶ When a process wants to update data item x , it locates the primary copy of x , and subsequently moves it to its own location
- ▶ Advantage (in non-blocking protocol only):
- ▶ Multiple, successive write operations can be carried out locally, while reading processes can still access their local copy
- ▶ Updates are propagated to the replicas after the primary has finished with locally performing the updates
- ▶ Example: primary-backup protocol with local writes

Local-Write Protocols

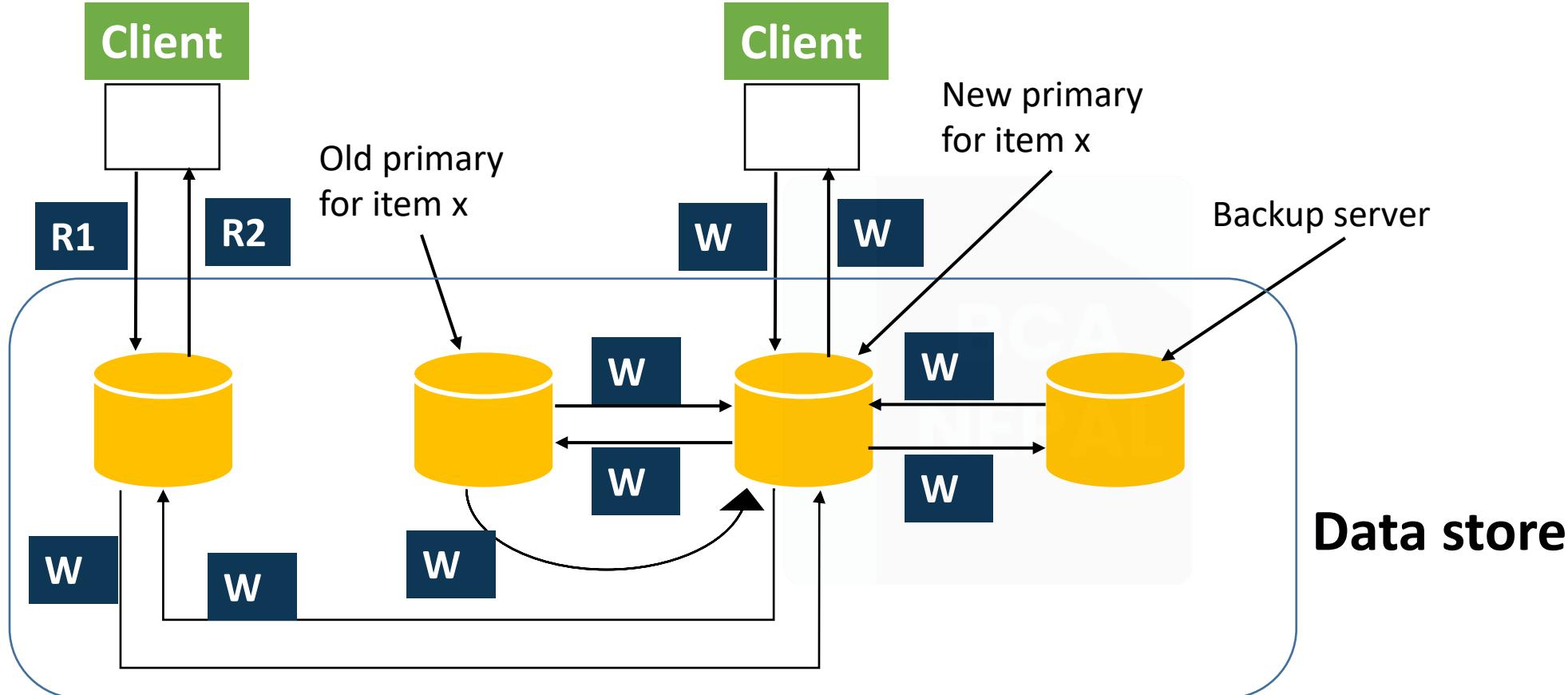
Case-1 : there is only a single copy of each data item x (no replication) a single copy is migrated between processes



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on clients server

Local-Write Protocols

Case 2: (primary back-up): the primary copy migrates



W1. Write request

W2. Move item x to new primary

W3. Acknowledge write completed

W4. Tell backups to update

W5. Acknowledge update

R1. Read request

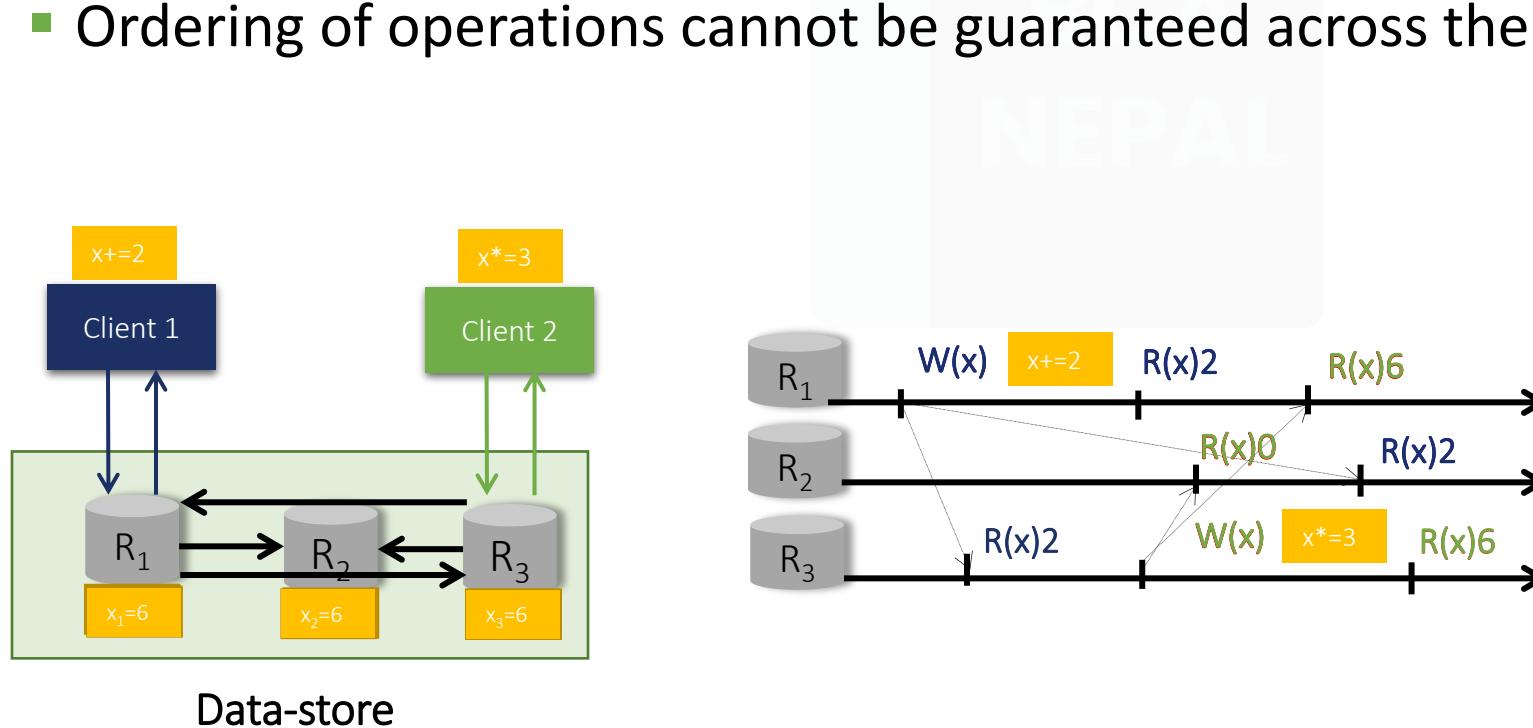
R2. Response to read

Replicated-Write Protocol

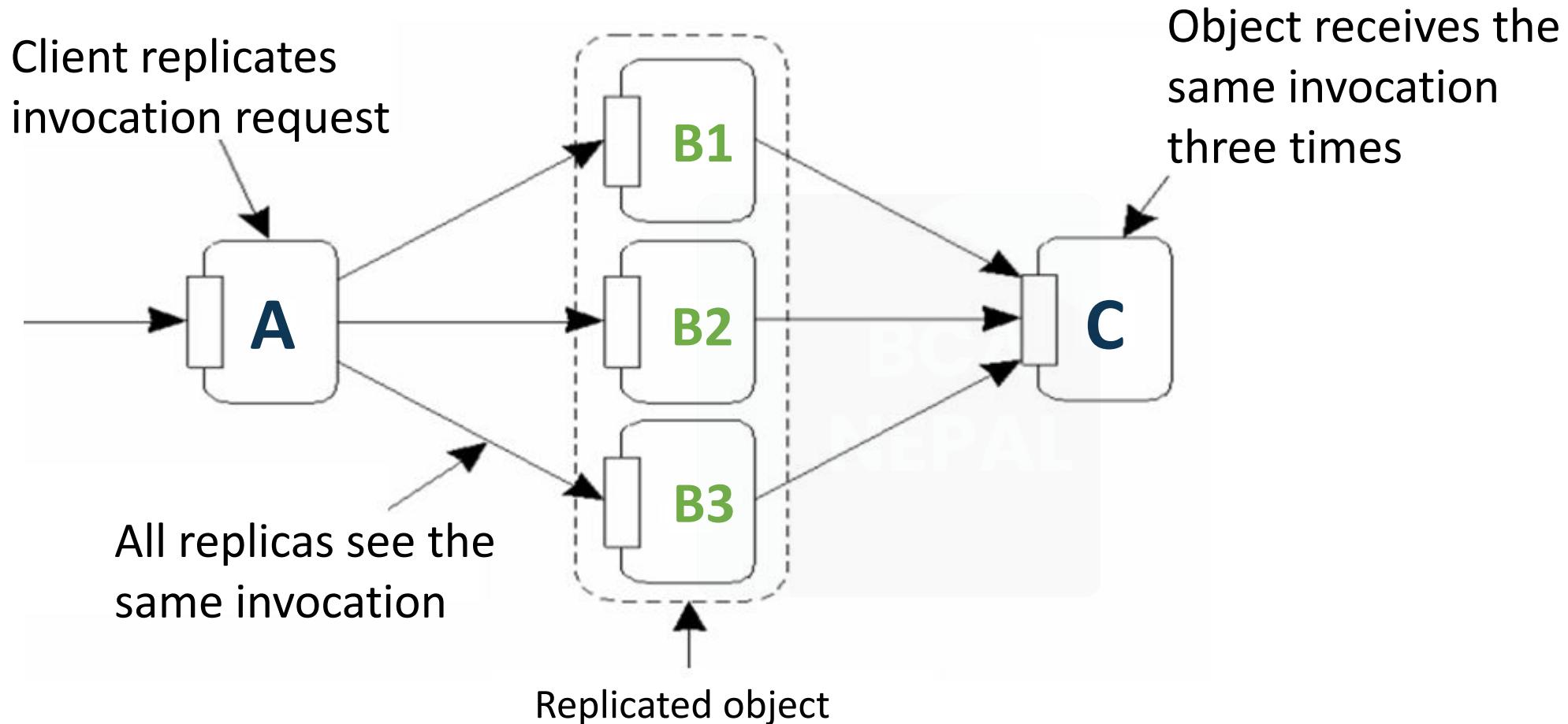
- ▶ In the primary base replication, the write operation is performed on one copy, but in the duplicate write protocol, the write operation is executed for plural copies.
- ▶ There are two ways for the replicated-write protocol,
 1. Active replication: An operation is forwarded to all replicas
 2. Quorum-Based Protocol: By Majority Voting

Active Replication Protocol

- When a client writes at a replica, the replica will send the write operation updates to all other replicas
- Each copy has a process of executing an update operation, and a write operation is executed. At this time, all replicas must execute operations in the same order.
- Challenges with Active Replication
 - Ordering of operations cannot be guaranteed across the replicas



Active Replication: The Problem

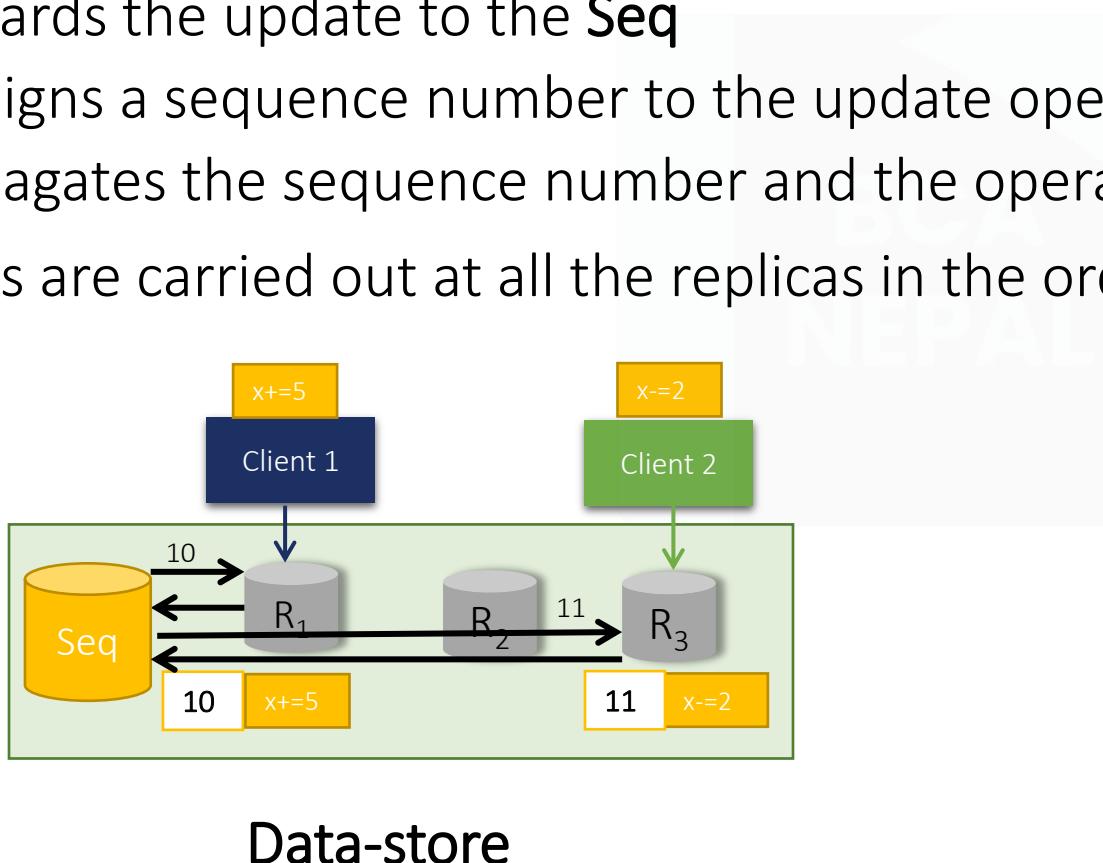


The problem of replicated invocations – ‘B’ is a replicated object (which itself calls ‘C’). When ‘A’ calls ‘B’, how do we ensure ‘C’ isn’t invoked three

