

Distributed System (DS)
GTU #3170719



Unit-1

Fundamentals of Distributed System



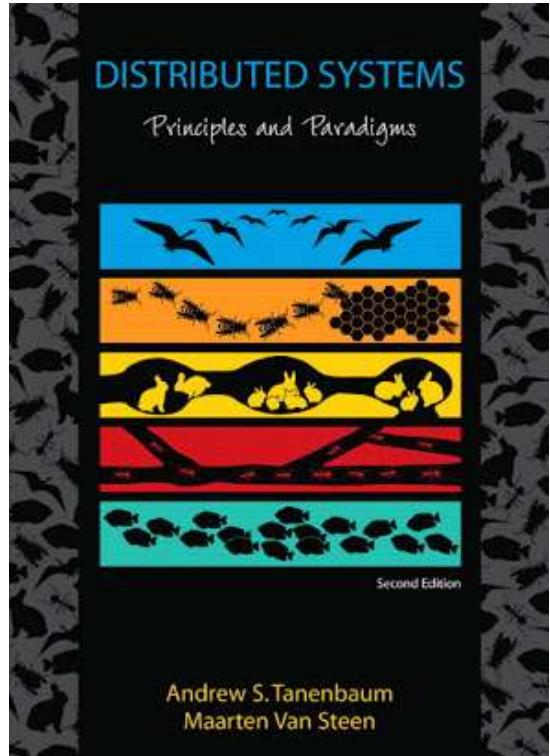
Prof. Umesh H. Thoriya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot

✉ umesh.thoriya@darshan.ac.in
📞 9714233355

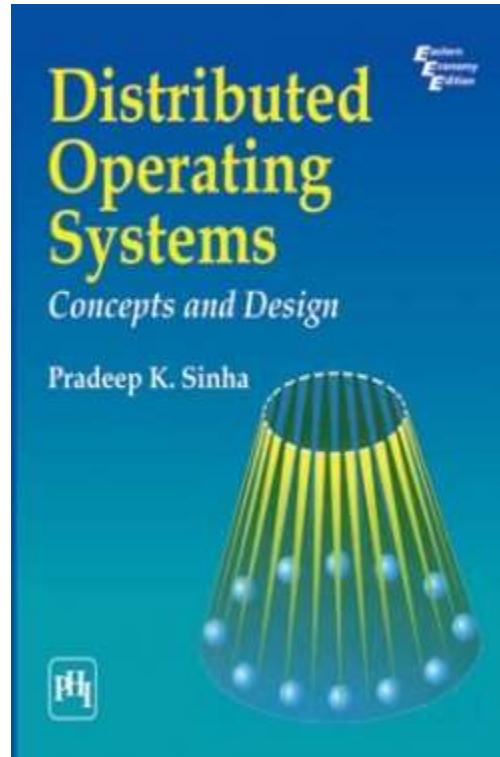
Distributed System-3170719 GTU Syllabus

Unit	Unit Name
1	Fundamentals of Distributed System
2	Basics of Architectures, Processes, and Communication
3	Naming
4	Synchronization
5	Consistency, Replication and Fault Tolerance
6	Security
7	Categories of 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



Topics to be covered

- Definition of a Distributed System
- Goals of a Distributed System
- Types of Distributed Systems
- Basics of Operating System and Networking



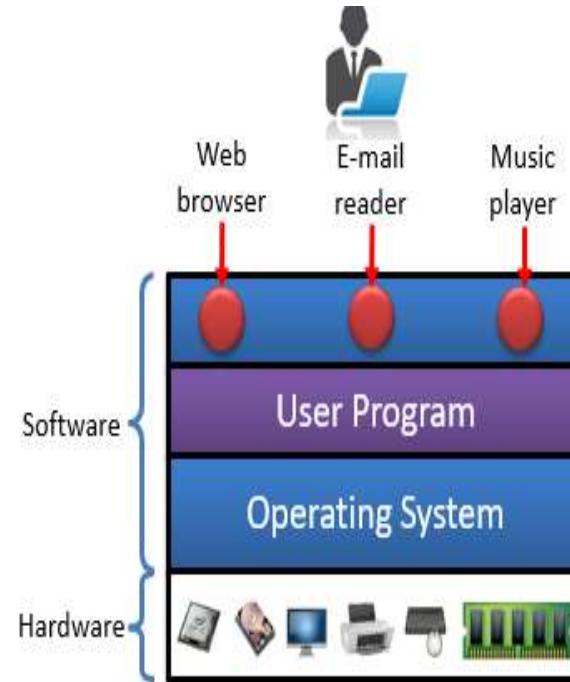
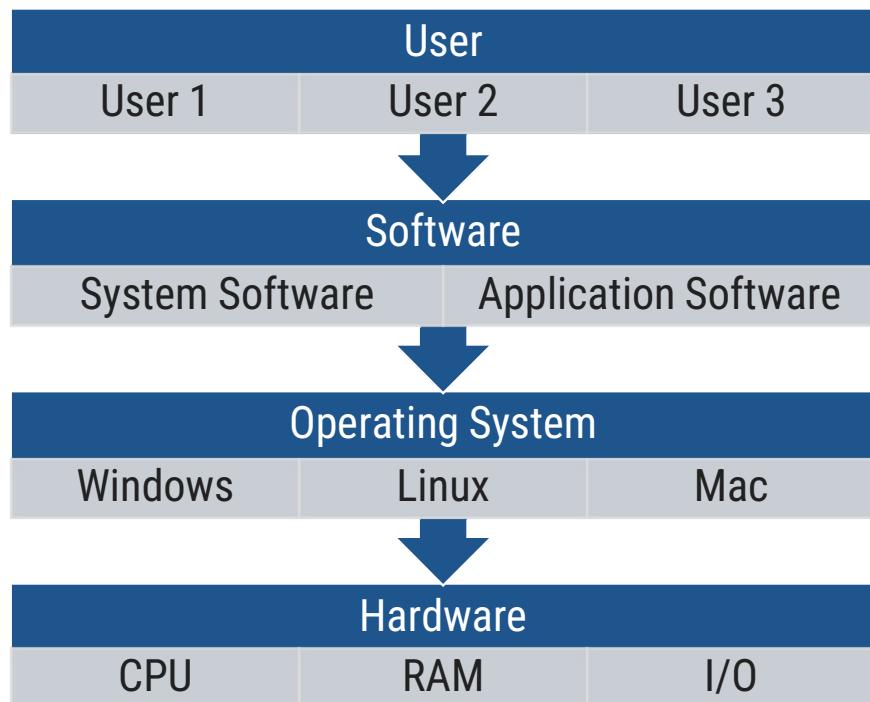
What is Operating System?

- ▶ 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

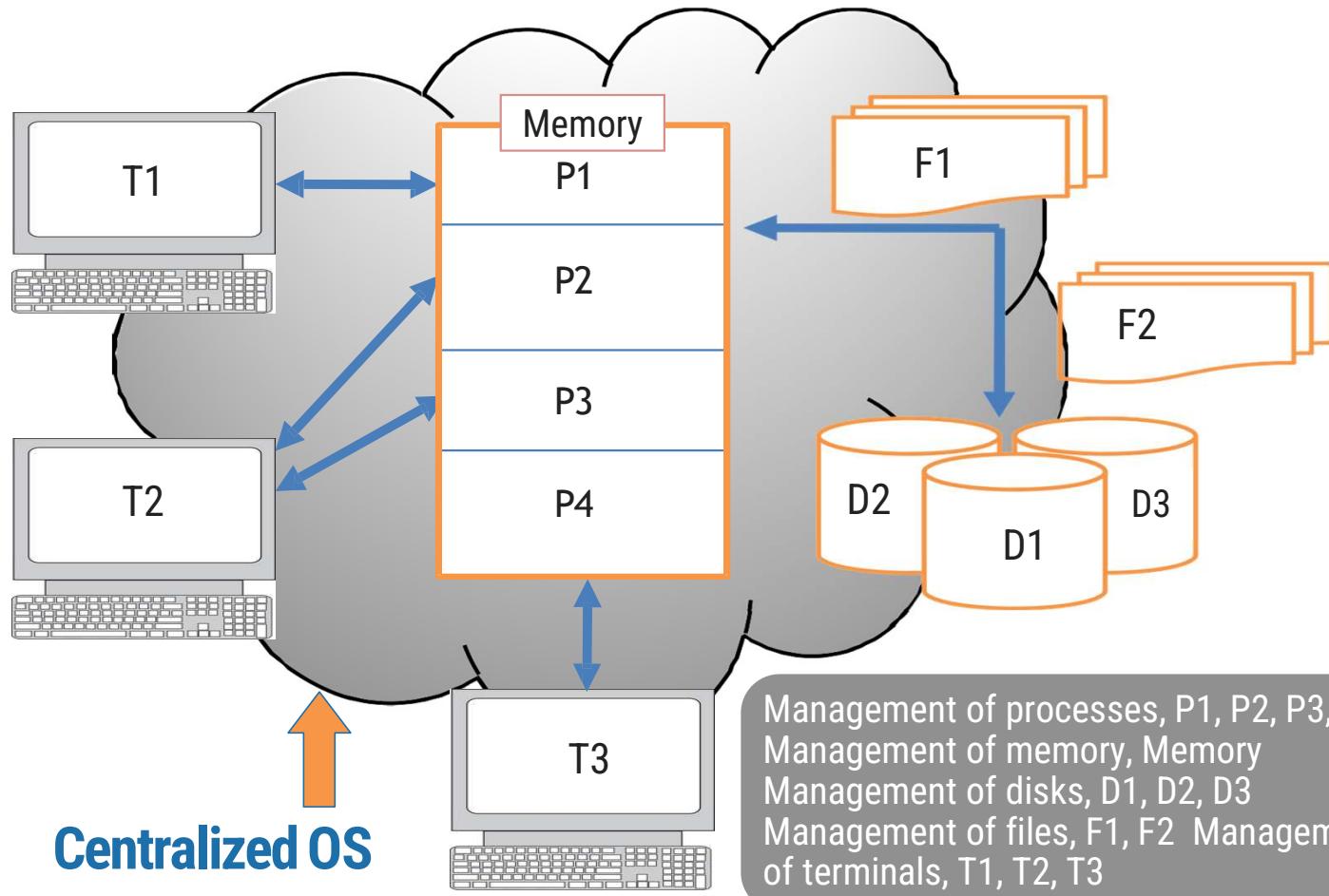
→ Characteristics:

- 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)

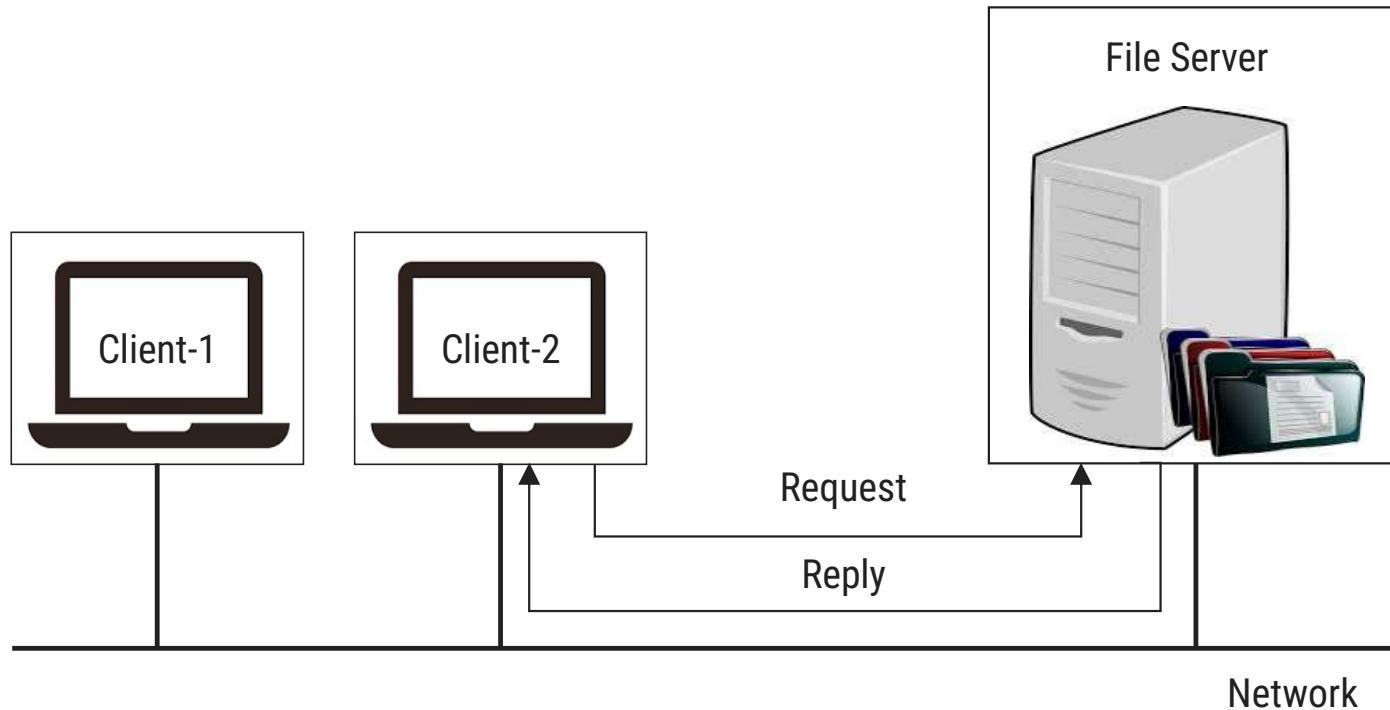
→ Characteristics:

- Remote access
- Information exchange
- Network browsing

→ Goals:

- Interoperability-Sharing of resources between the systems.

Network Operating System

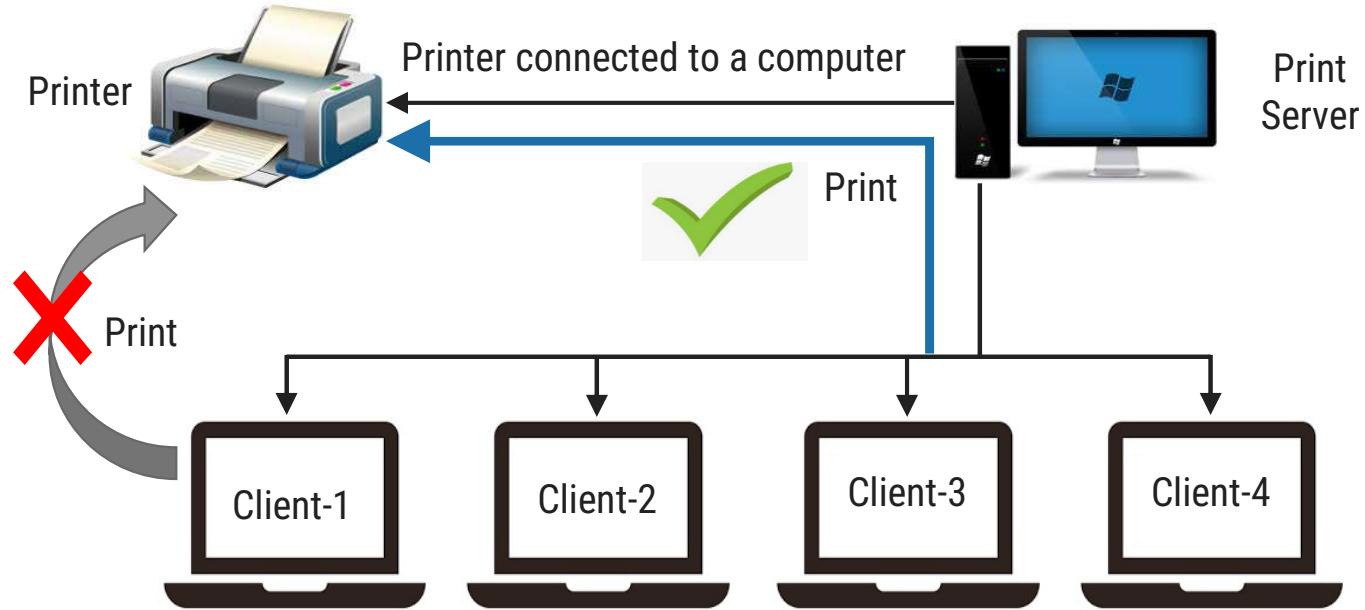


NOS Example



- ▶ 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:

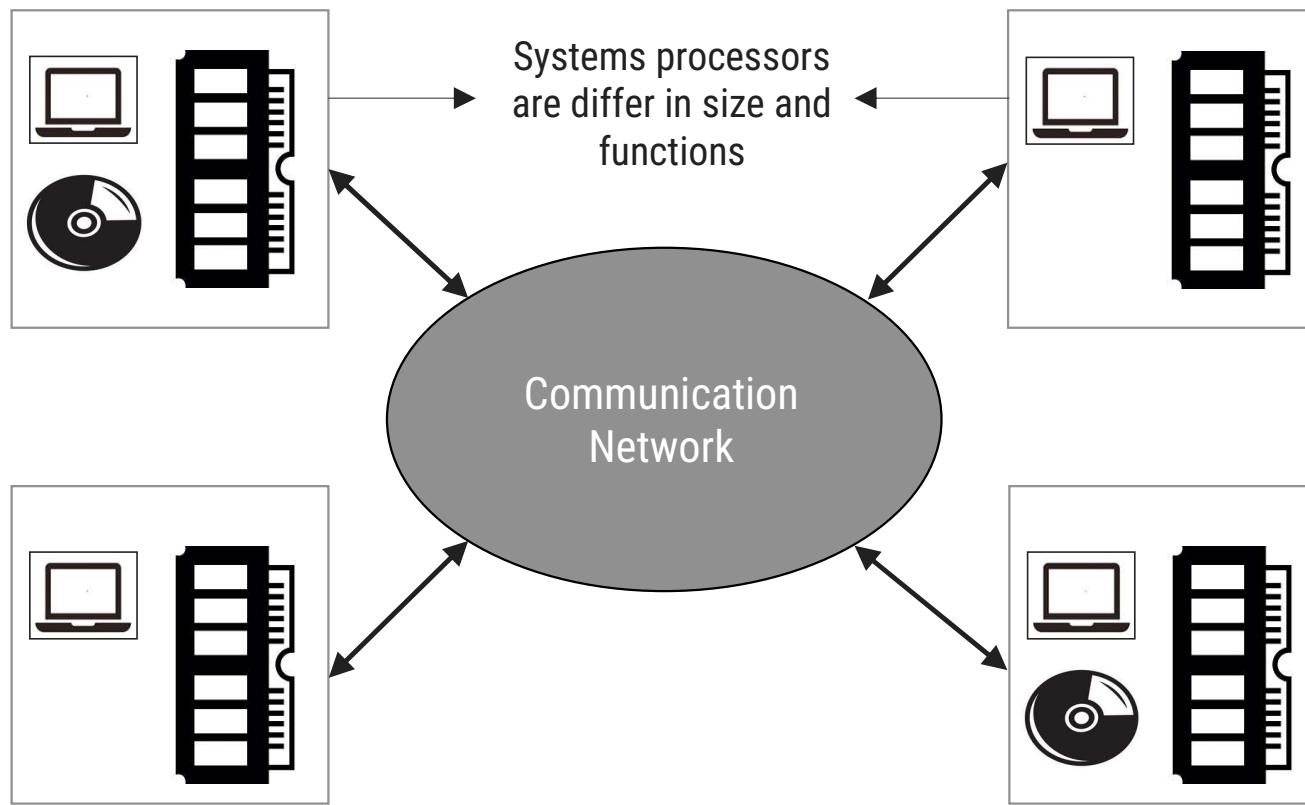
- Global View of Computational power, file system, name space, etc.

→ Goals:

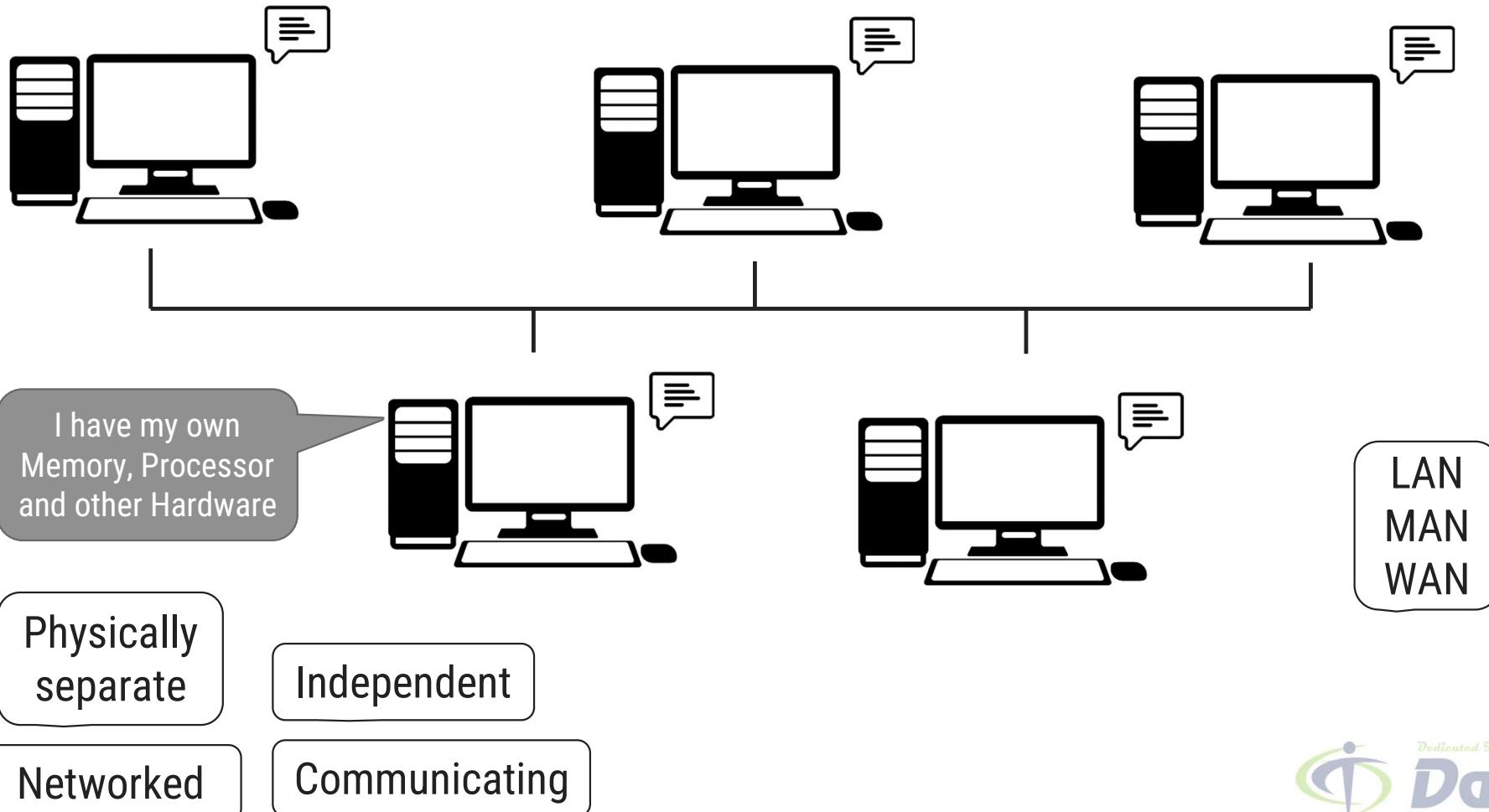
- Single computer view of multiple heterogeneous computer systems.