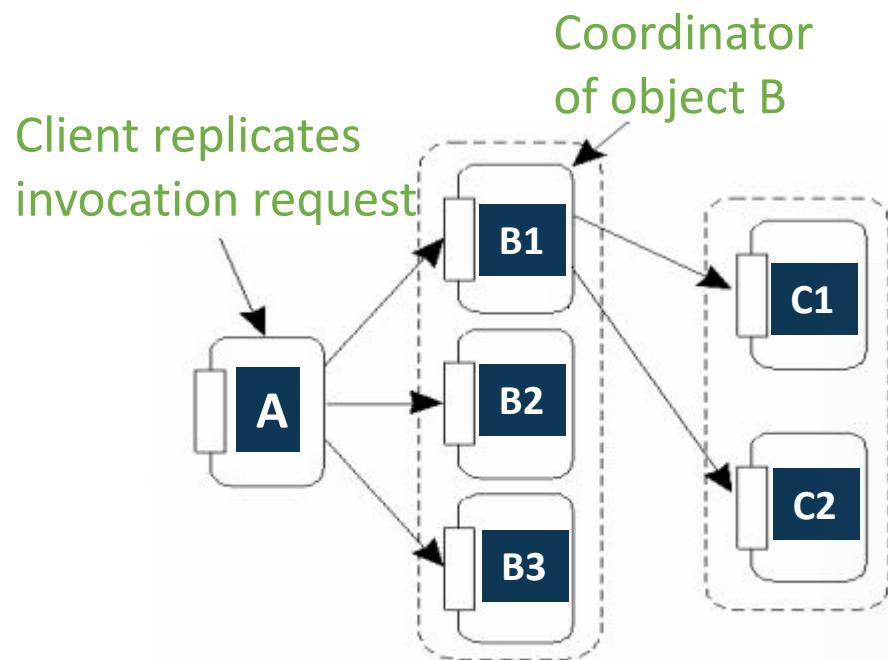
Centralized Active Replication Protocol

Approach

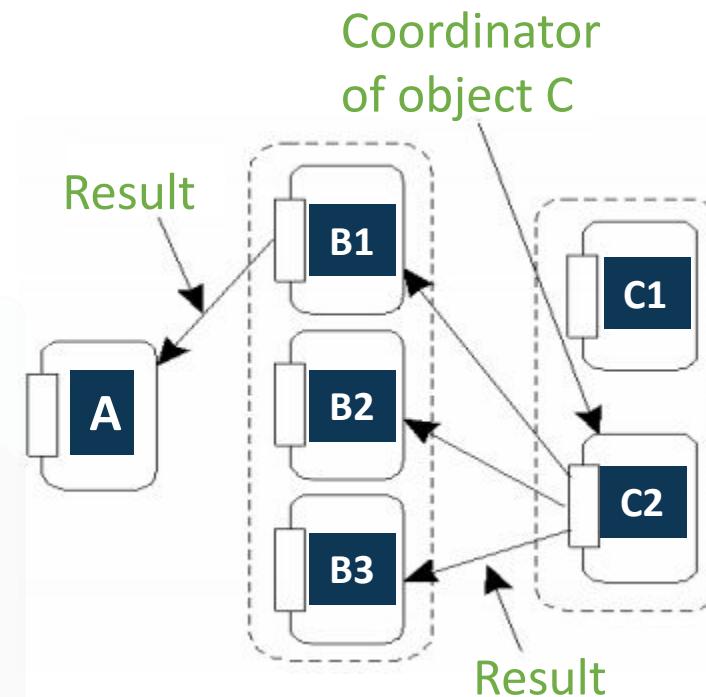
- There is a centralized coordinator called sequencer (**Seq**)
- When a client connects to a replica R_C and issues a write operation
 - R_C forwards the update to the **Seq**
 - Seq** assigns a sequence number to the update operation
 - R_C propagates the sequence number and the operation to other replicas
- Operations are carried out at all the replicas in the order of the sequence number



Active Replication: Solutions



Using a coordinator for 'B', which is responsible for forwarding an invocation request from the replicated object to 'C'.



Returning results from 'C' using the same idea: a coordinator is responsible for returning the result to all 'B's. Note the single result returned to 'A'.

Quorum-Based Protocols

- ▶ Resolves write-write or read-write conflicts
- ▶ Client processes are required to request and acquire the permission of multiple servers before reading and writing a replicated data item.
- ▶ Example protocol (on a distributed file system):
 - A process that wants to update a replicated file first contacts at majority of servers and get them to agree to do the update.
 - Once they agreed, the file is changed and a new version number is associated with the file.
 - To read a replicated file, a client must also contact at least half the servers plus one and ask them to send the version numbers associated with the file.
 - If all version numbers agree then the file is the most recent one

Quorum-Based Protocols

- ▶ A classic method of managing replicated data uses the idea of a quorum. A quorum system consists of a family of subsets of replicas with the property that any two of these subsets overlap.
- ▶ To maintain consistency, read and write operations engage these subsets of replicas, leading to several benefits:
 - First, the load is distributed and the load on each replica is minimized.
 - Second, the fault tolerance improves by minimizing the impact of failures, since the probability that every quorum has a faulty replica is quite low. This improves the availability too.

Quorum-Based Protocols

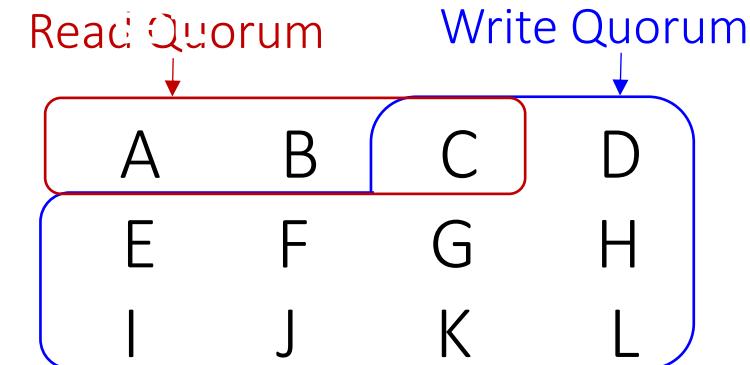
- ▶ Based on voting
- ▶ Read quorum (N_R) : Num. of servers must agree on version num. for a read
- ▶ Write quorum (N_W) : Num. of servers must agree on version num. for a write
- ▶ If N is the total number of replicas,
- ▶ N_R and N_W must fulfill:
 - $N_R + N_W > N$: Prevents read-write conflicts
 - $N_W > N/2$: Prevents write-write conflicts

Quorum-Based Protocols – Example 1

► $N_R = 3$ and $N_W = 10$

- Most recent write quorum consisted of the 10 servers C through L.
- All get the new version and the new version number.
- Any subsequent read quorum of three servers will have to contain at least one member of this set.
- When the client looks at the version numbers, it will know which is most recent and take that one.

A correct choice
of **read and write**



$N_R = 3$ and $N_W = 10$

C1: $N_R + N_W = 13 > N = 12$
→ No RW conflicts

C2: $N_W > 12/2 = 6$
→ No WW conflicts

Cache Coherence Protocols

- ▶ These are a special case, as the cache is typically controlled by the client not the server.
- ▶ Coherence Detection Strategy:
 - When are inconsistencies actually detected?
 - Statically at compile time: extra instructions inserted.
 - Dynamically at runtime: code to check with the server.
- ▶ Coherence Enforcement Strategy
 - How are caches kept consistent?
 - Server Sent: invalidation messages.
 - Update propagation techniques.
- ▶ Combinations are possible.

What about Writes to the Cache?

- ▶ **Read-only Cache**: updates are performed by the server (i.e., pushed) or by the client (i.e., pulled whenever the client notices that the cache is stale).
- ▶ **Write-through Cache**: the client modifies the cache, then sends the updates to the server.
- ▶ **Write-Back Cache**: delay the propagation of updates, allowing multiple updates to be made locally, then sends the most recent to the server (this can have a dramatic positive impact on performance).

Assignment

- ▶ Caching and Replication in Web



Distributed System

Course Code: CACS352
Year/Sem: III/VI

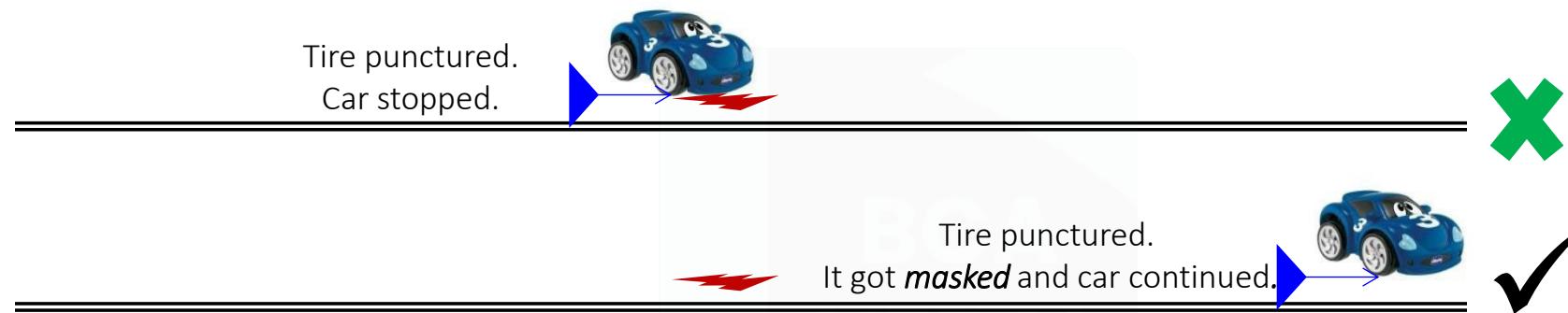
Unit 8. Fault Tolerance

5 Hrs.

- 8.1 Introduction to fault tolerance
- 8.2 Process resilience
- 8.3 Reliable client-server communication
- 8.4 Reliable group communication
- 8.5 Distributed commit
- 8.6 Recovery

Fault-Tolerance

- ▶ Systems can be designed in a way that can automatically recover from partial failures



- ▶ **Fault-tolerance** is the property that enables a system to continue operating properly even if a failure takes place during operation
- ▶ For example, TCP is designed to allow reliable two-way communications in packet-switched networks, even in the presence of communication links that are imperfect or overloaded

What is a Failure?

- ▶ A failure is a deviation from a specified behavior
 - E.g., Pressing brake pedal does not stop car → brake failure (could be catastrophic!)
 - E.g., Read of a disk sector does not return content → disk failure (not necessarily catastrophic)
- ▶ A system is said to “**fail**” when it cannot meet its promises.
- ▶ **A failure is brought about by the existence of “errors” in the system.**
- ▶ **The cause of an error is a “fault”.**
- ▶ Many failures are due to incorrect specified behavior
 - This typically happens when the designer misses addressing a scenario that makes the system perform incorrectly
 - It is especially true in complex systems with many subtle interactions

Failures, Due to What?

- ▶ Failures can happen due to a variety of reasons:
 - **Hardware faults**
 - **Software bugs**
 - **Operator errors**
 - **Network errors/outages**
- ▶ **A system is said to fail when it cannot meet its promises**

Failures in Distributed Systems

- ▶ A characteristic feature of distributed systems that distinguishes them from **single-machine systems** is the notion of **partial failure**
- ▶ A partial failure may happen when a **component** in a distributed system fails
- ▶ This failure may affect the proper operation of other components, while at the same time leaving yet other **components unaffected**



Fault Tolerance Basic Concepts

- ▶ Dealing successfully with partial failure within a Distributed System.
- ▶ Being fault tolerant is strongly related to what are called dependable systems.
- ▶ Dependability implies the following:
 - 1. Availability**
 - 2. Reliability**
 - 3. Safety**
 - 4. Maintainability**