Distributed Operating System

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



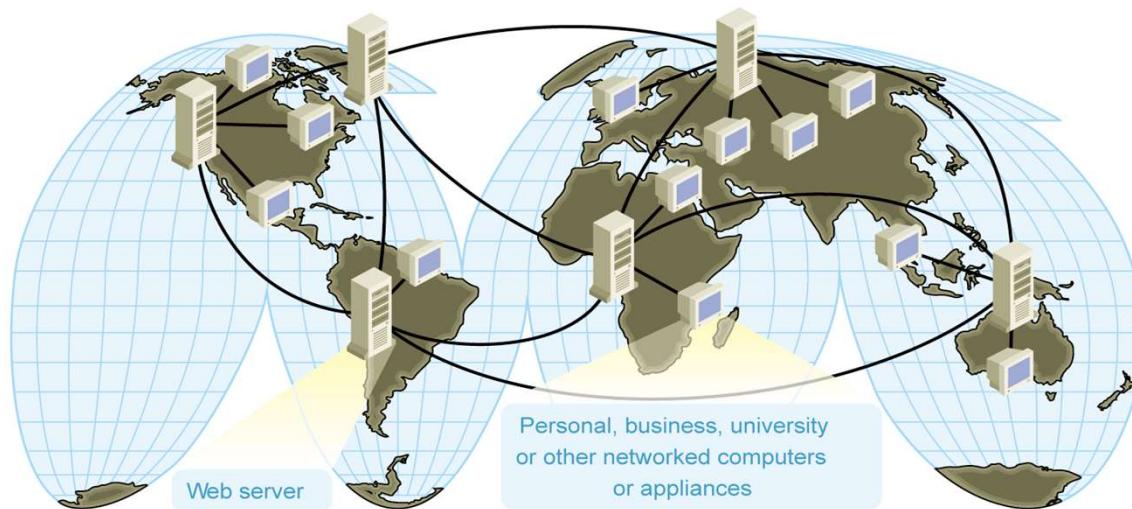
Distributed Operating System



Distributed Operating System

Definition by Coulouris, Dollimore, Kindberg and Blair

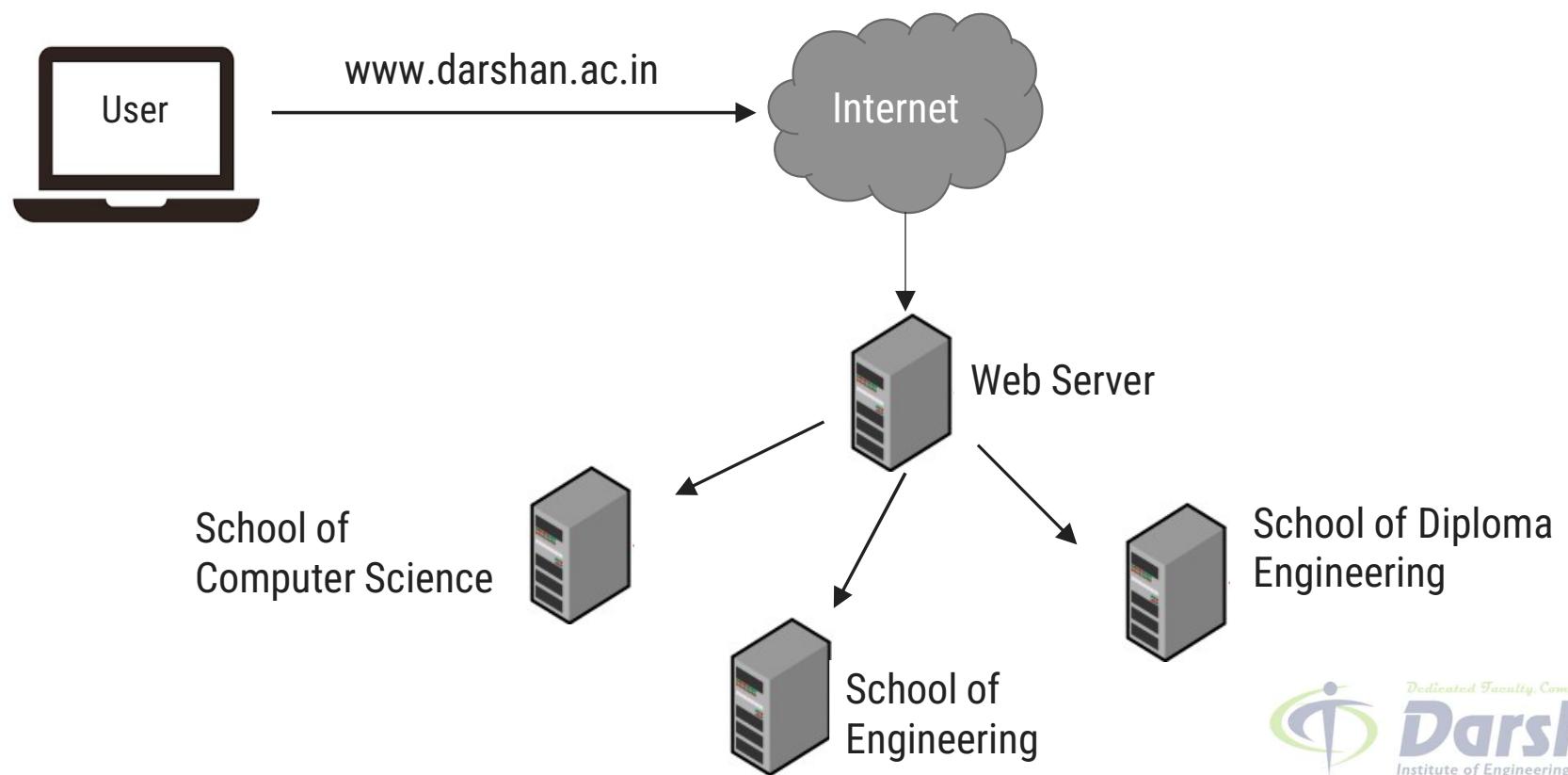
- ▶ “A distributed system is defined as one in which components at networked computers communicate and coordinate their actions only by passing messages.”
- ▶ “A Distributed system is collection of independent computers which are connected through network.”



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

Distributed Operating System

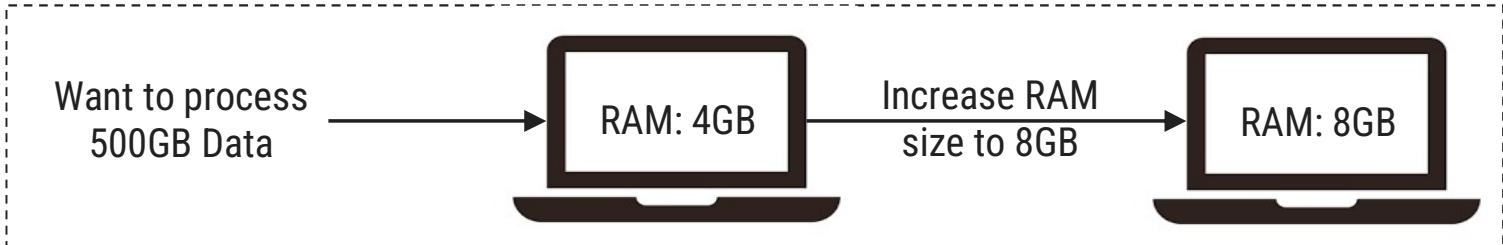
- ▶ A great example of distributed system is the web page of Darshan University.



Distributed Operating System

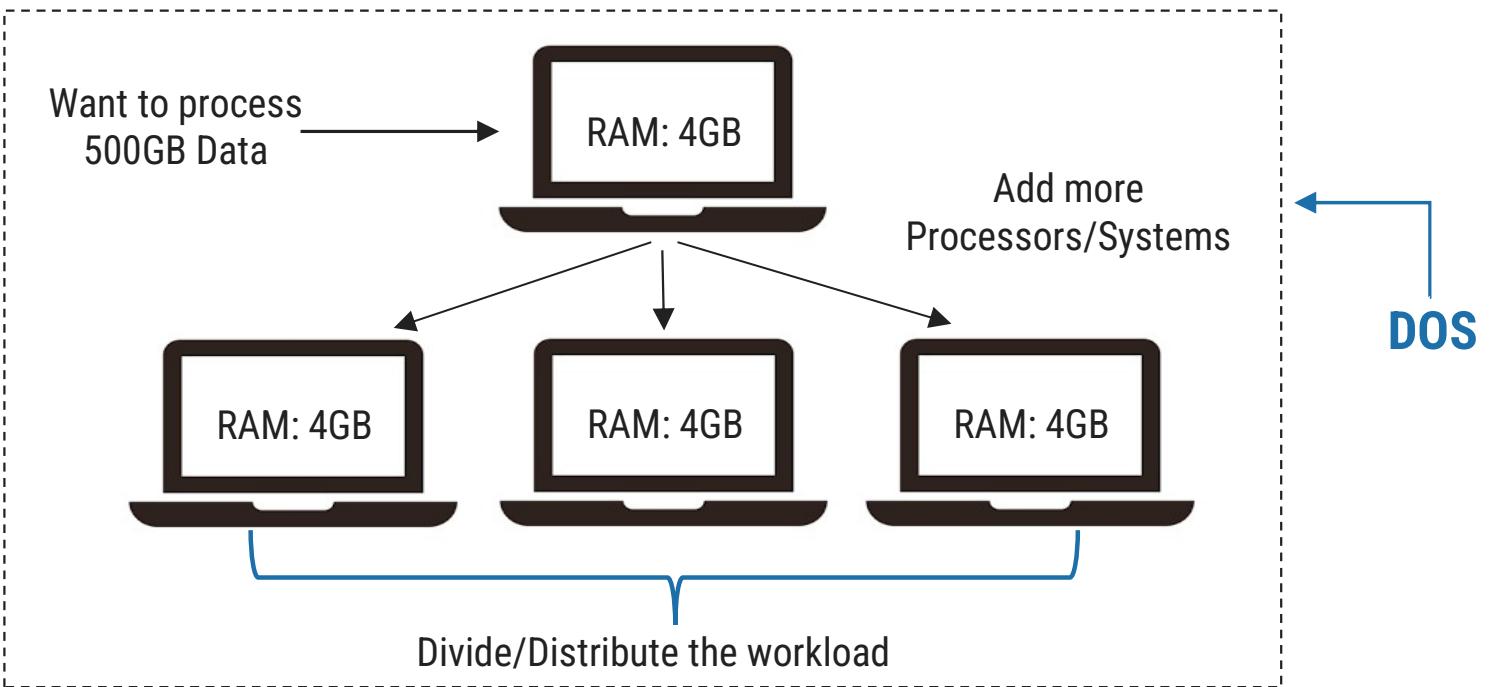
Scenario-1:

Vertical Scaling



Scenario-2:

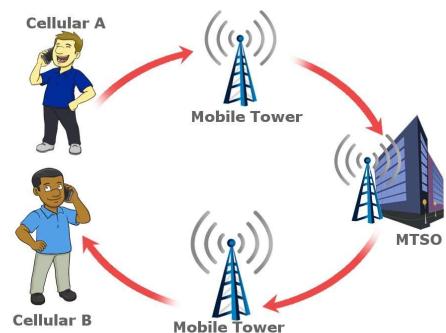
Horizontal Scaling



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



Computer network such internet



ATM machines



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

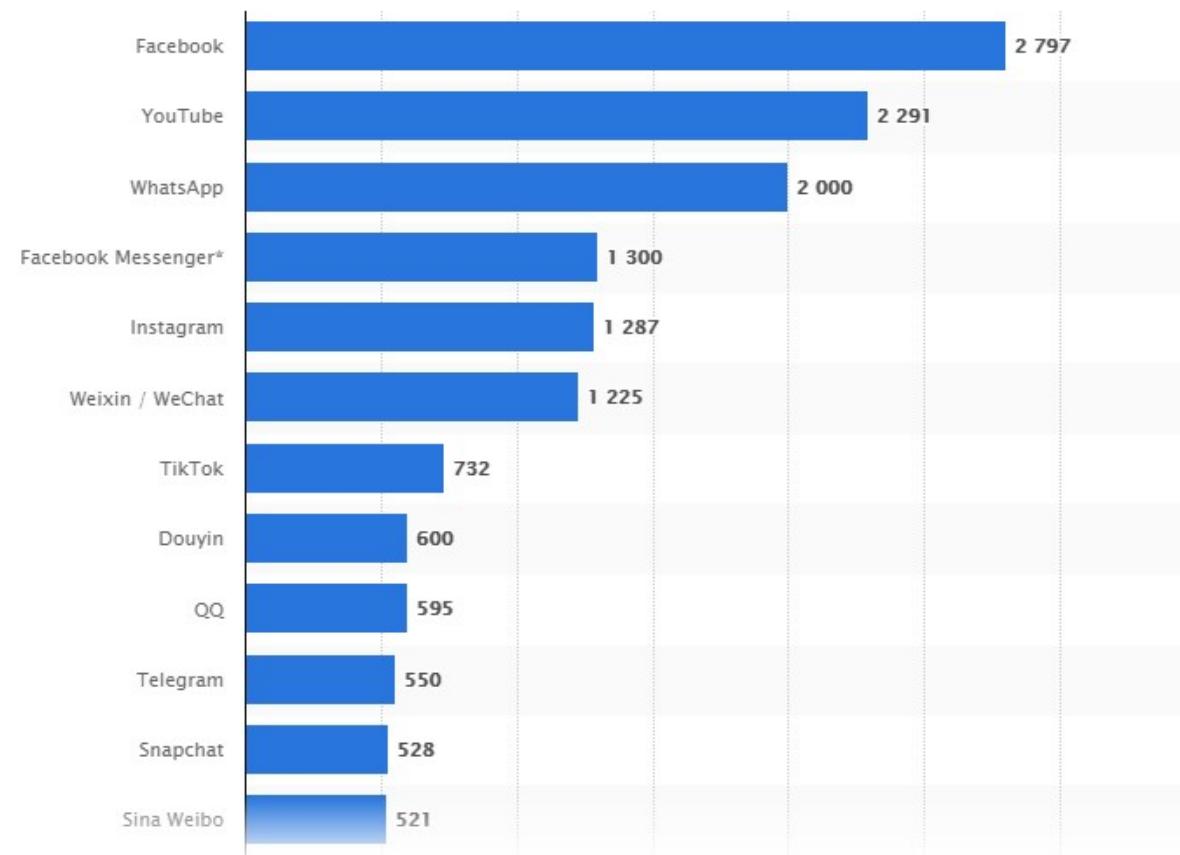


► Massively multiplayer online games:

- 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



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

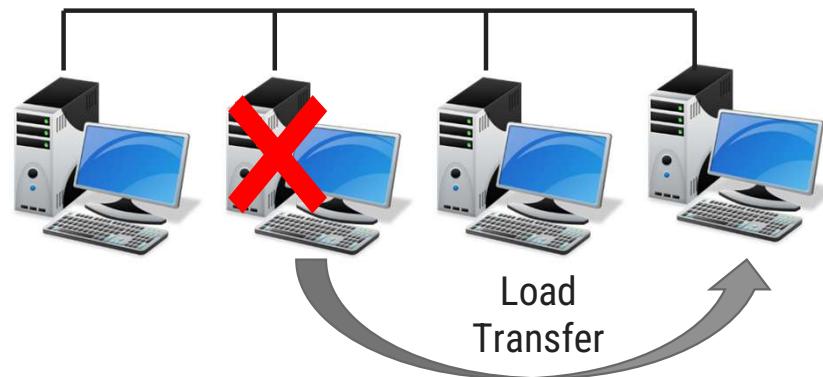
- ▶ **Economics:** A collection of microprocessors offer a better price/performance than mainframes. It is an cost effective way to increase computing power.



- ▶ **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.