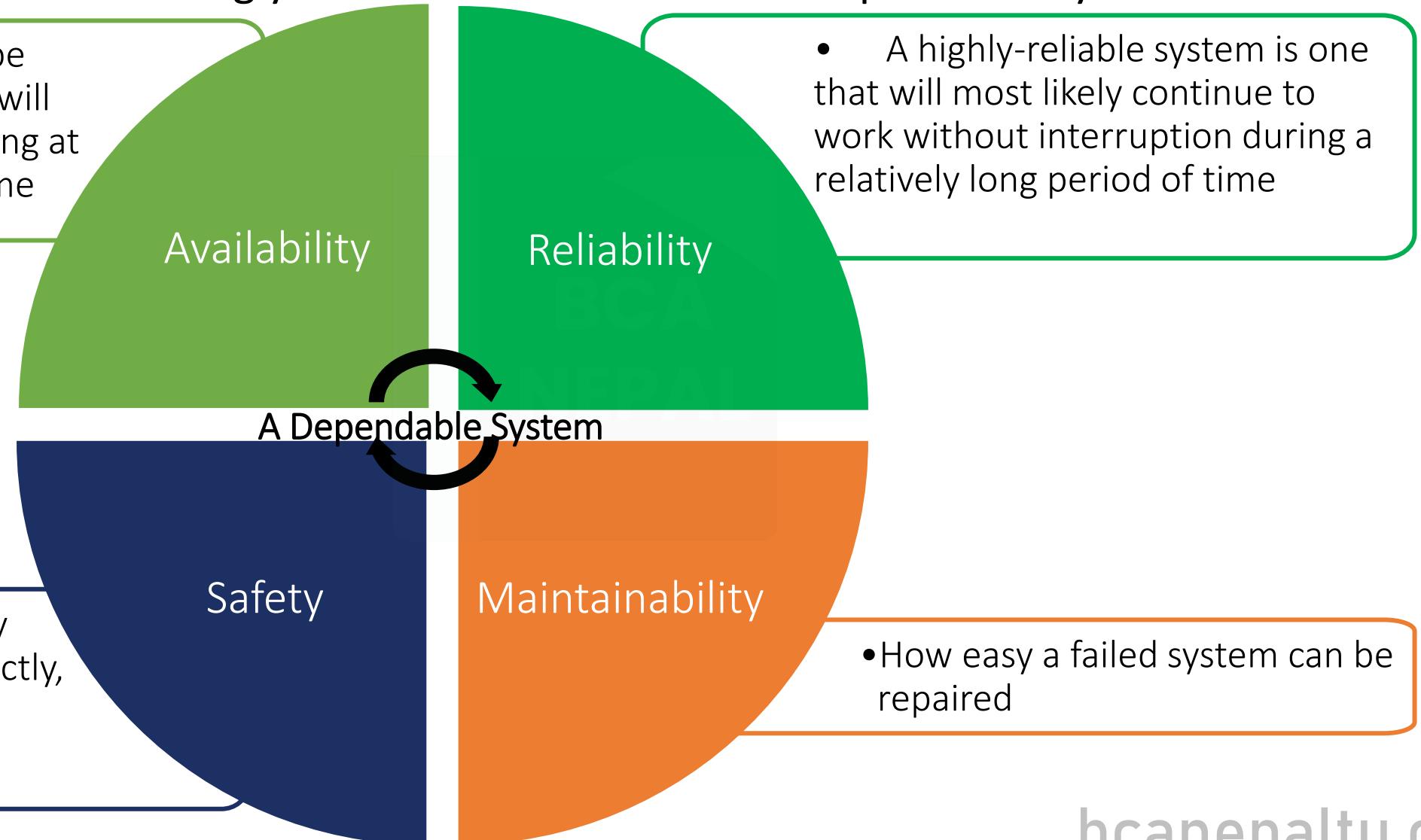
Dependability Basic Concepts

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

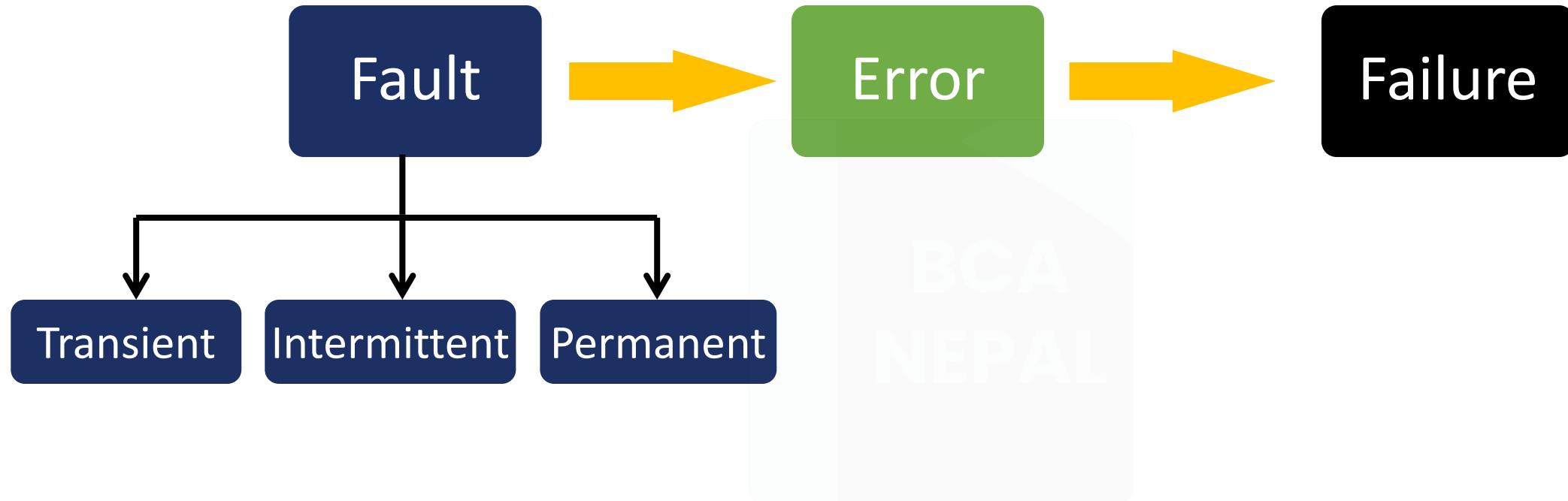
Dependable Systems

► Being fault tolerant is strongly related to what is called a dependable system

- A system is said to be highly available if it will be most likely working at a given instant in time



Faults, Errors and Failures



Failure Models

Type of Failure	Description
Crash Failure	A server halts but was <i>working correctly until it stopped</i>
Omission Failure <ul style="list-style-type: none">▪ ReceiveOmission▪ SendOmission	A server fails to respond to incoming requests <ul style="list-style-type: none">▪ A server fails to receive incoming messages▪ A server fails to send messages
Timing Failure	A server's response lies outside the specified time interval
Response Failure <ul style="list-style-type: none">▪ ValueFailure▪ StateTransitionFailure	A server's response is incorrect <ul style="list-style-type: none">▪ The value of the response is wrong▪ The server deviates from the correct flow of control
Byzantine Failure	A server may produce arbitrary responses at arbitrary times

Failure Characteristics

▶ Transient Failures:

- Also referred to as “soft failures” or “Heisenbugs”
- Occur temporarily then disappear
- Manifested only in a very unlikely combination of circumstances
- Typically go away upon rolling back and/or retrying/rebooting
- E.g., Frozen keyboard or window, race conditions and deadlocks, etc.

▶ Intermittent Fault:

- Occurs, vanishes, reappears; but: follows no real pattern (worst kind).

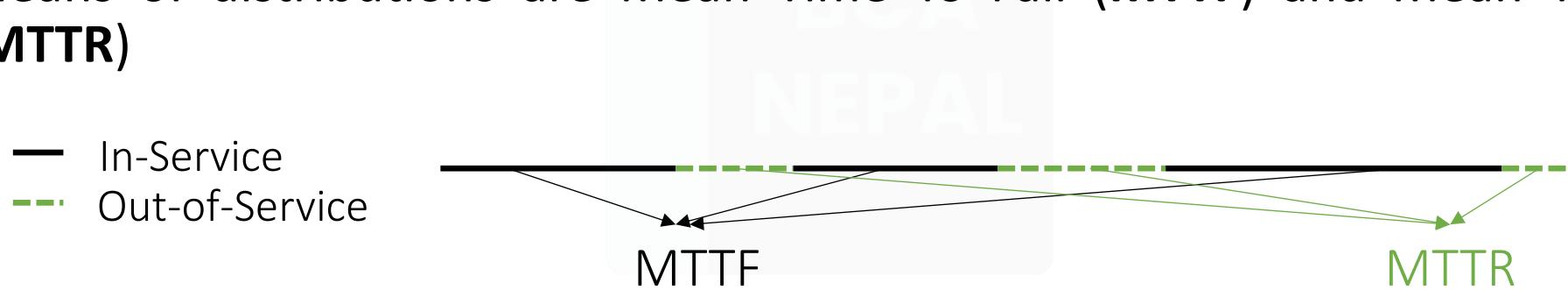
▶ Permanent Fault:

- Once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally.

Failure Characteristics

► Persistent Failures:

- Persist until explicitly repaired
- Retrying does not help
- E.g., Burnt-out chips, software bugs, crashed disks, broken Ethernet cable, etc.
- Durations of failures and repairs are random variables
- Means of distributions are Mean Time To Fail (**MTTF**) and Mean Time To Repair (**MTTR**)

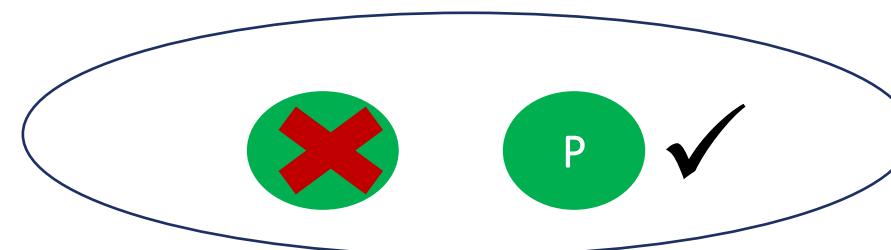


Fault Tolerance Requirements

- ▶ A robust fault tolerant system requires:
 - ➔ ***No single point of failure***
 - ➔ ***Fault isolation/containment to the failing component***
 - ➔ ***Availability of reversion modes***

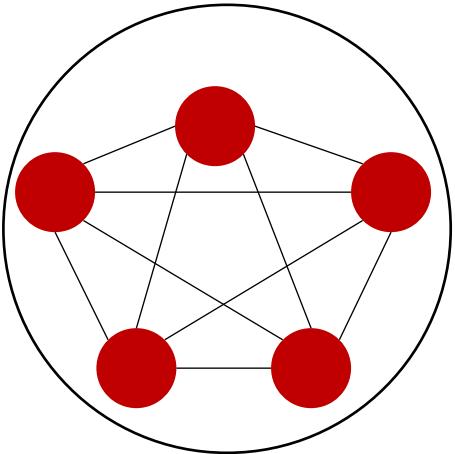
Process Resilience

- ▶ Processes can be made fault tolerant by arranging to have a **group of processes**, with each member of the group being **identical**.
- ▶ A message sent to the group is delivered to all of the “**copies**” of the process (the group members), and then **only one** of them performs the required service.
- ▶ If one of the processes **fail**, it is assumed that one of the **others** will still be able to function (and service any pending request or operation).
- ▶ **The key approach to tolerating a faulty process is to organize several identical processes into a group**
- ▶ If one process in a group fails, hopefully some other process can take over



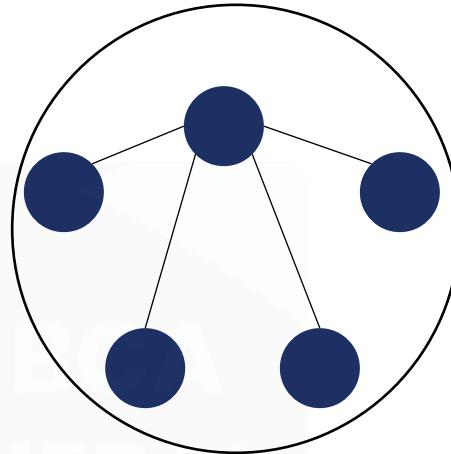
Flat Versus Hierarchical Groups

Flat Group



- ▶ All the processes are **equal**, decisions are made collectively.
- ▶ **No single point-of-failure**, however: decision making is complicated as **consensus(a general agreement)** is required

Hierarchical Group



- ▶ One of the processes is **elected** to be the **coordinator**, which selects another process (a **worker**) to perform the operation.
- ▶ **single point-of-failure**, however: decisions are easily and quickly made by the coordinator without first having to get **consensus**.

Reliable Client/Server Communications

- ▶ In addition to process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures.
- ▶ In practice, the focus is on masking **crash** and **omission failures**.
- ▶ For example: the point-to-point **TCP** masks omission failures by guarding against lost messages using **ACKs** and **retransmissions**. However, it performs poorly when a crash occurs (although a DS may try to mask a TCP crash by automatically re-establishing the lost connection).

RPC Semantics and Failures

- ▶ The RPC mechanism works well as long as both the **client and server function perfectly**.
- ▶ Five classes of RPC failure can be identified:
 1. The client cannot **locate** the server, so **no request** can be sent.
 2. The **client's request** to the server is **lost**, so **no response** is returned by the **server** to the waiting client.
 3. The **server crashes** after receiving the request, and the service request is **left acknowledged**, but **undone**.
 4. The **server's reply** is lost on its way to the client, the service has completed, but the **results never arrive at the client**
 5. The **client crashes** after sending its request, and the server sends a reply to a **newly-restarted** client that may not be expecting it.

RPC Semantic in the presence of Failure

► Client unable to locate server:

- One possible solution for the client being unable to locate the server is to have do Operation raise an exception at the client side
- Problem- When server goes down
- Solution- error raise an exception
 - ➔ Java- division by zero
 - ➔ C- signal handlers

► Request message from client to server is lost:

- The operating system starts a timer when the stub is generated and sends a request. If response is not received before timer expires, then a new request is sent.
- Lost message – works fine on retransmission.
- If request is not lost, we should make sure server knows that its a retransmission.

RPC Semantic in the presence of Failure

► Reply message from server to the client is lost:

- Some messages can be retransmitted any number of times without any loss.
- Some retranmissions cause severe loss.
- Solution- client assigns sequence number on requests made by client. So server has to maintain the sequence number while sending the reply.

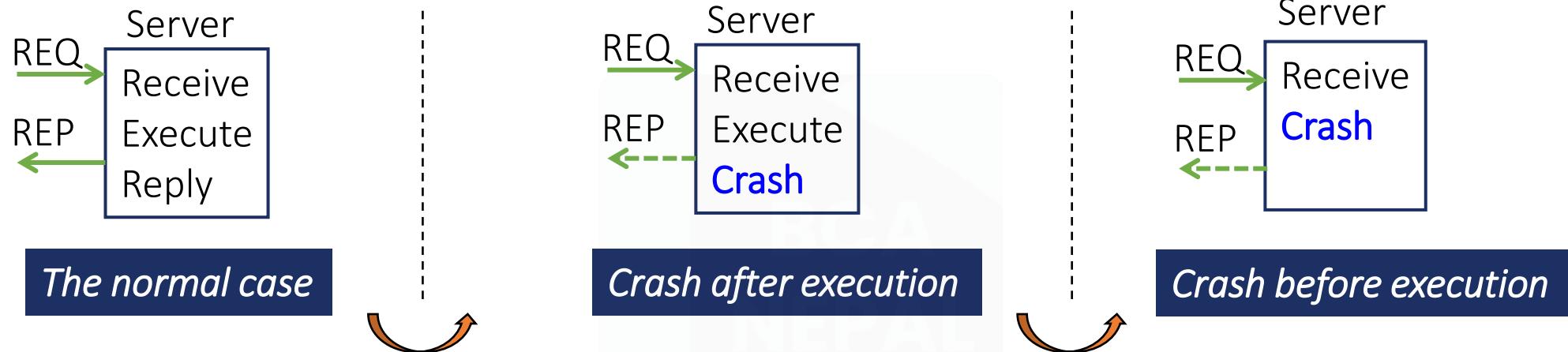
► Client crashes after sending request:

- When a client crashes, and when an 'old' reply arrives, such a reply is known as an orphan.
- Four orphan solutions have been proposed:
 - **Extermination**: Use logging to explicitly kill off an orphan after a client reboot
 - **Reincarnation**: Use broadcasting to kill all remote computations on a client's behalf after rebooting and getting a new epoch number
 - **Gentle Reincarnation**: After an epoch broadcast comes in, a machine kills a computation only if its owner cannot be located anywhere
 - **Expiration**: each remote invocation is given a standard amount of time to fulfill the job

RPC Semantic in the presence of Failure

► Server crashes after receiving request

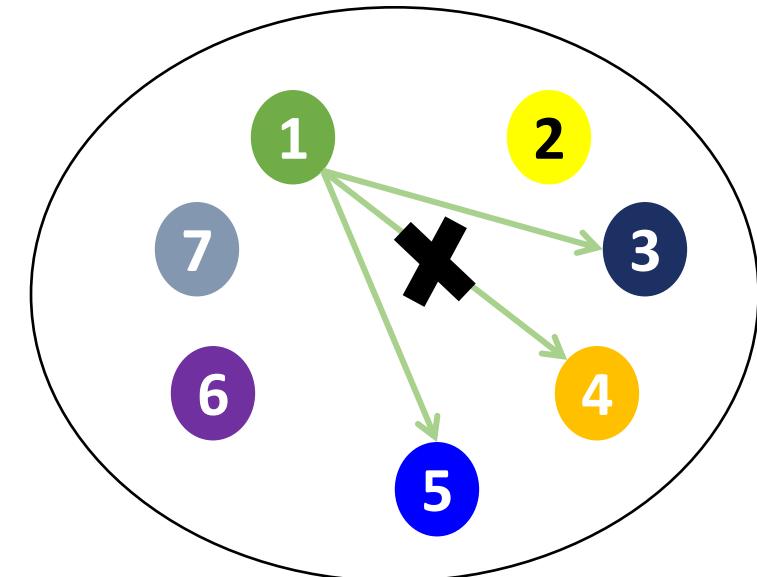
- The client cannot tell if the crash occurred before or after the request is carried out



- A client's remote operation consists of printing some text at a server
- When a client issues a request, it receives an ACK that the request has been delivered to the server
- The server sends a completion message to the client when the text is printed
- Remote operation:** print some text and (when done) send a completion message.

Reliable Group Communication

- As we considered reliable request-reply communication, we need also to consider reliable **multicasting** services
- E.g., Election algorithms use multicasting schemes**
- Reliable multicast services guarantee that all messages are **delivered to all** members of a process group.
- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution scales poorly as the group membership grows.
 - What happens if a process joins the group **during communication?**
 - Worse:** what happens if the sender of the multiple, reliable point-to-point channels crashes half way through sending the messages?



Reliable Multicasting

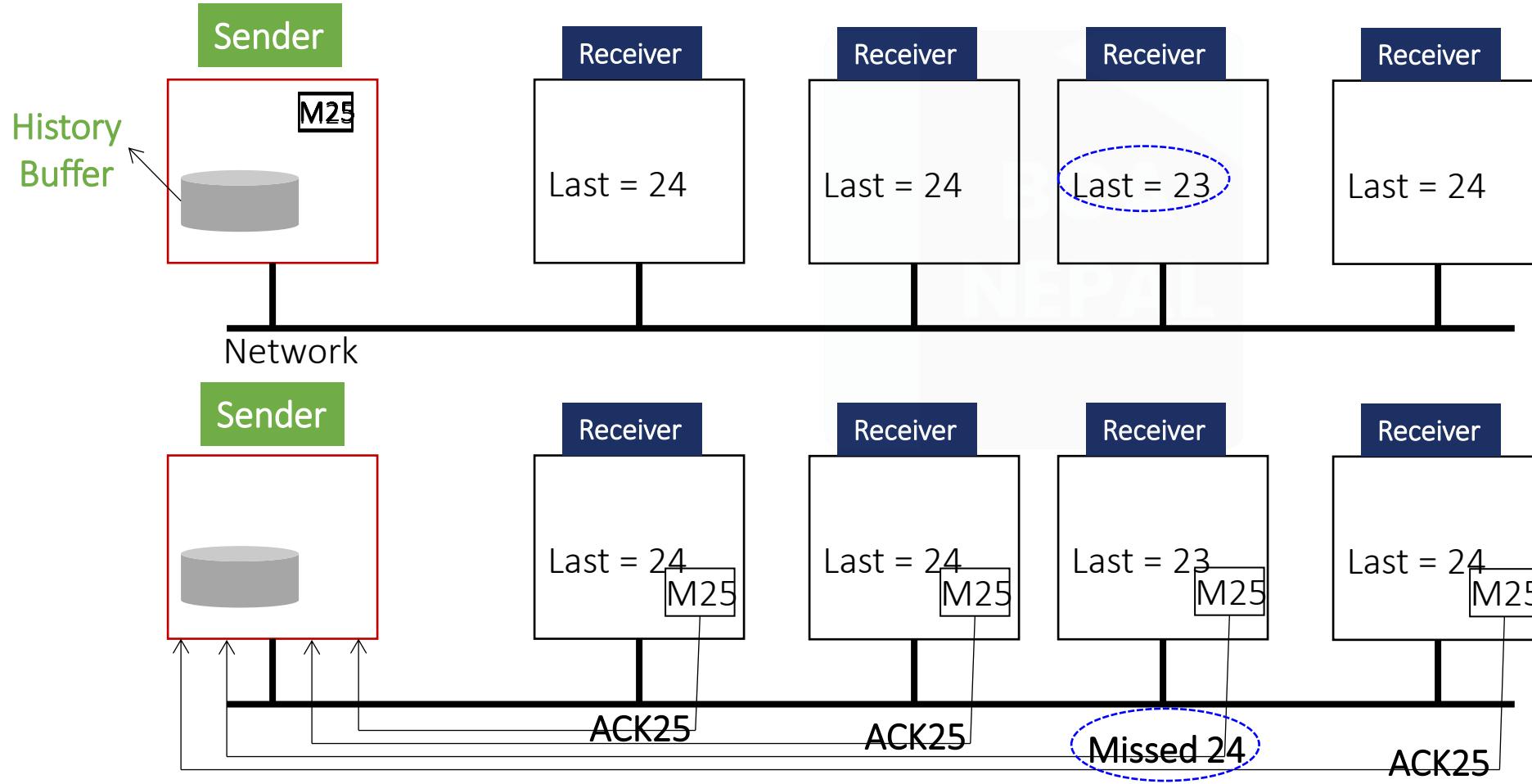
- ▶ Reliable multicasting indicates that a message that is sent to a process group should be delivered to each member of that group
- ▶ A distinction should be made between:
 - Reliable communication in the presence of **faulty processes**
 - Reliable communication when processes are assumed to **operate correctly**
- ▶ In the presence of **faulty processes**, **multicasting** is considered to be reliable when it can be guaranteed that all **non-faulty group members receive the message**

A Basic Reliable-Multicasting Scheme

- ▶ What happens if during communication (i.e., a message is being delivered) a process **P** joins a group?
 - Should **P** also receive the message?
- ▶ What happens if a (sending) process crashes during communication?
- ▶ What about message ordering?

Reliable Multicasting with Feedback Messages

- Consider the case when a **single sender S** wants to **multicast** a message to **multiple receivers**
- An **S**'s multicast message may be lost part way and delivered to some, but not to all, of the intended receivers. Assume that messages are received in the same order as they are sent



Scalability in Reliable Multicasting

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

Distributed Commit (saving)

- ▶ Atomic multicasting problem is an example of a more general problem, known as **distributed commit**
- ▶ The distributed commit problem involves having an operation being performed by each member of a process group, or none at all
 - With reliable multicasting, the operation is the delivery of a message
 - With distributed transactions, the operation may be the commit of a transaction at a single site that takes part in the transaction
- ▶ Distributed commit is often established by means of a **coordinator** and **participants**

One-Phase Commit Protocol

- ▶ In a simple scheme, **a coordinator can tell all participants** whether or not to (locally) perform the operation in question
- ▶ This scheme is referred to as a **one-phase commit protocol**
- ▶ The one-phase commit protocol has a main **drawback** that **if one of the participants cannot actually perform the operation, there is no way to tell the coordinator**
- ▶ In practice, more sophisticated schemes are needed. The most common utilized one is the **two-phase commit protocol**

Two-Phase Commit Protocol

- ▶ Assuming that no failures occur, the two-phase commit protocol (2PC) consists of the following two phases, each consisting of two steps:

Phase I: Voting Phase

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

Two-Phase Commit Protocol

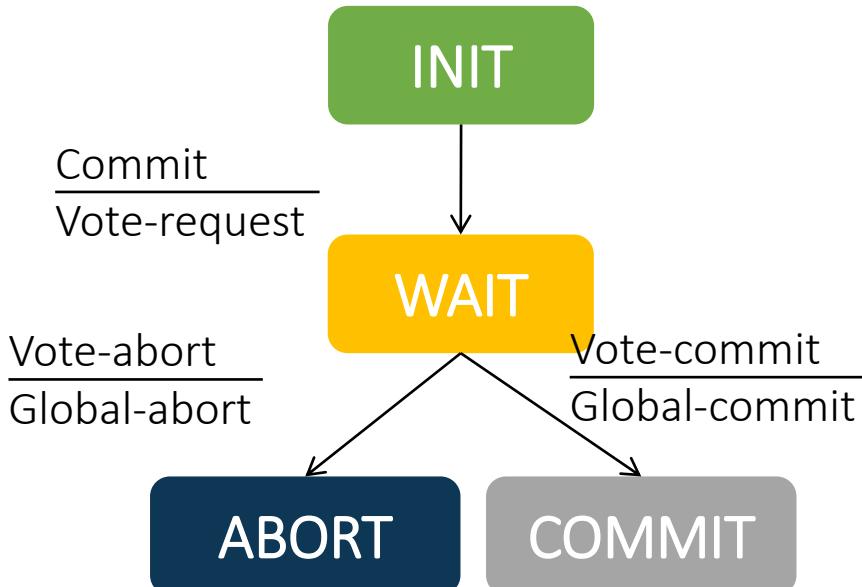
Phase II: Decision Phase

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

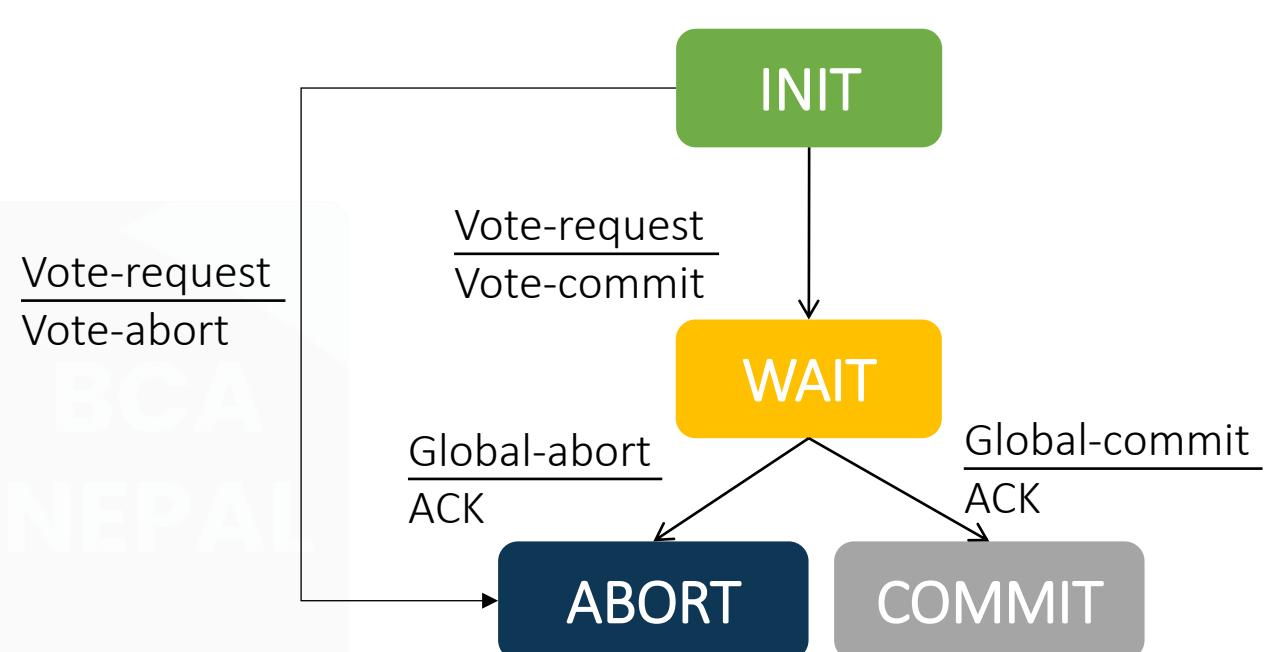
Step 1

Step 2

2PC Finite State Machines



The finite state machine for the coordinator in 2PC



The finite state machine for a participant in 2PC



Recovery Strategies

- ▶ So far, we have mainly concentrated on algorithms that allow us to tolerate faults
- ▶ However, once a failure has occurred, it is essential that the process where the failure has happened can recover to a correct state
- ▶ Focus on:
 - What it actually means to recover to a correct state
 - When and how the state of a distributed system can be recorded and recovered, by means of check pointing and message logging

Recovery Strategies

- ▶ Once a failure has occurred, it is essential that the process where the failure happened recovers to a correct state.
- ▶ Recovery from an error is fundamental to fault tolerance.
- ▶ The idea of error recovery is to replace an erroneous state with an error-free state
- ▶ Two main forms of recovery:
 1. Backward recovery
 2. Forward recovery

Recovery Strategies

Backward recovery:

- ▶ Return the system to some previous correct state (using checkpoints), then continue executing.
- ▶ Problem:
 - Restoring a system or a process to a previous state is generally expensive in terms of performance
 - Some states can never be rolled back

Forward Recovery:

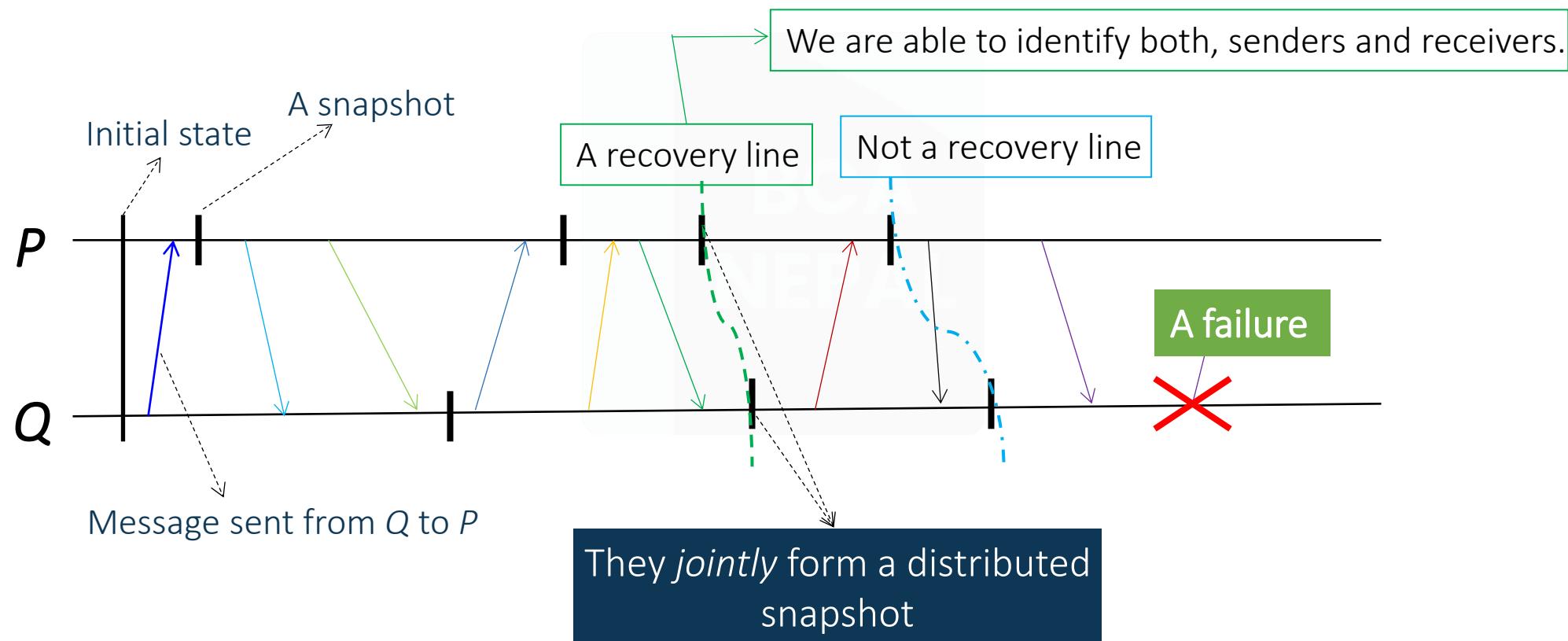
- ▶ Bring the system into a correct state, from which it can then continue to execute
- ▶ Forward recovery is typically faster than backward recovery but requires that it has to be known in advance which errors may occur
- ▶ Problem:
 - In order to work, all potential errors need to be accounted for up-front.
 - When an error occurs, the recovery mechanism then knows what to do to bring the system forward to a correct state.