Classification of Distributed System



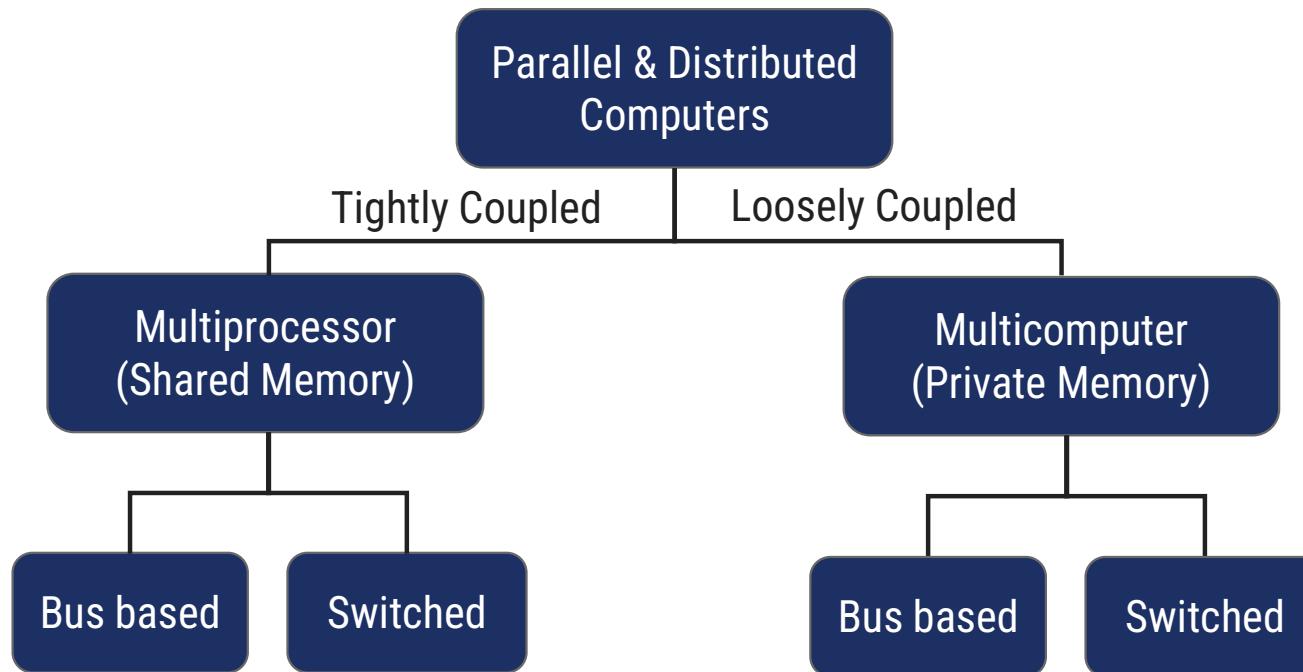
Based on Hardware



Based on number of
instructions and
DataStream

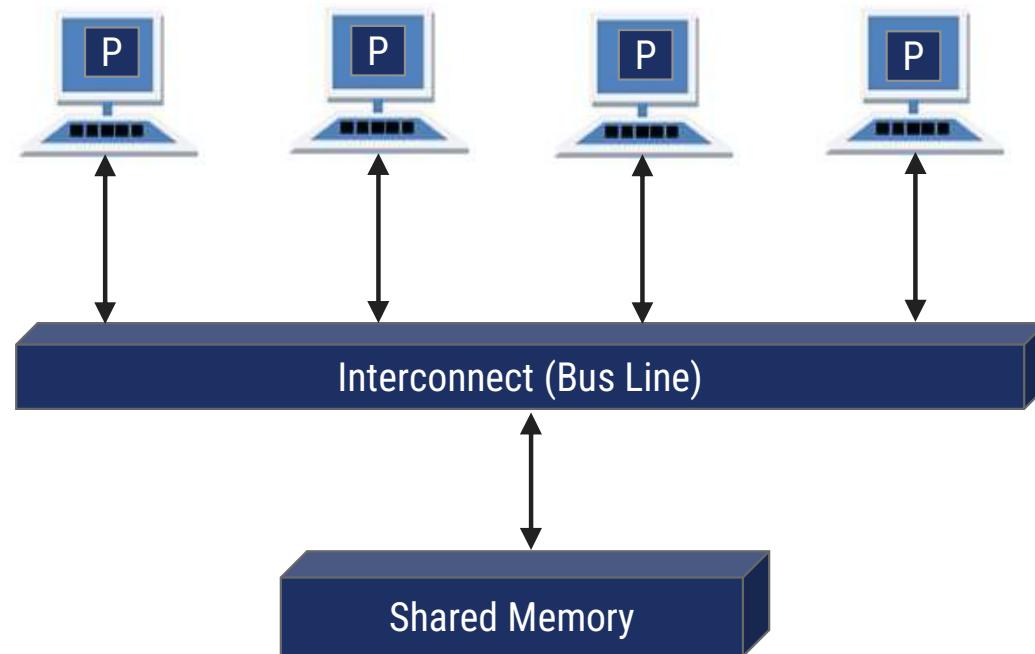
Classification based on Hardware

- 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.**



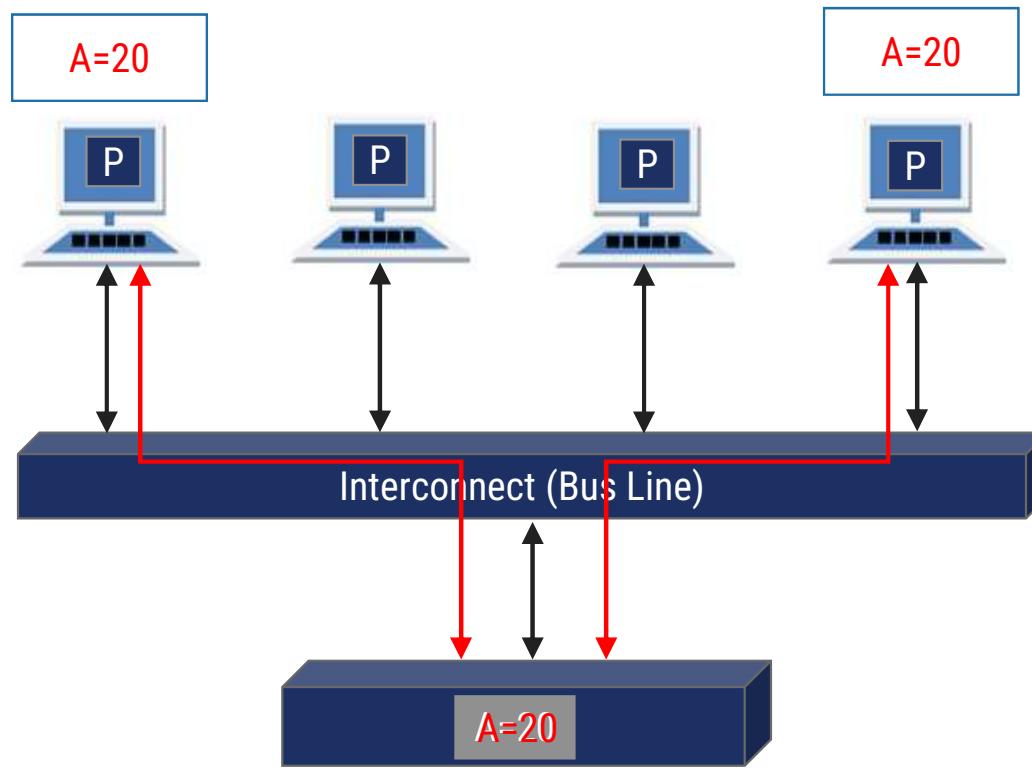
Tightly-Coupled OS(Shared Memory)

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



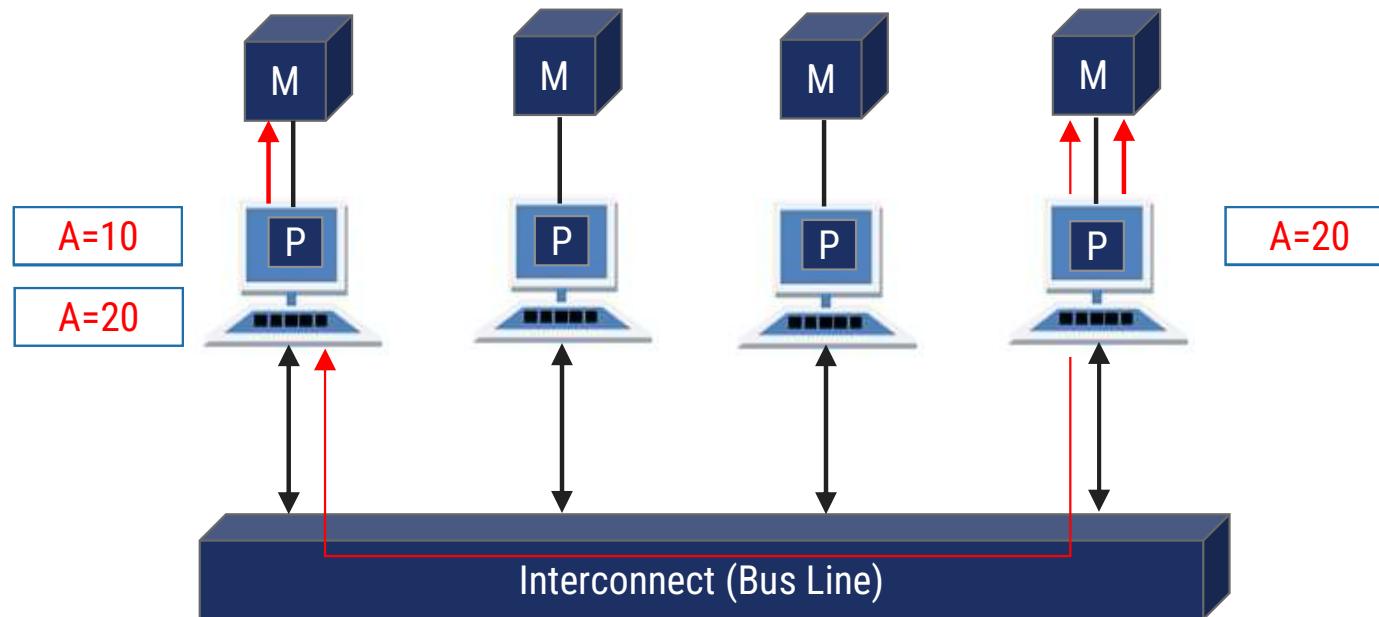
Tightly-Coupled OS(Shared Memory)

- ▶ **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**.

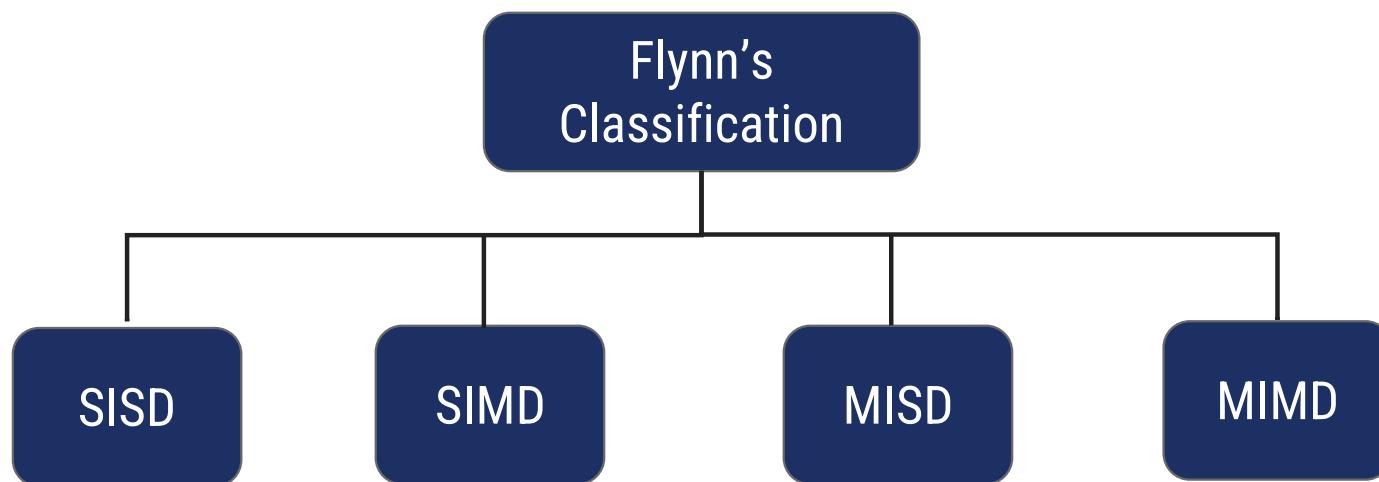


Classification based on Hardware

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 multicomputer Distributed Systems.	Manages multiprocessors & homogeneous multicomputer.
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

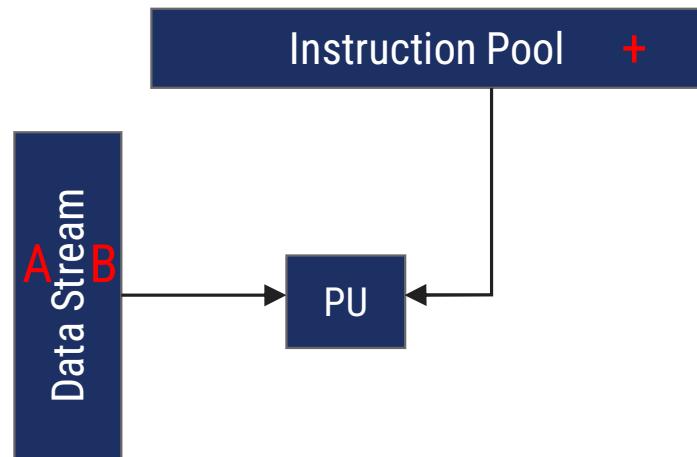
- ▶ According to Flynn's classification can be done based on the number of instruction streams and number of data streams.



Classification based on Instruction & DataStream

► Single instruction stream single data stream (SISD)

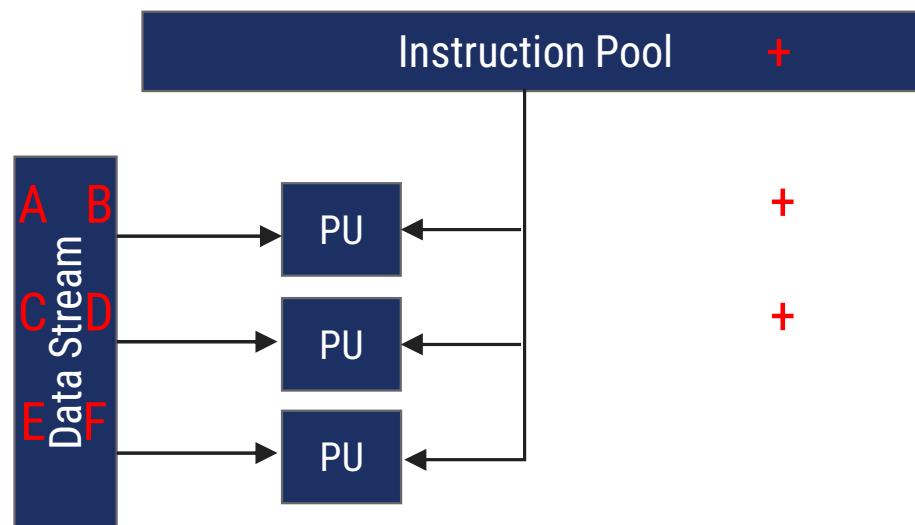
- One Program counter and one path to data memory.
- A computer is capable of executing one instruction at a time operating on one piece of data.
- An ordinary (Sequential) computer.