Checkpointing

- ▶ In a fault-tolerant distributed system, backward recovery requires that the system regularly saves its state onto a stable storage
- ▶ This process is referred to as checkpointing
- ▶ In particular, checkpointing consists of storing a distributed snapshot of the current application state (i.e., a consistent global state), and later on, use it for restarting the execution in case of a failure
- ▶ In a distributed snapshot, if a process P has recorded the receipt of a message, then there should be also a process Q that has recorded the sending of that message

Recovery Line

- In a distributed snapshot, if a process P has recorded the receipt of a message, then there should be also a process Q that has recorded the sending of that message



Checkpointing

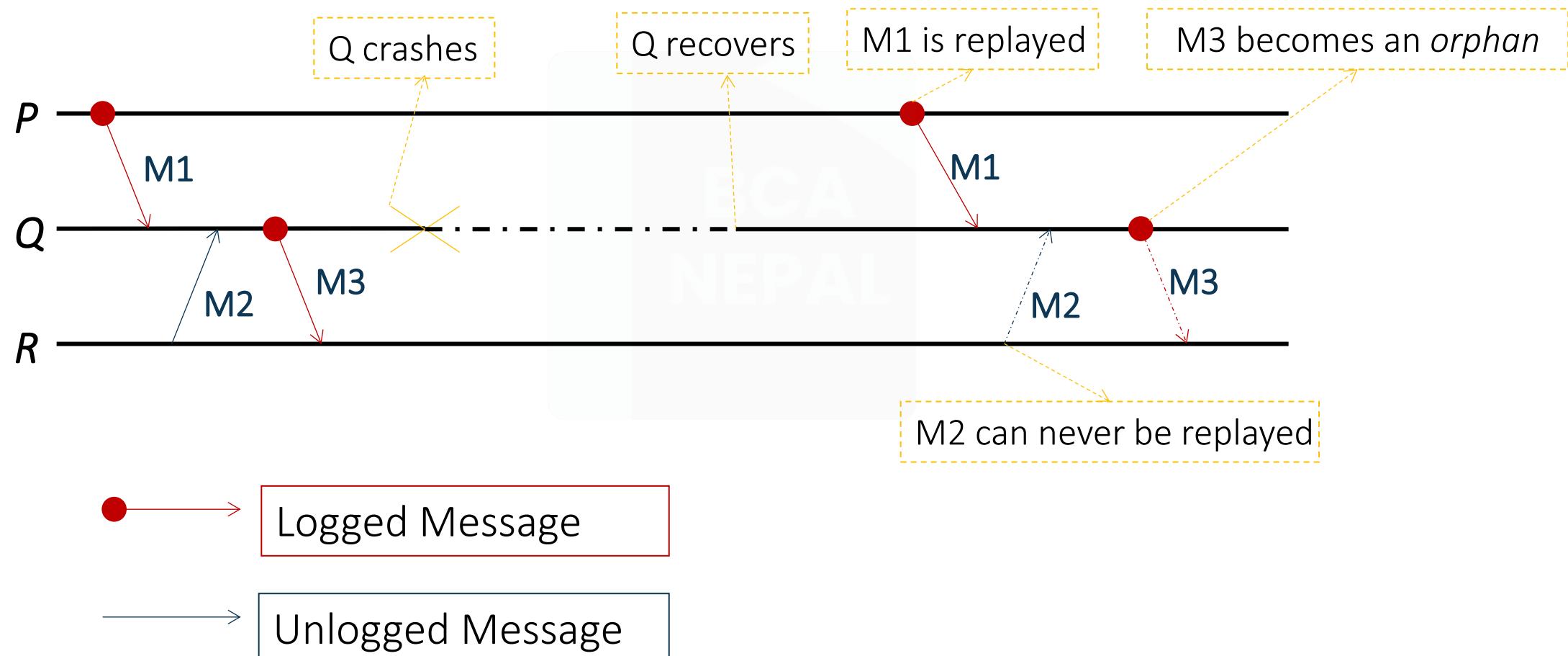
- ▶ Checkpointing can be of two types:
 1. Independent Checkpointing: each process simply records its local state from time to time in an uncoordinated fashion
 2. Coordinated Checkpointing: all processes synchronize to jointly write their states to local stable storages

Message Logging

- ▶ Considering that checkpointing is an expensive operation, techniques have been sought to reduce the number of checkpoints, but still enable recovery
- ▶ An important technique in distributed systems is message logging
- ▶ The basic idea is that if transmission of messages can be replayed, we can still reach a globally consistent state but without having to restore that state from stable storage
- ▶ In practice, the combination of having fewer checkpoints and message logging is more efficient than having to take many checkpoints
- ▶ Message logging can be of two types:
 1. **Sender-based logging**: A process can log its messages before sending them off
 2. **Receiver-based logging**: A receiving process can first log an incoming message before delivering it to the application
- ▶ When a sending or a receiving process crashes, it can restore the most recently checkpointed state, and from there on **replay** the logged messages (important for non-deterministic behaviors)

Replay of Messages and Orphan Processes

- Incorrect replay of messages after recovery can lead to orphan processes. This should be avoided



► Softcopy Format of Marks Distribution



Distributed System

Course Code: CACS352

Year/Sem: III/VI

Unit-9

Security

Unit 9. Security

- 9.1 Introduction to security
- 9.2 Secure channels
- 9.3 Access control
- 9.4 Secure naming
- 9.5 Security Management

4 Hrs.

9.1 INTRODUCTION TO SECURITY

- We start our description of security in distributed systems by taking a look at some general security issues.
- First, it is necessary to define what a secure system is. We distinguish security *policies* from security *mechanisms*. and take a look at the Globus wide-area system for which a security policy has been explicitly formulated.
- Our second concern is to consider some general design issues for secure systems.
- Finally, we briefly discuss some cryptographic algorithms, which play a key role in the design of security protocols.

- **Security** in a computer system is strongly related to the notion of dependability.
- Informally, a dependable computer system is one that we justifiably trust to deliver its services (Laprie, 1995).
- As mentioned in Chap. 7, dependability includes availability, reliability, safety, and maintainability.
- However, if we are to put our trust in a computer system, then confidentiality and integrity should also be taken into account.
- **Confidentiality** refers to the property of a computer system whereby its information is disclosed only to authorized parties.
- **Integrity** is the characteristic that alterations to a system's assets can be made only in an authorized way.
- In other words, improper alterations in a secure computer system should be detectable and recoverable.
- Major assets of any computer system are its hardware, software, and data.

- Another way of looking at security in computer systems is that we attempt to protect the services and data it offers against security threats.
- There are four types of security threats to consider (Pfleeger, 2003):
 1. Interception
 2. Interruption
 3. Modification
 4. Fabrication

- The concept of **interception** refers to the situation that an unauthorized party has gained access to a service or data.
- A typical example of interception is where communication between two parties has been overheard by someone else.
- Interception also happens when data are illegally copied, for example, after breaking into a person's private directory in a file system.
- An example of interruption is when a file is corrupted or lost.
- More generally **interruption** refers to the situation in which services or data become unavailable, unusable, destroyed, and so on.
- In this sense, denial of service attacks by which someone maliciously attempts to make a service inaccessible to other parties is a security threat that classifies as interruption.

- **Modifications** involve unauthorized changing of data or tampering with a service so that it no longer adheres to its original specifications.
- Examples of modifications include intercepting and subsequently changing transmitted data, tampering with database entries, and changing a program so that it secretly logs the activities of its user.

Fabrication refers to the situation in which additional data or activity are generated that would normally not exist.

- For example, an intruder may attempt to add an entry into a password file or database.
- Likewise, it is sometimes possible to break into a system by replaying previously sent messages

Important security mechanisms are:

1. Encryption
2. Authentication
3. Authorization
4. Auditing

- **Encryption** is fundamental to computer security.
- Encryption transforms data into something an attacker cannot understand.
- In other words, encryption provides a means to implement data confidentiality.
- In addition, encryption allows us to check whether data have been modified.
- It thus also provides support for integrity checks.
- Authentication is used to verify the claimed identity of a user, client, server, host, or other entity.
- In the case of clients, the basic premise is that before a service starts to perform any work on behalf of a client, the service must learn the client's identity (unless the service is available to all).
- Typically, users are authenticated by means of passwords, but there are many other ways to authenticate clients.

- After a client has been authenticated, it is necessary to check whether that client is authorized to perform the action requested.
- Access to records in a medical database is a typical example.
- Depending on who accesses the database. Permission may be granted to read records, to modify certain fields in a record, or to add or remove a record.
- Auditing tools are used to trace which clients accessed what, and which way.
- Although auditing does not really provide any protection against security threats, audit logs can be extremely useful for the analysis of a security breach, and subsequently taking measures against intruders.
- For this reason, attackers are generally keen not to leave any traces that could eventually lead to exposing their identity.
- In this sense, logging accesses makes attacking sometimes a riskier business.

9.2 SECURE CHANNELS

- In the preceding chapters. we have frequently used the client-server model as a convenient way to organize a distributed system.
- In this model, servers may possibly be distributed and replicated, but also act as clients with respect to other servers.
- When considering security in distributed systems, it is once again useful to think in terms of clients and servers.
- In particular, making a distributed system secure essentially boils down to two predominant issues.
- The first issue is how to make the communication between clients and servers secure. Secure communication requires authentication of the communicating parties. In many cases it also requires ensuring message integrity and possibly confidentiality as well.
- As part of this problem, we also need to consider protecting the communication within a group of servers.

- The second issue is that of authorization: once a server has accepted a request from a client, how can it find out whether that client is authorized to have that request carried out?
- Authorization is related to the problem of controlling access to resources, which we discuss extensively in the next section.
- In this section, we concentrate on protecting the communication within a distributed system.
- The issue of protecting communication between clients and servers, can be thought of in terms of setting up a secure **channel** between communicating parties (Voydock and Kent, 1983).
- A secure channel protects senders and receivers against interception, modification, and fabrication of messages.
- It does not also necessarily protect against interruption.

- Protecting messages against interception is done by ensuring confidentiality: the secure channel ensures that its messages cannot be eavesdropped by intruders.
- Protecting against modification and fabrication by intruders is done through protocols for mutual authentication and message integrity.
- In the following pages, we first discuss various protocols that can be used for authentication, using symmetric as well as public-key cryptosystems.
- A detailed description of the logics underlying authentication can be found in Lampson et al. (1992). We discuss confidentiality and message integrity separately.

Cryptography

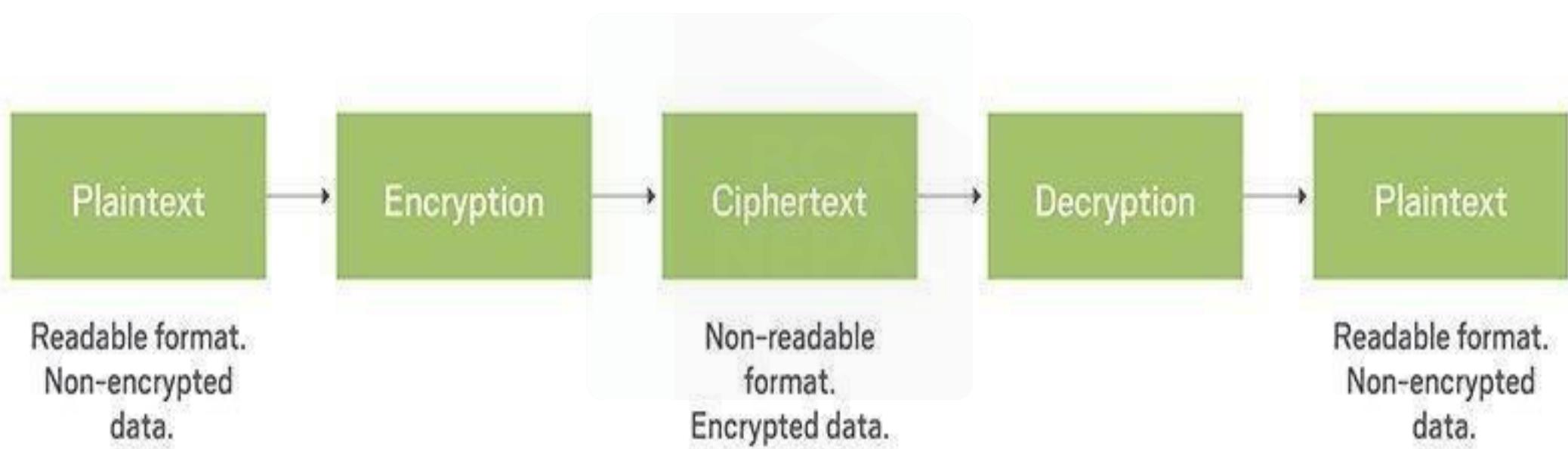
- Cryptography is a method of protecting information and communications through the use of codes, so that only those for whom the information is intended can read and process it. The prefix "crypt-" means "hidden" or "vault" -- and the suffix "-graphy" stands for "writing."
- In computer science, cryptography refers to secure information and communication techniques derived from mathematical concepts and a set of rule-based calculations called algorithms, to transform messages in ways that are hard to decipher.
- These deterministic algorithms are used for cryptographic key generation, digital signing, verification to protect data privacy, web browsing on the internet, and confidential communications such as credit card transactions and email.

- Cryptography is closely related to the disciplines of cryptology and cryptanalysis.
- It includes techniques such as microdots, merging words with images, and other ways to hide information in storage or transit.
- However, in today's computer-centric world, cryptography is most often associated with scrambling plaintext(ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), then back again (known as decryption).
- Individuals who practice this field are known as cryptographers.

- Modern cryptography concerns itself with the following four objectives:
 - 1. Confidentiality:** the information cannot be understood by anyone for whom it was unintended
 - 2. Integrity:** the information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected
 - 3. Non-repudiation:** the creator/sender of the information cannot deny at a later stage his or her intentions in the creation or transmission of the information
 - 4. Authentication:** the sender and receiver can confirm each other's identity and the origin/destination of the information

- While **cryptography** is the science of securing data, **cryptanalysis** is the science of analyzing and breaking secure communication.
- Classical **cryptanalysis** involves an interesting combination of analytical reasoning, application of mathematical tools, pattern finding, patience, determination, and luck.
- **Cryptanalysts** are also called **attackers**.
- **Cryptology** embraces both **cryptography** and **cryptanalysis**.(<https://youtu.be/5jpgMXt1Z9Y>)

Cryptography

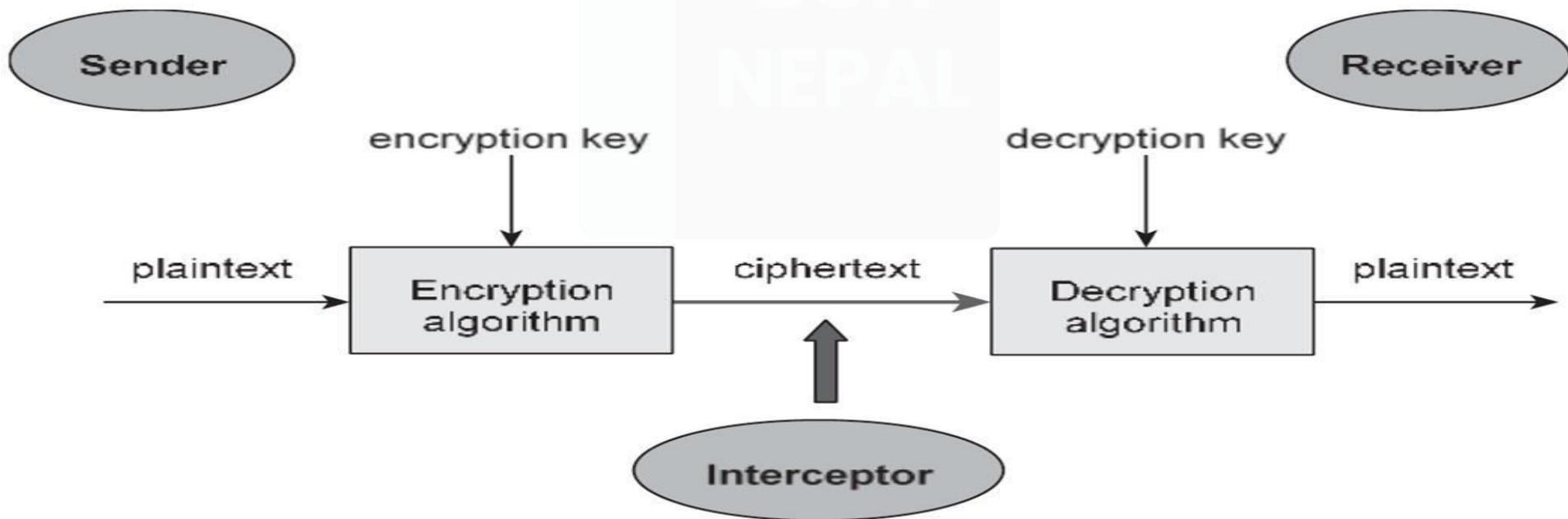


Components of a Cryptosystem

The various components of a basic cryptosystem are as follows:

- **Plaintext.** It is the data to be protected during transmission.
- **Encryption Algorithm.** It is a mathematical process that produces a ciphertext for any given plaintext and encryption key. It is a cryptographic algorithm that takes plaintext and an encryption key as input and produces a ciphertext.
- **Ciphertext.** It is the scrambled version of the plaintext produced by the encryption algorithm using a specific the encryption key. The ciphertext is not guarded. It flows on public channel. It can be intercepted or compromised by anyone who has access to the communication channel.
- **Decryption Algorithm,** It is a mathematical process, that produces a unique plaintext for any given ciphertext and decryption key. It is a cryptographic algorithm that takes a ciphertext and a decryption key as input, and outputs a plaintext. The decryption algorithm essentially reverses the encryption algorithm and is thus closely related to it.
- **Encryption Key.** It is a value that is known to the sender. The sender inputs the encryption key into the encryption algorithm along with the plaintext in order to compute the ciphertext.
- **Decryption Key.** It is a value that is known to the receiver. The decryption key is related to the encryption key, but is not always identical to it. The receiver inputs the decryption key into the decryption algorithm along with the ciphertext in order to compute the plaintext.

- An **interceptor** (an **attacker**) is an unauthorized entity who attempts to determine the plaintext.
- S/He can see the ciphertext and may know the decryption algorithm.
- S/He, however, must never know the decryption key.

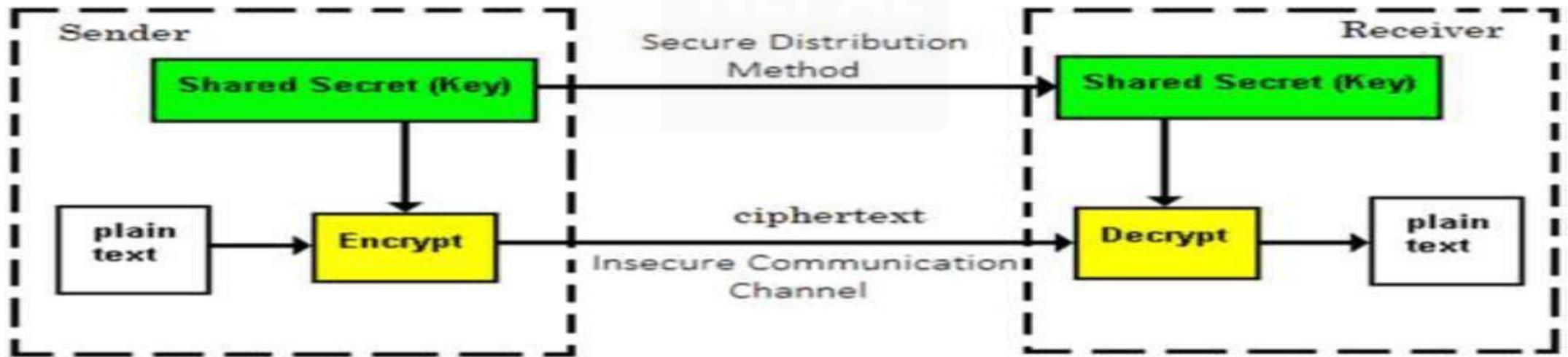


Types of Cryptosystems

- Fundamentally, there are two types of cryptosystems based on the manner in which encryption-decryption is carried out in the system:
 - Symmetric Key Encryption
 - Asymmetric Key Encryption
- The main difference between these cryptosystems is the relationship between the encryption and the decryption key. Logically, in any cryptosystem, both the keys are closely associated. It is practically impossible to decrypt the ciphertext with the key that is unrelated to the encryption key.

Symmetric Key Encryption / Secret key Cryptography

- The encryption process where **same keys are used for encrypting and decrypting** the information is known as Symmetric Key Encryption.
- The study of symmetric cryptosystems is referred to as **symmetric cryptography**. Symmetric cryptosystems are also sometimes referred to as **secret key cryptosystems**.
- A few well-known examples of symmetric key encryption methods are: Digital Encryption Standard (DES), Triple-DES (3DES), IDEA, and BLOWFISH



The salient features of cryptosystem based on symmetric key encryption are:

- Persons using symmetric key encryption must share a common key prior to exchange of information.
- Keys are recommended to be changed regularly to prevent any attack on the system.
- A robust mechanism needs to exist to exchange the key between the communicating parties. As keys are required to be changed regularly, this mechanism becomes expensive and cumbersome.
- In a group of n people, to enable two-party communication between any two persons, the number of keys required for group is $n \times (n - 1)/2$.
- Length of Key (number of bits) in this encryption is smaller and hence, process of encryption-decryption is faster than asymmetric key encryption.
- Processing power of computer system required to run symmetric algorithm is less.

Challenge of Symmetric Key Cryptosystem

- There are two restrictive challenges of employing symmetric key cryptography.
 - **Key establishment** – Before any communication, both the sender and the receiver need to agree on a secret symmetric key. It requires a secure key establishment mechanism in place.
 - **Trust Issue** – Since the sender and the receiver use the same symmetric key, there is an implicit requirement that the sender and the receiver ‘trust’ each other. For example, it may happen that the receiver has lost the key to an attacker and the sender is not informed.
- These two challenges are highly restraining for modern day communication. Today, people need to exchange information with non-familiar and non-trusted parties. For example, a communication between online seller and customer. These limitations of symmetric key encryption gave rise to asymmetric key encryption schemes

Asymmetric Key Encryption / Public key Cryptography

- The encryption process where **different keys are used for encrypting and decrypting the information** is known as Asymmetric Key Encryption.
- Though the keys are different, they are mathematically related and hence, retrieving the plaintext by decrypting cipher text is feasible.
- Asymmetric Key Encryption was invented in the 20th century to come over the necessity of pre-shared secret key between communicating persons.

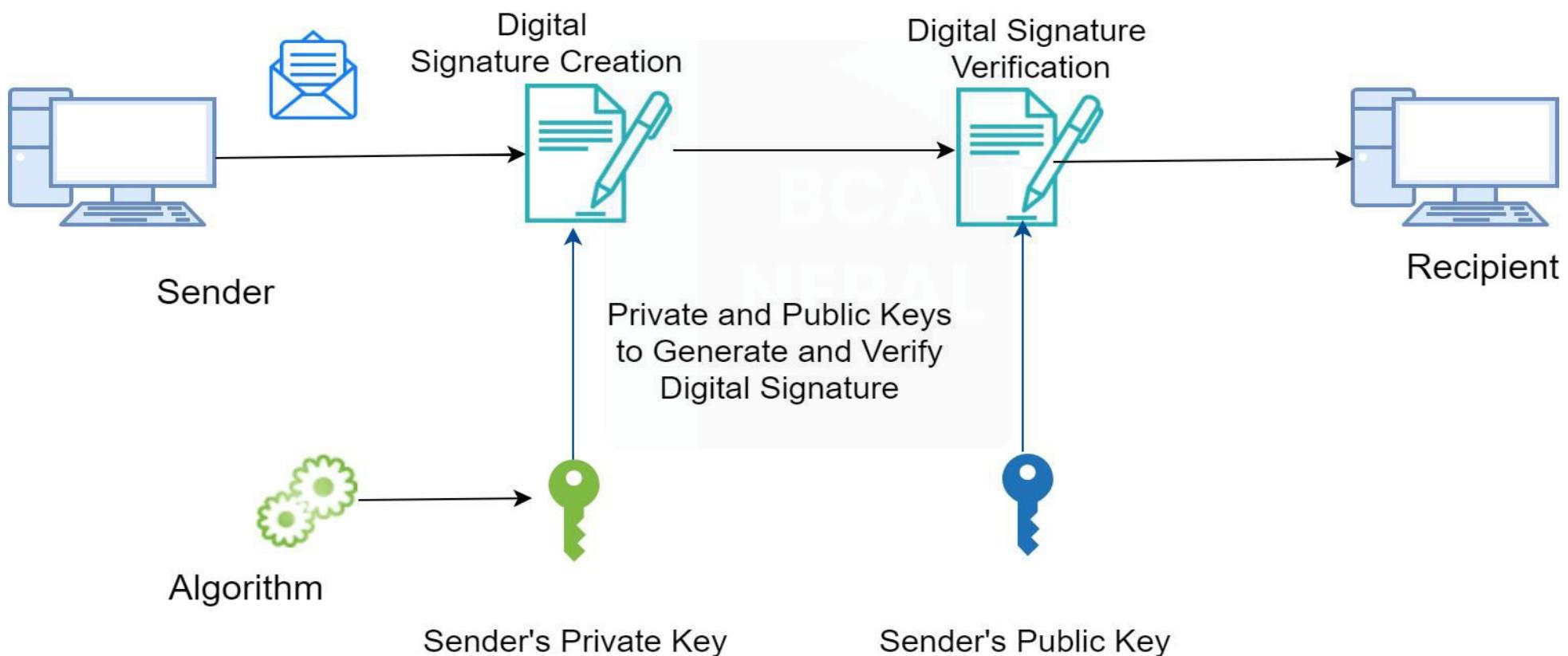
The salient features of this encryption scheme are as follows:

- Every user in this system needs to have a pair of dissimilar keys, **private key** and **public key**. These keys are mathematically related – when one key is used for encryption, the other can decrypt the ciphertext back to the original plaintext.
- It requires to put the public key in public repository and the private key as a well guarded secret. Hence, this scheme of encryption is also called **Public Key Encryption**.
- Though public and private keys of the user are related, it is computationally not feasible to find one from another. This is a strength of this scheme.
- When *Host1* needs to send data to *Host2*, he obtains the public key of *Host2* from repository, encrypts the data, and transmits.
- *Host2* uses his private key to extract the plaintext.
- Length of Keys (number of bits) in this encryption is large and hence, the process of encryption-decryption is slower than symmetric key encryption.
- Processing power of computer system required to run asymmetric algorithm is higher.

Digital Signature

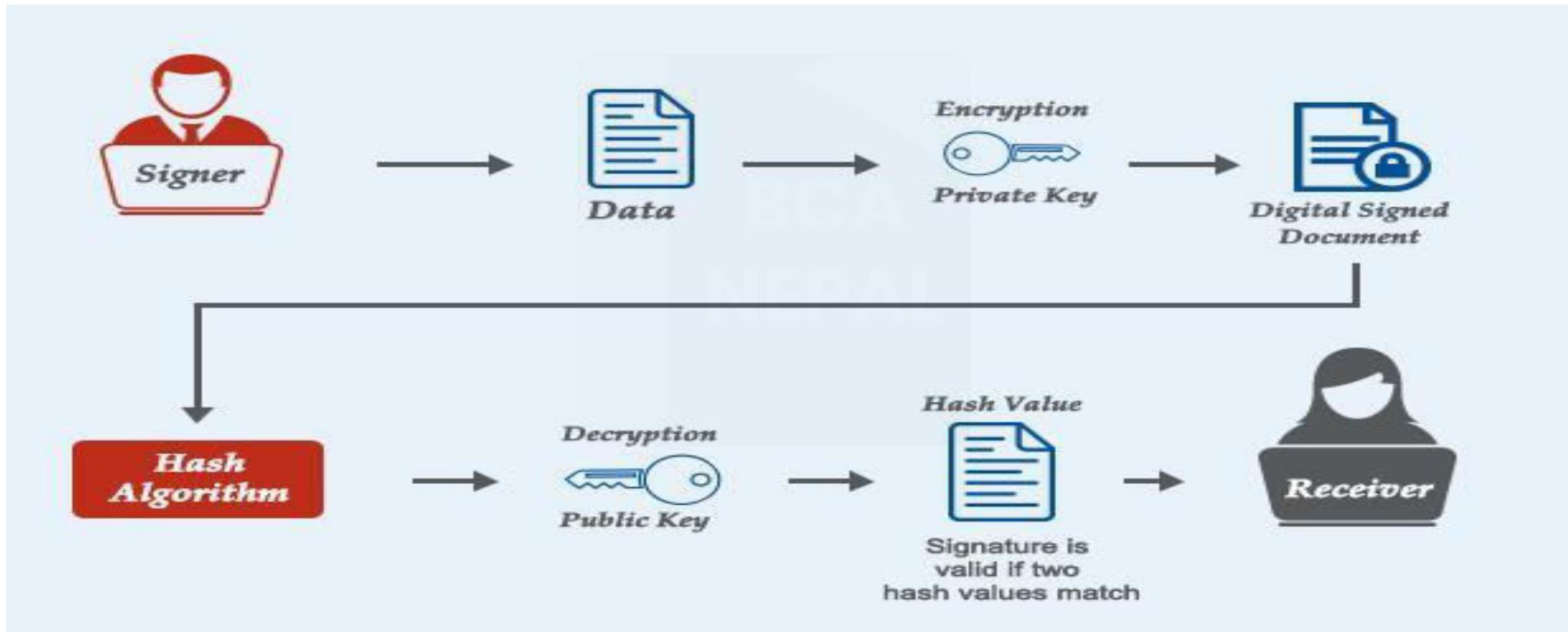
- A digital signature—a type of electronic signature—is a mathematical algorithm routinely used to validate the authenticity and integrity of a message (e.g., an email, a credit card transaction, or a digital document).
- Digital signatures create a virtual fingerprint that is unique to a person or entity and are used to identify users and protect information in digital messages or documents.
- In emails, the email content itself becomes part of the digital signature.
- Digital signatures are significantly more secure than other forms of electronic signatures.
- Digital signatures increase the transparency of online interactions and develop trust between customers, business partners, and vendors. (<https://youtu.be/TmA2QWSLSPg>)

Digital Signatures



How do digital signatures work?

- **Hash function** – A hash function (also called a “hash”) is a fixed-length string of numbers and letters generated from a mathematical algorithm and an arbitrarily sized file such as an email, document, picture, or other type of data.
- This generated string is unique to the file being hashed and is a one-way function— a computed hash cannot be reversed to find other files that may generate the same hash value.
- Some of the more popular hashing algorithms in use today are Secure Hash Algorithm-1 (SHA-1), the Secure Hashing Algorithm-2 family (SHA-2 and SHA-256), and Message Digest 5 (MD5).



Public key cryptography

- Public key cryptography (also known as asymmetric encryption) is a cryptographic method that uses a key pair system.
- One key, called the public key, encrypts the data. The other key, called the private key, decrypts the data.
- Public key cryptography can be used several ways to ensure confidentiality, integrity, and authenticity.
- Public key cryptography can ensure integrity by creating a digital signature of the message using the sender's private key.
- This is done by hashing the message and encrypting the hash value with their private key. By doing this, any changes to the message will result in a different hash value.
- Ensure confidentiality by encrypting the entire message with the recipient's public key. This means that only the recipient, who is in possession of the corresponding private key, can read the message.
- Verify the user's identity using the public key and checking it against a certificate authority.