Classification based on Instruction & DataStream

► Single instruction stream, multiple data streams (SIMD)

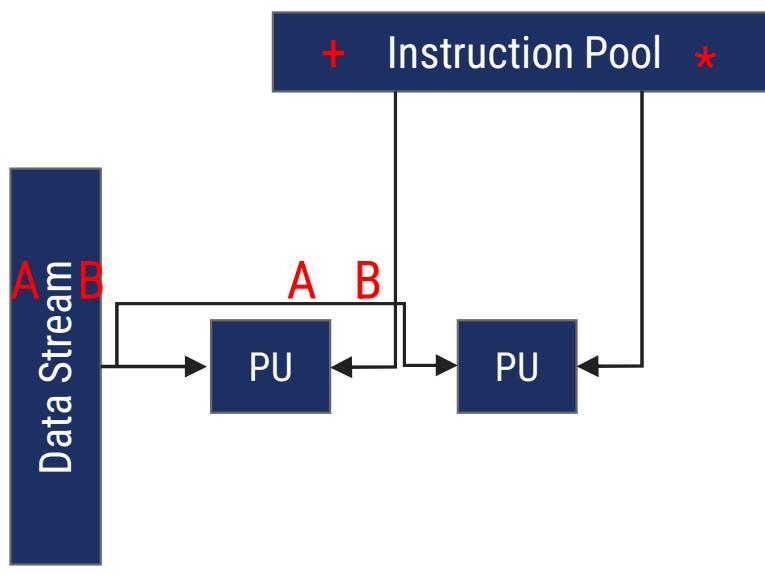
- One Program counter and multiple paths to data memory.
- A computer is capable of executing one instruction at a time, but operating on different pieces of data.



Classification based on Instruction & DataStream

► Multiple instruction streams, single data stream (MISD)

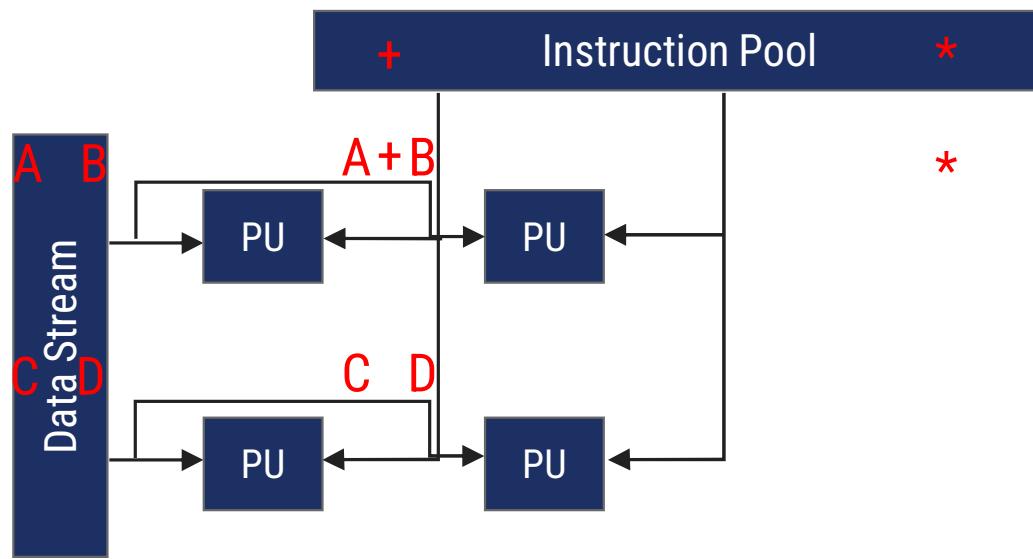
- No more computers fit this model.
- Uncommon architecture which is generally used for fault tolerance.



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.

Distributed Computing System Models

- ▶ Distributed Computing system models can be broadly classified into five categories.

Minicomputer Model

Workstation Model

Workstation – Server Model

Processor – Pool Model

Hybrid Model

Minicomputer Model

► Extension of Time sharing system

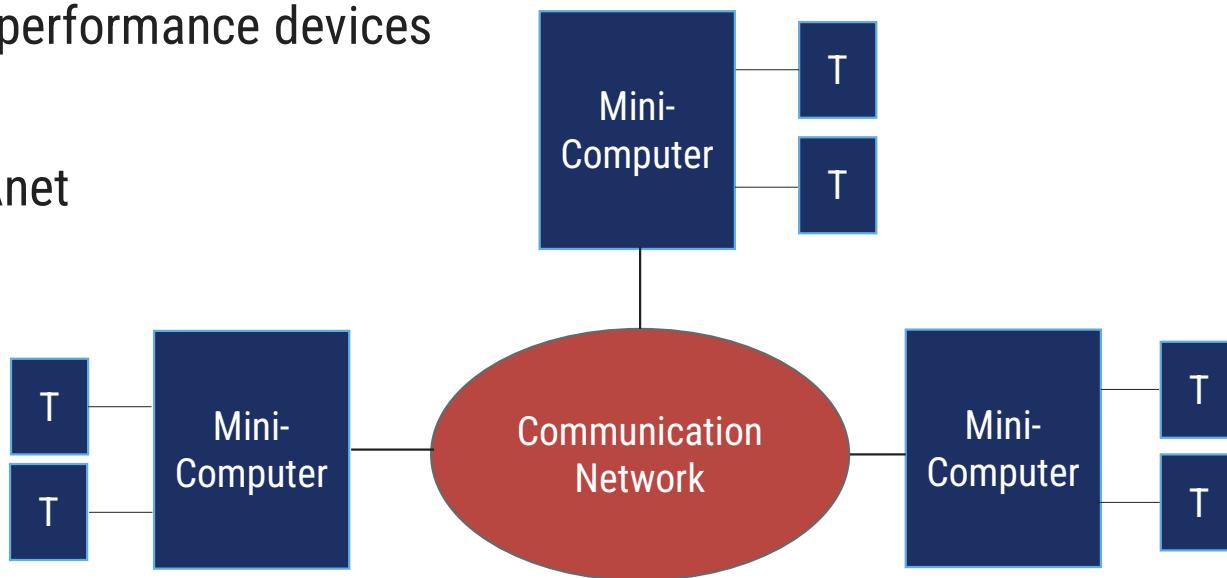
- User must log on his/her home minicomputer.
- Thereafter, he/she can log on a remote machine by telnet.

► Resource sharing

- Database
- High-performance devices

► Example:

- ARPAnet



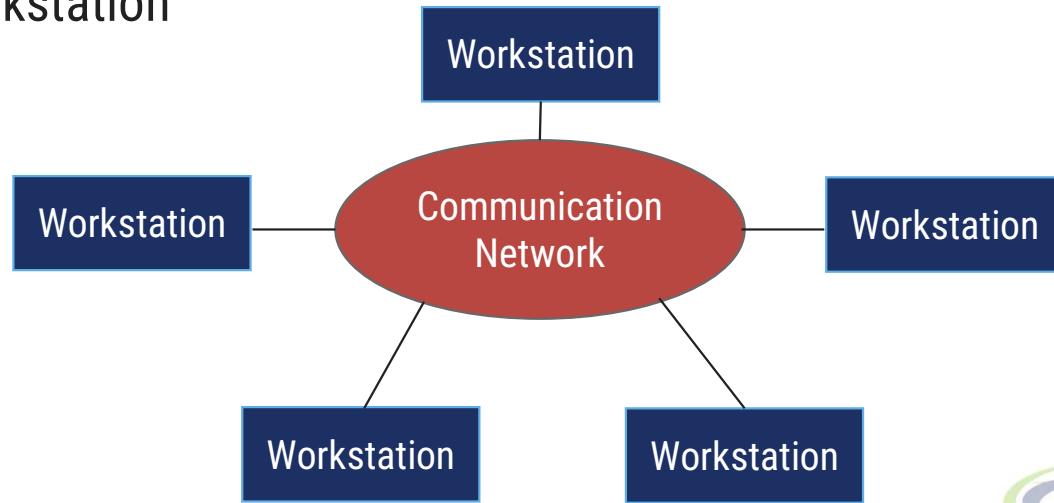
Workstation Model

▶ Process migration

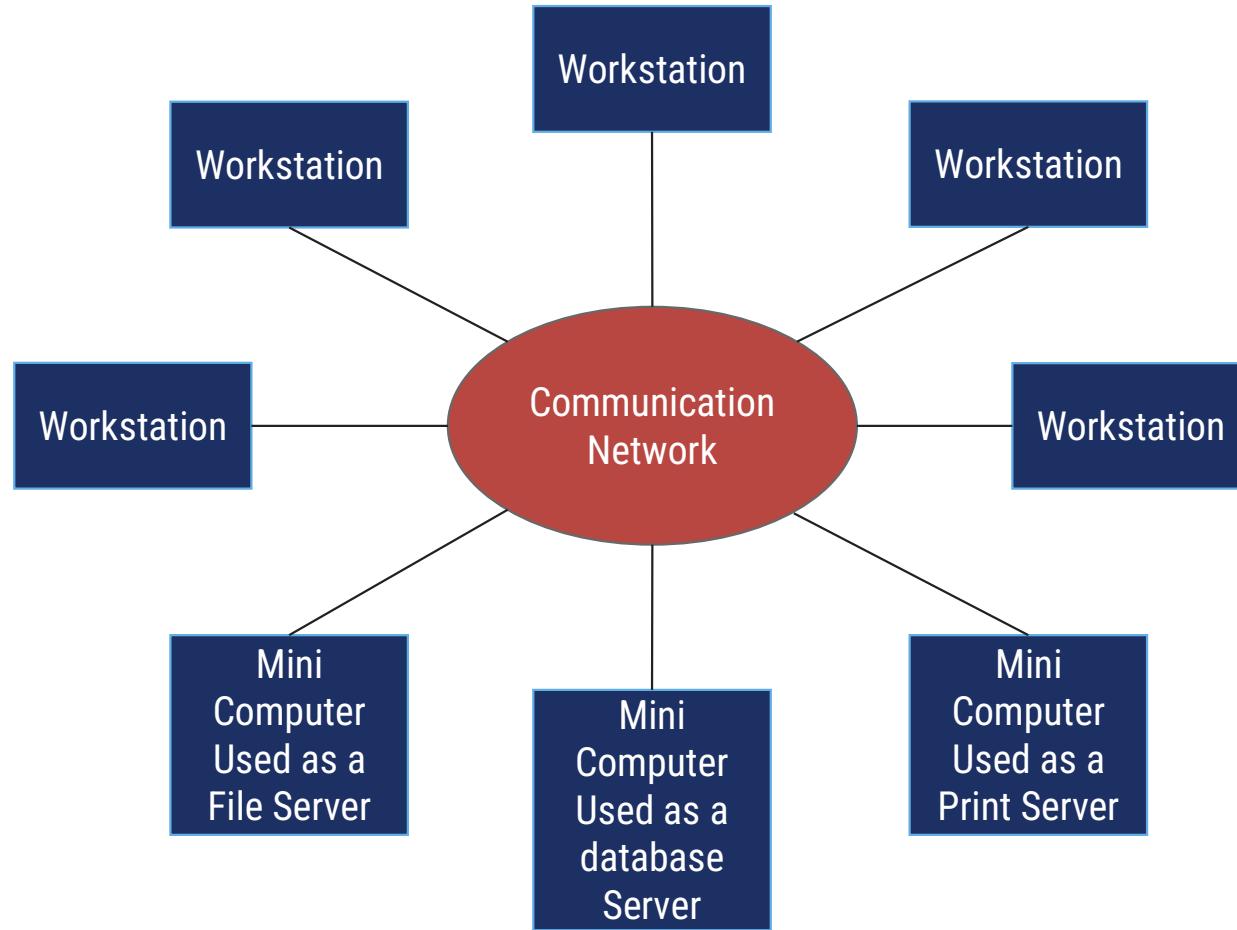
- Users first log on his/her personal workstation.
- If there are idle remote workstations, a heavy job may migrate to one of them.