Digital certificates

- One issue with public key cryptosystems is that users must be constantly vigilant to ensure that they are encrypting to the correct person's key.
- In an environment where it is safe to freely exchange keys via public servers, *man-in-the-middle* attacks are a potential threat.
- In this type of attack, someone posts a phony key with the name and user ID of the user's intended recipient. Data encrypted to, and intercepted by, the true owner of this bogus key is now in the wrong hands.
- In a public key environment, it is vital that you are assured that the public key to which you are encrypting data is in fact the public key of the intended recipient and not a forgery.
- You could simply encrypt only to those keys which have been physically handed to you. But suppose you need to exchange information with people you have never met; how can you tell that you have the correct key?
- **Digital certificates** simplify the task of establishing whether a public key truly belongs to the purported owner.

- A digital certificate is data that functions much like a physical certificate.
- A digital certificate is information included with a person's public key that helps others verify that a key is genuine or *valid*.

A digital certificate consists of three things:

- A public key.
- Certificate information ("Identity" information about the user, such as name, user ID, and so on.)
- One or more digital signatures.

The purpose of the digital signature on a certificate is to state that the certificate information has been attested to by some other person or entity. The digital signature does not attest to the authenticity of the certificate as a whole; it vouches only that the signed identity information goes along with, or *is bound to*, the public key.

Thus, a certificate is basically a public key with one or two forms of ID attached, plus a hearty stamp of approval from some other trusted individual.

([https://youtu.be/UbMlPIgzT_BBXA\(cE-commerce\)](https://youtu.be/UbMlPIgzT_BBXA(cE-commerce)))

9.3 ACCESS CONTROL

- In the client-server model, which we have used so far, once a client and a server have set up a secure channel, the client can issue requests that are to be carried out by the server.
- Requests involve carrying out operations on resources that are controlled by the server. A general situation is that of an object server that has a number of objects under its control.
- A request from a client generally involves invoking a method of a specific object.
- Such a request can be carried out only if the client has sufficient access rights for that invocation.

- Formally, verifying access rights is referred to as access control, whereas authorization is about granting access rights.
- The two terms are strongly related to each other and are often used in an interchangeable way. There are many ways to achieve access control.
- We start with discussing some of the general issues, concentrating on different models for handling access control.
- One important way of actually controlling access to resources is to build a firewall that protects applications or even an entire network.
- Firewalls are discussed separately.
- With the advent of code mobility, access control could no longer be done using only the traditional methods.

9.3.1 General Issues in Access Control

- In order to understand the various issues involved in access control, the simple model shown in Fig. 9-25 is generally adopted.
- It consists of subjects that issue a request to access an object. An object is very much like the objects we have been discussing so far.
- It can be thought of as encapsulating its own state and implementing the operations on that state.
- The operations of an object that subjects can request to be carried out are made available through interfaces. Subjects can best be thought of as being processes acting on behalf of users, but can also be objects that need the services of other objects in order to carry out their work

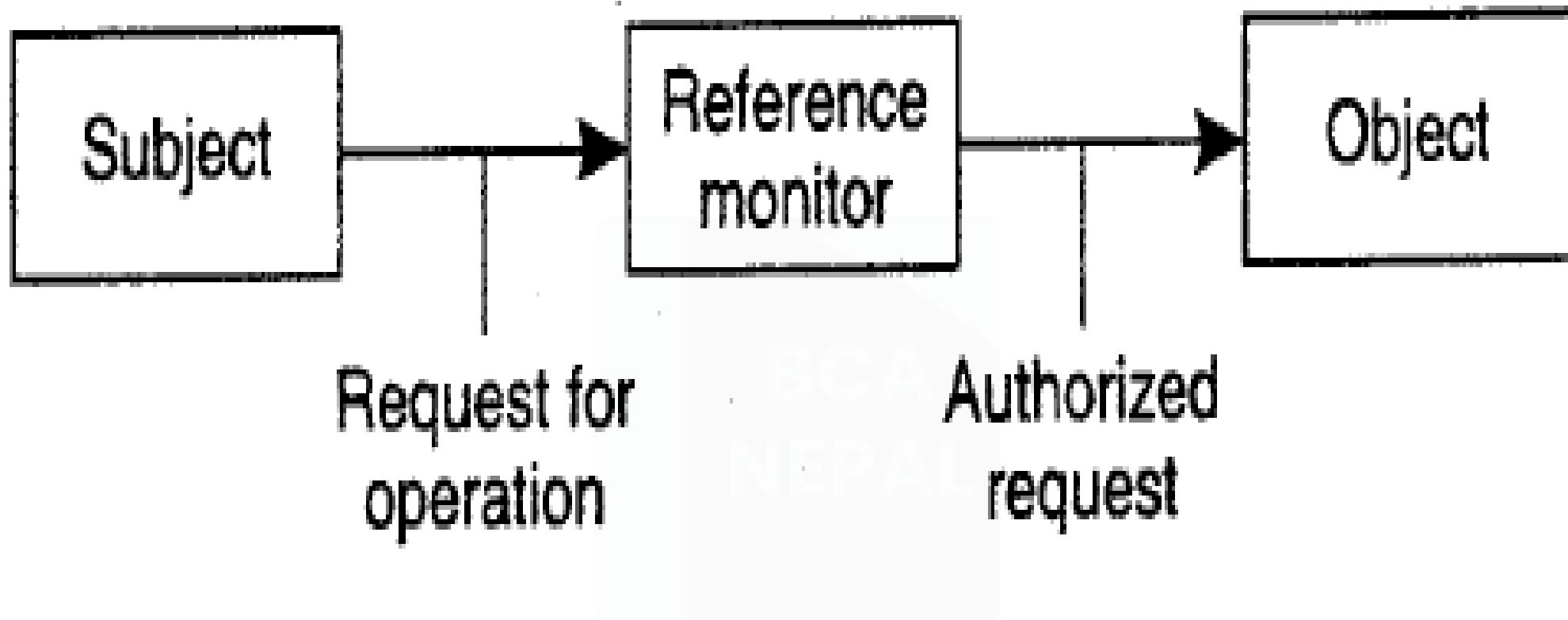


Figure 9-25. General model of controlling access to objects.

- Controlling the access to an object is all about protecting the object against invocations by subjects that are not allowed to have specific (or even any) of the methods carried out.
- Also, protection may include object management issues, such as creating, renaming, or deleting objects.
- Protection is often enforced by a program called a reference monitor.
- A reference monitor records which subject may do what, and decides whether a subject is allowed to have a specific operation carried out.
- This monitor is called (e.g., by the underlying trusted operating system) each time an object is invoked.
- Consequently, it is extremely important that the reference monitor is itself tamperproof: an attacker must not be able to fool around with it.

Access Control Matrix

- A common approach to modeling the access rights of subjects with respect to objects is to construct an access control matrix.
- Each subject is represented by a row in this matrix; each object is represented by a column.
- If the matrix is denoted M , then an entry $M[s,o]$ lists precisely which operations subject s can request to be carried out on object o .
- In other words, whenever a subject s requests the invocation of method m of object o , the reference monitor should check whether m is listed in $M[s,o]$. If m is not listed in $M[s,o]$, the invocation fails.

- Considering that a system may easily need to support thousands of users and millions of objects that require protection, implementing an access control matrix as a true matrix is not the way to go.
- Many entries in the matrix will be empty: a single subject will generally have access to relatively few objects.
- Therefore, other, more efficient ways are followed to implement an access control matrix.
- One widely-applied approach is to have each object maintain a list of the access rights of subjects that want to access the object.
- In essence, this means that the matrix is distributed column-wise across all objects, and that empty entries are left out.
- This type of implementation leads to what is called an Access Control List(ACL). Each object is assumed to have its own associated ACL.

- Another approach is to distribute the matrix row-wise by giving each subject a list of capabilities it has for each object. In other words, a capability corresponds to an entry in the access control matrix.
- Not having a capability for a specific object means that the subject has no access rights for that object.
- A capability can be compared to a ticket: its holder is given certain rights that are associated with that ticket. It is also clear that a ticket should be protected against modifications by its holder.
- One approach that is particularly suited in distributed systems and which has been applied extensively in Amoeba (Tanenbaum et al., 1990), is to protect (a list of) capabilities with a signature.
- We return to these and other matters later when discussing security management.

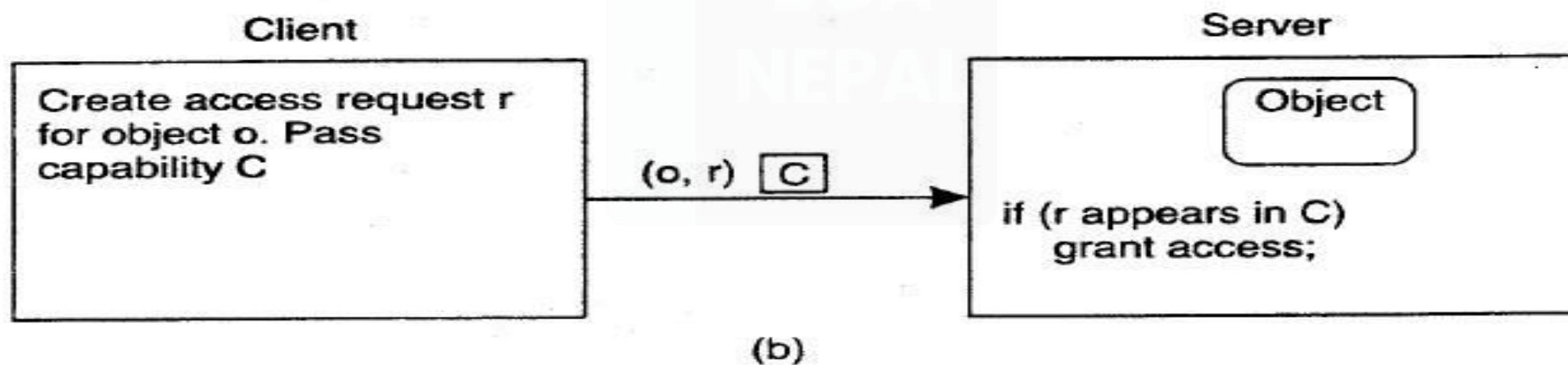
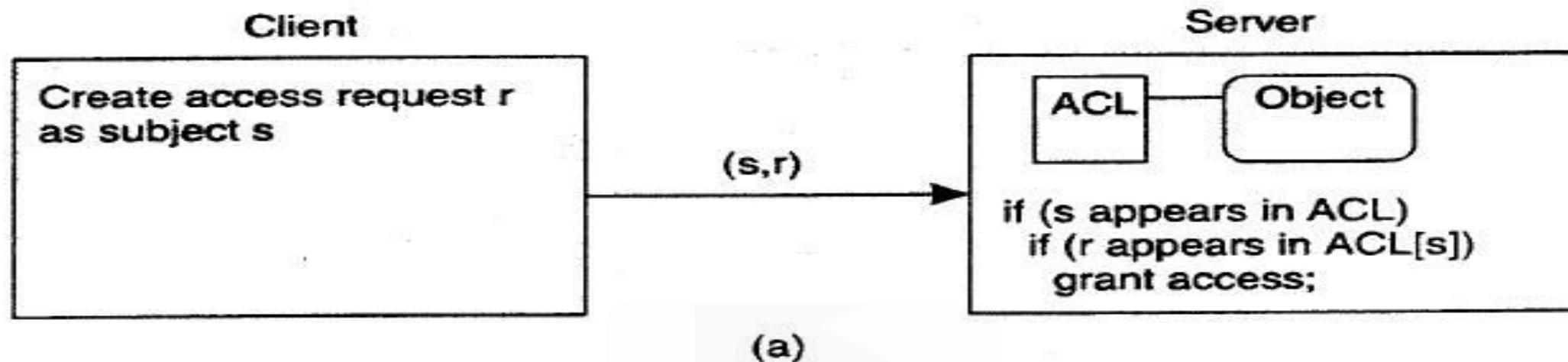


Figure 9-26. Comparison between ACLs and capabilities for protecting objects.
 (a) Using an ACL. (b) Using capabilities.

- The difference between how ACLs and capabilities are used to protect the access to an object is shown in Fig. 9-26.
- Using ACLs, when a client sends a request to a server, the server's reference monitor will check whether it knows the client and if that client is known and allowed to have the requested operation carried out, as shown in Fig. 9-26(a).
- However, when using capabilities, a client simply sends its request to the server. The server is not interested in whether it knows the client; the capability says enough.
- Therefore, the server need only check whether the capability is valid and whether the requested operation is listed in the capability. This approach to protecting objects by means of capabilities is shown in Fig. 9-26(b).

9.3.2 Firewalls

- So far, we have shown how protection can be established using cryptographic techniques, combined with some implementation of an access control matrix.
- These approaches work fine as long as all communicating parties play according to the same set of rules.
- Such rules may be enforced when developing a standalone distributed system that is isolated from the rest of the world.
- However, matters become more complicated when outsiders are allowed to access the resources controlled by a distributed system.
- Examples of such accesses including sending mail, downloading files, uploading tax forms, and so on.

- To protect resources under these circumstances, a different approach is needed.
- In practice, what happens is that external access to any part of a distributed system is controlled by a special kind of reference monitor known as a firewall (Cheswick and Bellovin, 2000; and Zwicky et al., 2000).
- Essentially, a firewall - disconnects any part of a distributed system from the outside world, as shown in
- Fig. 9-28. All outgoing, but especially all incoming packets are routed through a special computer and inspected before they are passed.
- Unauthorized traffic is discarded and not allowed to continue.
- An important issue is that the firewall itself should be heavily protected against any kind of security threat: it should never fail.