▶ Problems:

- What if a user log on the remote machine
- How to find an idle workstation
- How to migrate a job



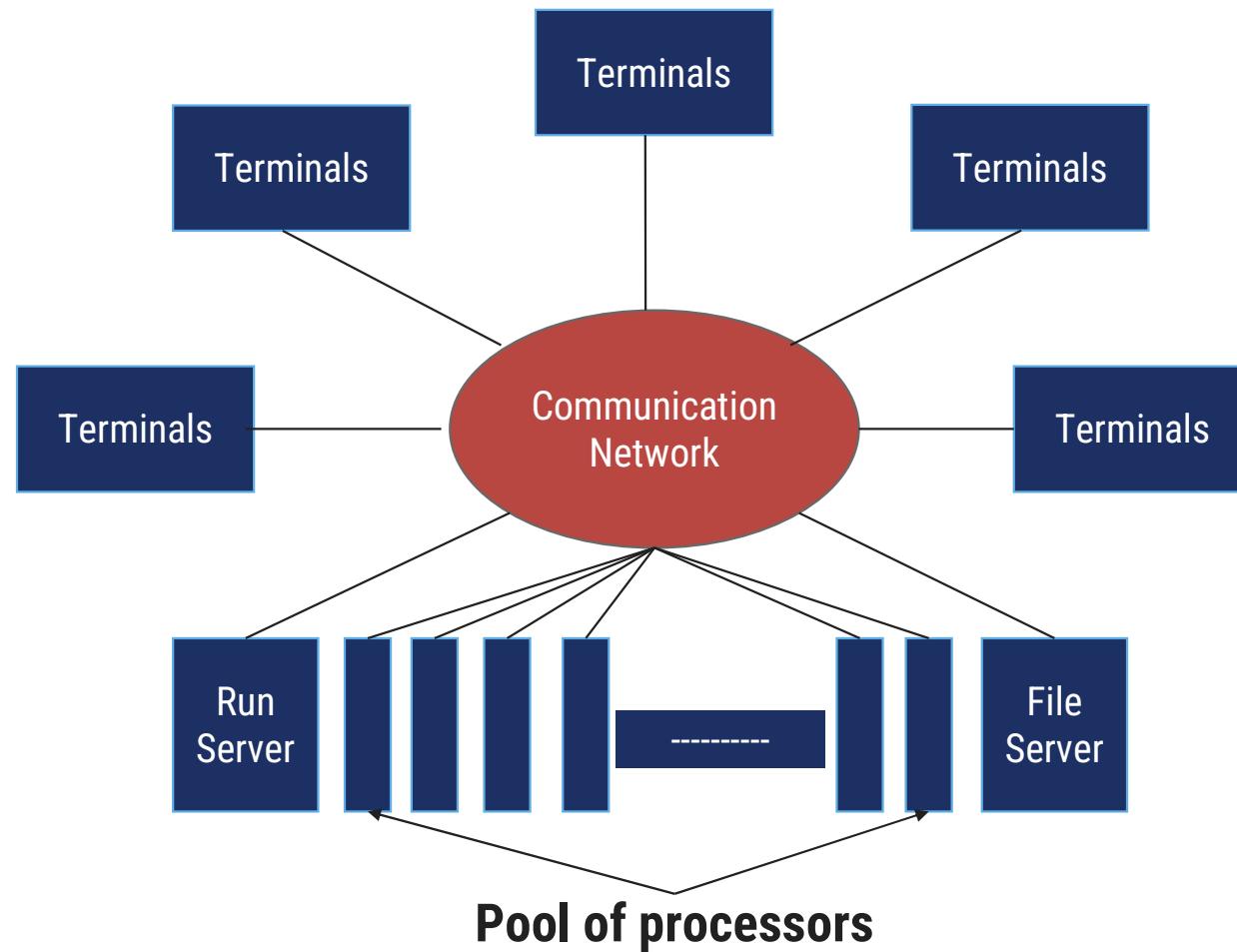
Workstation-Server Model



Workstation-Server Model

- ▶ Client workstations
 - Diskless
 - Graphic/interactive applications processed in local.
 - All file, Print, http and even cycle computation requests are sent to servers.
- ▶ Server minicomputers
 - Each minicomputer is dedicated to one or more different types of services.
- ▶ Client-Server model of communication
 - RPC (Remote Procedure Call)
 - RMI (Remote Method Invocation)
 - A Client process calls a server process function.
 - No process migration invoked

Processor-Pool Model



Processor-Pool Model

▶ Clients:

- They log in one of terminals (diskless workstations)
- All services are dispatched to servers.

▶ Servers:

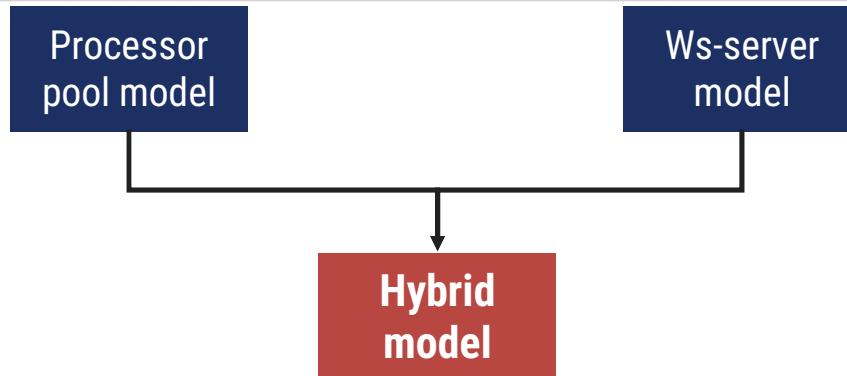
- Necessary number of processors are allocated to each user from the pool.

▶ Better utilization of resources.

▶ Example:

- Web Search Engines

Hybrid Model



- ▶ Advantages of the workstation-server and processor-pool models are combined to build a hybrid model.
- ▶ It is built on the workstation-server model with a pool of processors.
- ▶ Processors in the pool can be allocated dynamically for large computations, that cannot be handled by the workstations, and require several computers running concurrently for efficient execution.
- ▶ This model is more expensive to implement than the workstation-server model or the processor-pool model.

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 vary 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.

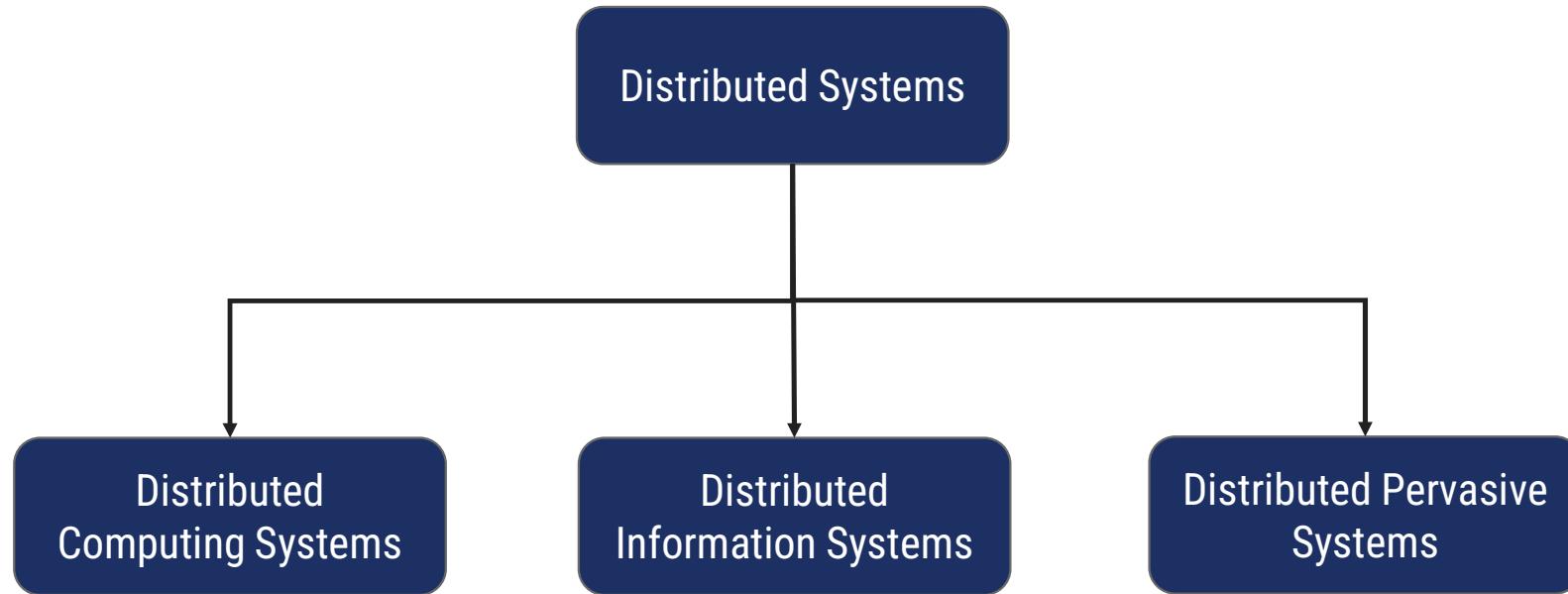
Heterogeneity

It consists of dissimilar hardware or software systems.

Security

Must be protected against destruction and unauthorized access.

Types of Distributed Systems



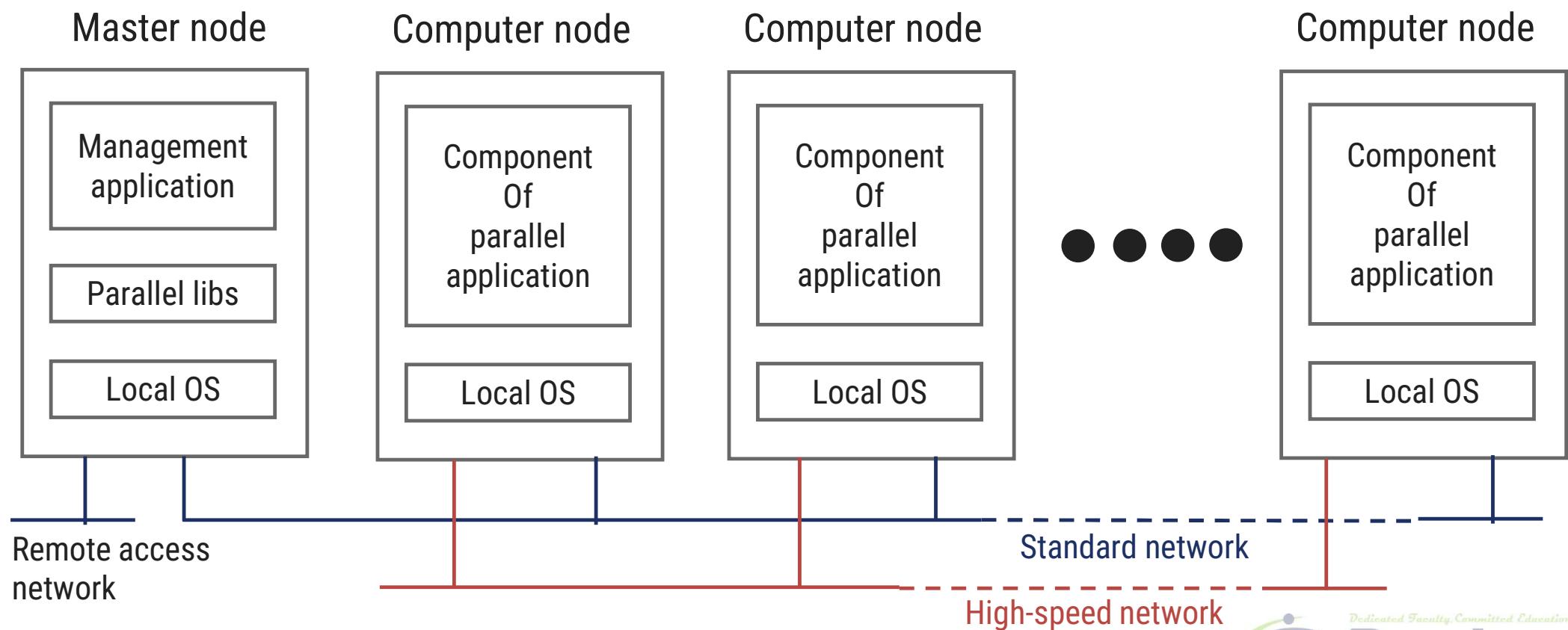
Distributed Computing Systems

- ▶ 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 ratio 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.