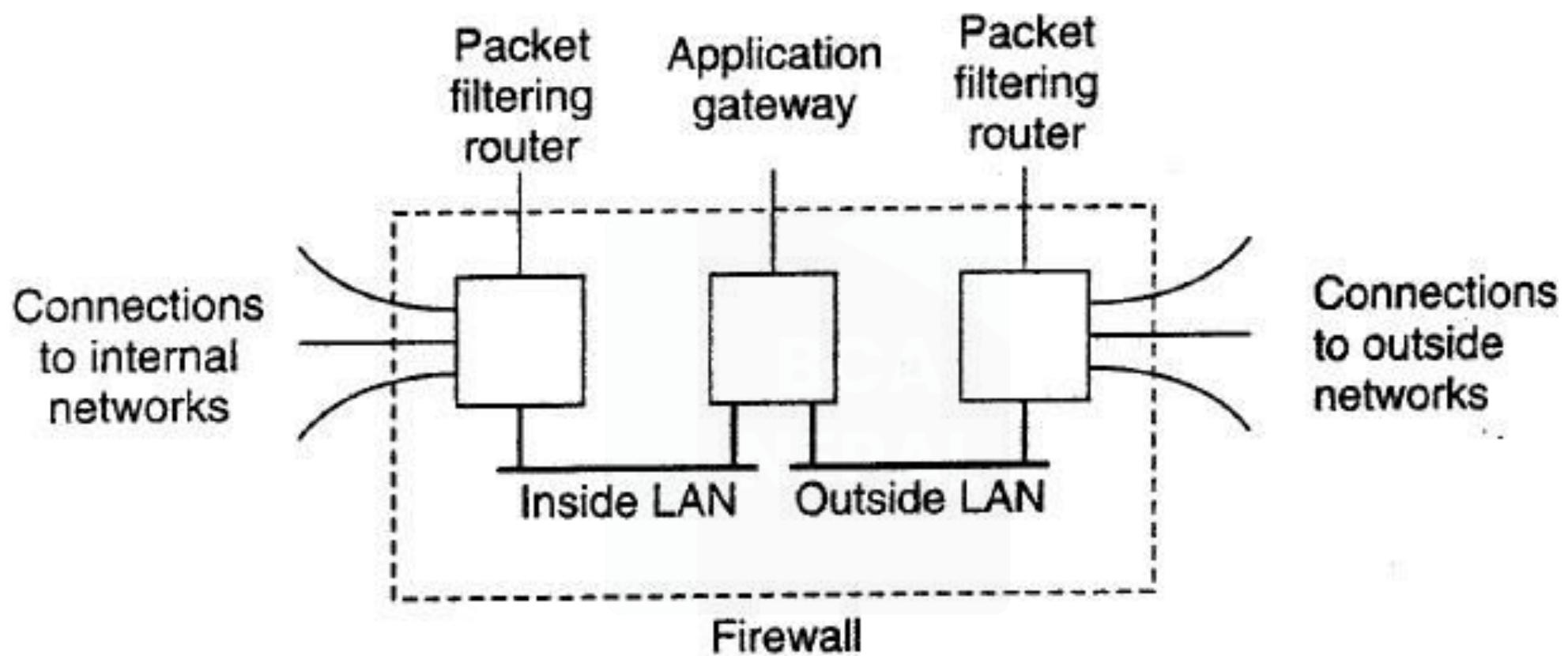


Figure 9-28. A common implementation of a firewall.

- Firewalls essentially come in two different flavors that are often combined.
- An important type of firewall is a **packet-filtering gateway**.
- This type of firewall operates as a router and makes decisions as to whether or not to pass a network packet based on the source and destination address as contained in the packet's header.
- Typically, the packet-filtering gateway shown on the outside LAN in Fig. 9-28 would protect against incoming packets, whereas the one on the inside LAN would filter outgoing packets.
- For example, to protect an internal Web server against requests from hosts that are not on the internal network, a packet-filtering gateway could decide to drop all incoming packets addressed to the Web server.

- The other type of firewall is an **application-level gateway**.
- In contrast to a packet-filtering gateway, which inspects only the header of network packets, this type of firewall actually inspects the content of an incoming or outgoing message.
- A typical example is a mail gateway that discards incoming or outgoing mail exceeding a certain size.
- More sophisticated mail gateways exist that are, for example, capable of filtering spam e-mail.
- Another example of an application-level gateway is one that allows external access to a digital library server, but will supply only abstracts of documents.
- If an external user wants more, an electronic payment protocol is started. Users inside the firewall have direct access to the library service.

- A special kind of application-level gateway is what is known as a **proxy gateway**.
- This type of firewall works as a front end to a specific kind of application, and ensures that only those messages are passed that meet certain criteria. Consider, for example, surfing the Web.
- As we discuss in the next section, many Web pages contain scripts or applets that are to be executed in a user's browser.
- To prevent such code to be downloaded to the inside LAN, all Web traffic could be directed through a Web proxy gateway.
- This gateway accepts regular HTTP requests, either from inside or outside the firewall.
- In other words, it appears to its users as a normal Web server.
- However, it filters all incoming and outgoing traffic, either by discarding certain requests and pages, or modifying pages when they contain executable code.

9.3.3 Secure Mobile Code

- As we discussed in Chap. 3, an important development in modern distributed systems is the ability to migrate code between hosts instead of just migrating passive data.
- However, mobile code introduces a number of serious security threats.
- For one thing, when sending an agent across the Internet, its owner will want to protect it against malicious hosts that try to steal or modify information carried by the agent.
- Another issue is that hosts need to be protected against malicious agents. Most users of distributed systems will not be experts in systems technology and have no way of telling whether the program they are fetching from another host can be trusted not to corrupt their computer.
- In many cases it may be difficult even for an expert to detect that a program is actually being downloaded at all.

- Unless security measures are taken, once a malicious program has settled itself in a computer, it can easily corrupt its host.
- We are faced with an access control problem: the program should not be allowed unauthorized access to the host's resources.
- As we shall see protecting a host against downloaded malicious programs is not always easy.
- The problem is not so much as to avoid downloading of programs.
- Instead, what we are looking for is supporting mobile code that we can allow access to local resources in a flexible, yet fully controlled manner.

Protecting an Agent

- Before we take a look at protecting a computer system against downloaded malicious code, let us first take a look at the opposite situation.
- Consider a mobile agent that is roaming a distributed system on behalf of a user. Such an agent may be searching for the cheapest airplane ticket from Nairobi to Malindi, and has been authorized by its owner to make a reservation as soon as it found a flight.
- For this purpose, the agent may carry an electronic credit card.
- Obviously, we need protection here. Whenever the agent moves to a host, that host should not be allowed to steal the agent's credit card information.

- Also, the agent should be protected against modifications that make the owner pay much more than actually is needed.
- For example, if Chuck's Cheaper Charters can see that the agent has not yet visited its cheaper competitor Alice Airlines, Chuck should be prevented from changing the agent so that it will not visit Alice Airlines' host.
- Other examples that require protection of an agent against attacks from a hostile host include maliciously destroying an agent, or tampering with an agent such that it will attack or steal from its owner when it returns.

Protecting the Target

- Although protecting mobile code against a malicious host is important, more attention has been paid to protecting hosts against malicious mobile code.
- If sending an agent into the outside world is considered too dangerous, a user will generally have alternatives to get the job done for which the agent was intended.
- However, there are often no alternatives to letting an agent into your system, other than locking it out completely.
- Therefore, if it is once decided that the agent can come in, the user needs full control over what the agent can do.

9.3.4 Denial of Service

- Access control is generally about carefully ensuring that resources are accessed only by authorized processes.
- A particularly annoying type of attack that is related to access control is maliciously preventing authorized processes from accessing resources.
- Defenses against such denial-of-service (DoS) attacks are becoming increasingly important as distributed systems are opened up through the Internet.
- Where DoS attacks that come from one or a few sources can often be handled quite effectively, matters become much more difficult when having to deal with distributed denial of service (DDoS).

- In DDoS attacks, a huge collection of processes jointly attempt to bring down a networked service.
- In these cases, we often see that the attackers have succeeded in hijacking a large group of machines which unknowingly participate in the attack.
- Specht and Lee (2004) distinguish two types of attacks: those aimed at bandwidth depletion and those aimed at resource depletion.

9.4 SECURITY MANAGEMENT

- So far, we have considered secure channels and access control, but have hardly touched upon the issue how, for example, keys are obtained.
- In this section, we take a closer look at security management.
- In particular, we distinguish three different subjects. First, we need to consider the general management of cryptographic keys, and especially the means by which public keys are distributed.
- As it turns out, certificates play an important role here.
- Second, we discuss the problem of securely managing a group of servers by concentrating on the problem of adding a new group member that is trusted by the current members.
- Clearly, in the face of distributed and replicated services, it is important that security is not compromised by admitting a malicious process to a group.

- Third, we pay attention to authorization management by looking at capabilities and what are known as attribute certificates.
- An important issue in distributed systems with respect to authorization management is that one process can delegate some or all of its access rights to another process.
- Delegating rights in a secure way has its own subtleties as we also discuss in this section.

9.4.1 Key Management

- So far, we have described various cryptographic protocols in which we (implicitly) assumed that various keys were readily available.
- For example, in the case of public-key cryptosystems, we assumed that a sender of a message had the public key of the receiver at its disposal so that it could encrypt the message to ensure confidentiality.
- Likewise, in the case of authentication using a key distribution center (KDC), we assumed each party already shared a secret key with the KDC.
- However, establishing and distributing keys is not a trivial matter.
- Also, mechanisms are needed to revoke keys, that is, prevent a key from being used after it has been compromised or invalidated.
- For example, revocation is necessary when a key has been compromised.

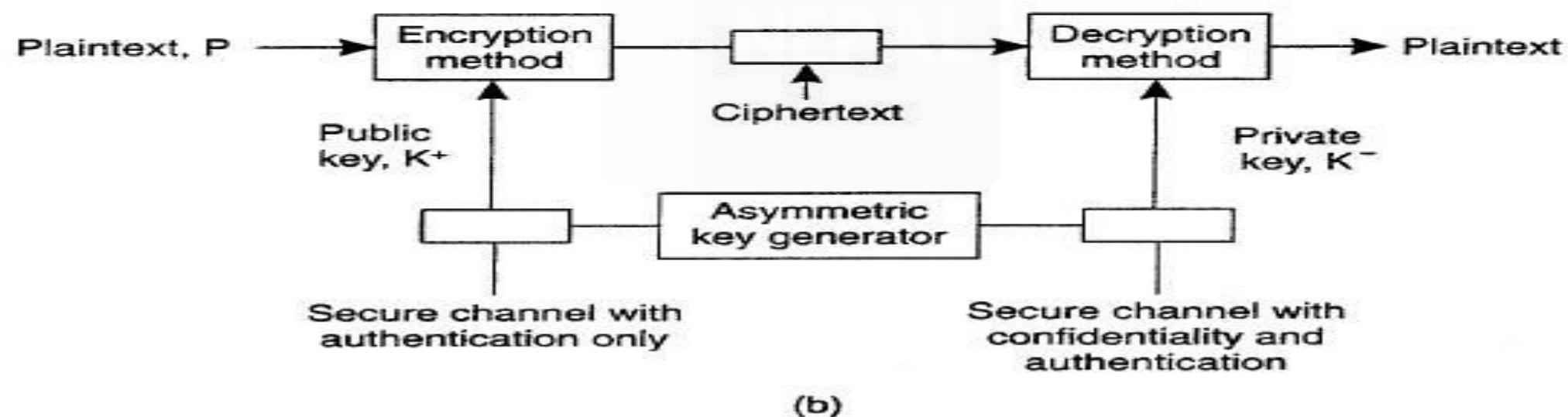
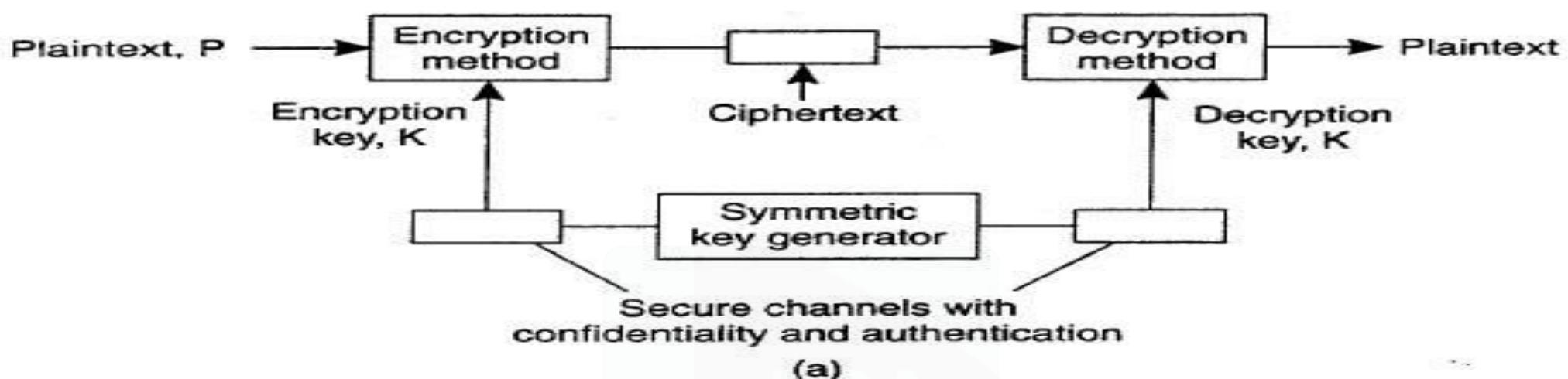


Figure 9-34. (a) Secret-key distribution. (b) Public-key distribution [see also Menezes et al. (1996)].

Lifetime of Certificates

- An important issue concerning certificates is their longevity.
- First let us consider the situation in which a certification authority hands out lifelong certificates.
- Essentially, what the certificate then states is that the public key will always be valid for the entity identified by the certificate.
- Clearly, such a statement is not what we want.
- If the private key of the identified entity is ever compromised, no unsuspecting client should ever be able to use the public key (let alone malicious clients).
- In that case, we need a mechanism to revoke the certificate by making it publicly-known that the certificate is no longer valid,

- There are several ways to revoke a certificate. One common approach is with a Certificate Revocation List (CRL) published regularly by the certification authority.
- Whenever a client checks a certificate, it will have to check the CRL to see whether the certificate has been revoked or not.
- This means that the client will at least have to contact the certification authority each time a new CRL is published.
- Note that if a CRL is published daily, it also takes a day to revoke a certificate.
- Meanwhile, a compromised certificate can be falsely used until it is published on the next CRL.
- Consequently, the time between publishing CRLs cannot be too long.
- In addition, getting a CRL incurs some overhead.

- An alternative approach is to restrict the lifetime of each certificate.
- The validity of a certificate automatically expires after some time.
- If for whatever reason the certificate should be revoked before it expires, the certification authority can still publish it on a CRL.
- However, this approach will still force clients to check the latest CRL whenever verifies a certificate.
- In other words, they will need to contact the certification authority or a trusted database containing the latest CRL.

Secure Group Management

- Many security systems make use of special services such as Key Distribution Centers (KDCs) or Certification Authorities (CAs).
- These services demonstrate a difficult problem in distributed systems.
- In the first place, they must be trusted. To enhance the trust in security services, it is necessary to provide a high degree of protection against all kinds of security threats.
- For example, as soon as a CA has been compromised, it becomes impossible to verify the validity of a public key, making the entire security system completely worthless.

- On the other hand, it is also necessary that many security services offer high availability.
- For example, in the case of a KDC, each time two processes want to set up a secure channel, at least one of them will need to contact the KDC for a shared secret key.
- If the KDC is not available, secure communication cannot be established unless an alternative technique for key establishment is available, such as the Diffie-Hellman key exchange.
- The solution to high availability is replication. On the other hand, replication makes a server more vulnerable to security attacks.
- We already discussed how secure group communication can take place by sharing a secret among the group members.
- In effect, no single group member is capable of compromising certificates, making the group itself highly secure.

Authorization Management

- Managing security in distributed systems is also concerned with managing access rights.
- So far, we have hardly touched upon the issue of how access rights are initially granted to users or groups of users, and how they are subsequently maintained in an unforgeable way.
- It is time to correct this omission.
- In nondistributed systems, managing access rights is relatively easy.
- When a new user is added to the system, that user is given initial rights, for example, to create files and subdirectories in a specific directory. create processes, use CPU time, and so on.
- In other words, a complete account for a user is set up for one specific machine in which all rights have been specified in advance by the system administrators.

Capabilities and Attribute Certificates

- A much better approach that has been widely applied in distributed systems is the use of capabilities.
- As we explained briefly above, a capability is an unforgeable data structure for a specific resource, specifying exactly the access rights that the holder of the capability has with respect to that resource.