Architecture of a Grid Computing System

▶ Fabric layer

- Interface to local resources at a specific site.

▶ Connectivity layer

- Communication protocols for grid transactions spanning over multiple resources, plus security protocols for authentication

▶ Resource layer

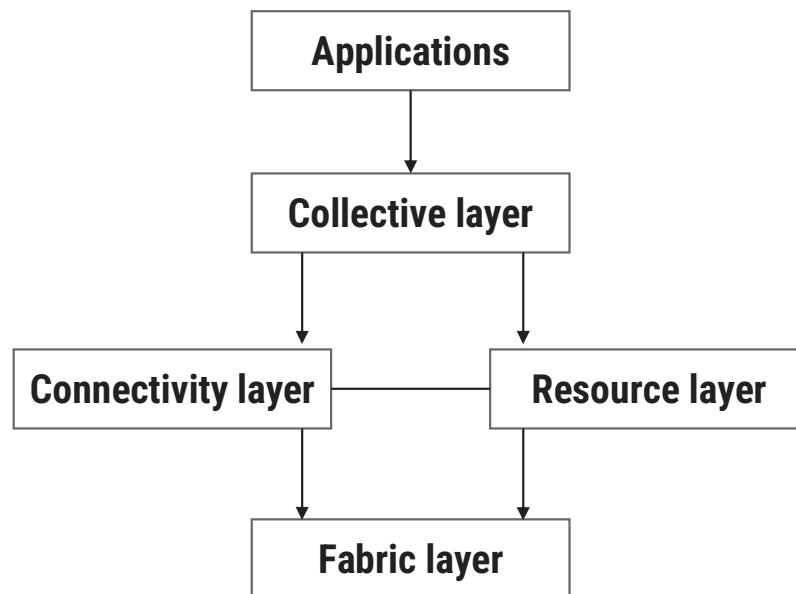
- Management of single resources—e.g., access control

▶ Collective layer

- Handling access to multiple resources—resource discovery, allocation,

▶ Application layer

- Applications operating in the virtual organization.



Grid Middleware layer

collective, connectivity, and resource layers.

- ▶ Provide access to and management of resources that are potentially dispersed across multiple sites.
- ▶ Shift toward a service-oriented architecture in which sites offer access to the various layers through a collection of Web services
- ▶ Led to the definition of an alternative architecture known as the **Open Grid Services Architecture (OGSA)**.
- ▶ Consists of various layers and many components, making it rather complex.

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

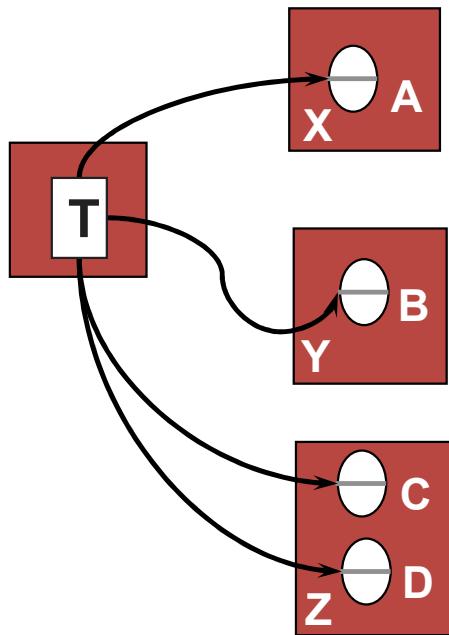
- ▶ A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties (ACID):
 - **Atomicity**: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected 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

Example Primitives for Transactions

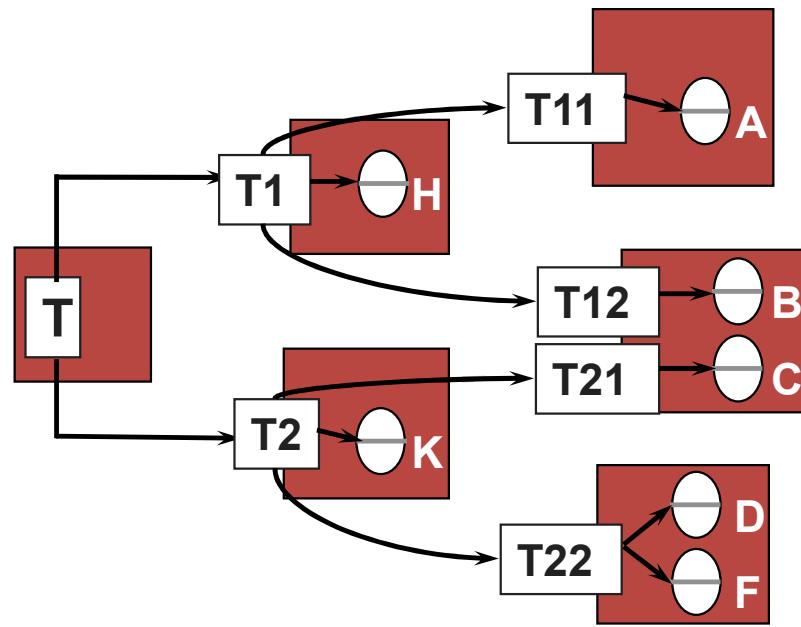
Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Distributed Transactions(flat and nested)

Flat Transactions



Nested Transactions

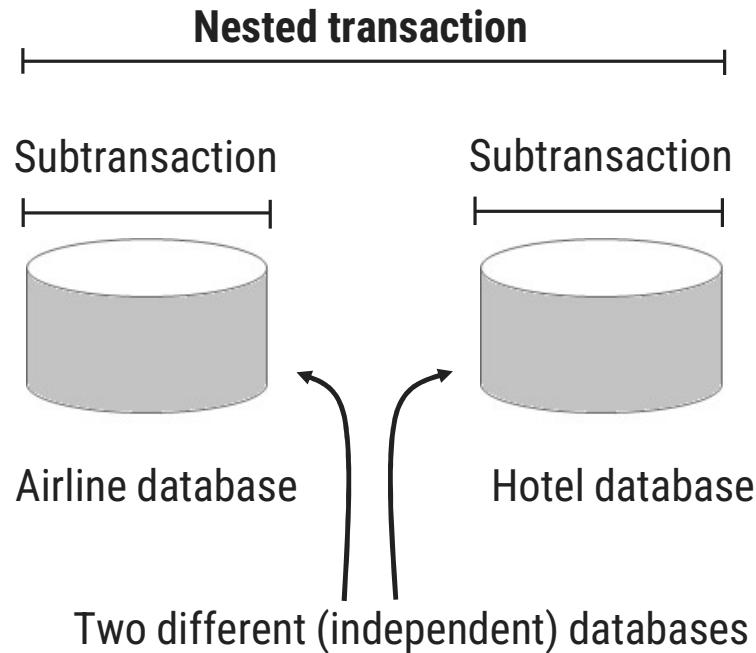


- ▶ A flat client transaction completes each of its requests before going on to the next one.
- ▶ Therefore, each transaction accesses servers' objects sequentially

- ▶ In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting

Nested Transactions

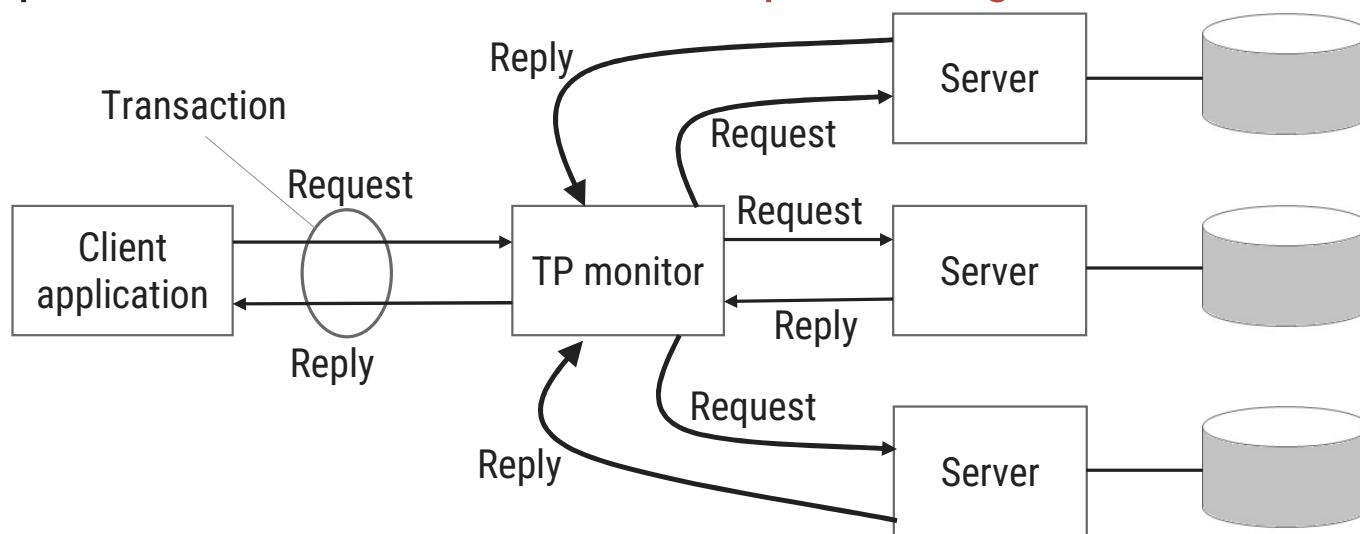
- ▶ A nested transaction is made of a number of sub transactions.



- ▶ The top-level transaction may fork off children that run in parallel with one another, on different machines, to gain performance or simplify programming.

TP(Transaction Processing) Monitor

- In the early days of enterprise middleware systems, the component that handled distributed (or nested) transactions formed the core for integrating applications at the server or database level.
- This component was called a **Transaction processing monitor or TP monitor** for short.



- Its main task was to allow an application to access multiple server/databases by offering it a transactional programming mode

Popular TP Monitor Products

- ▶ TUXEDO from BAE Systems



- ▶ Microsoft Transaction Server (MTS)



- ▶ IBM CICS (Customer Information Control System)

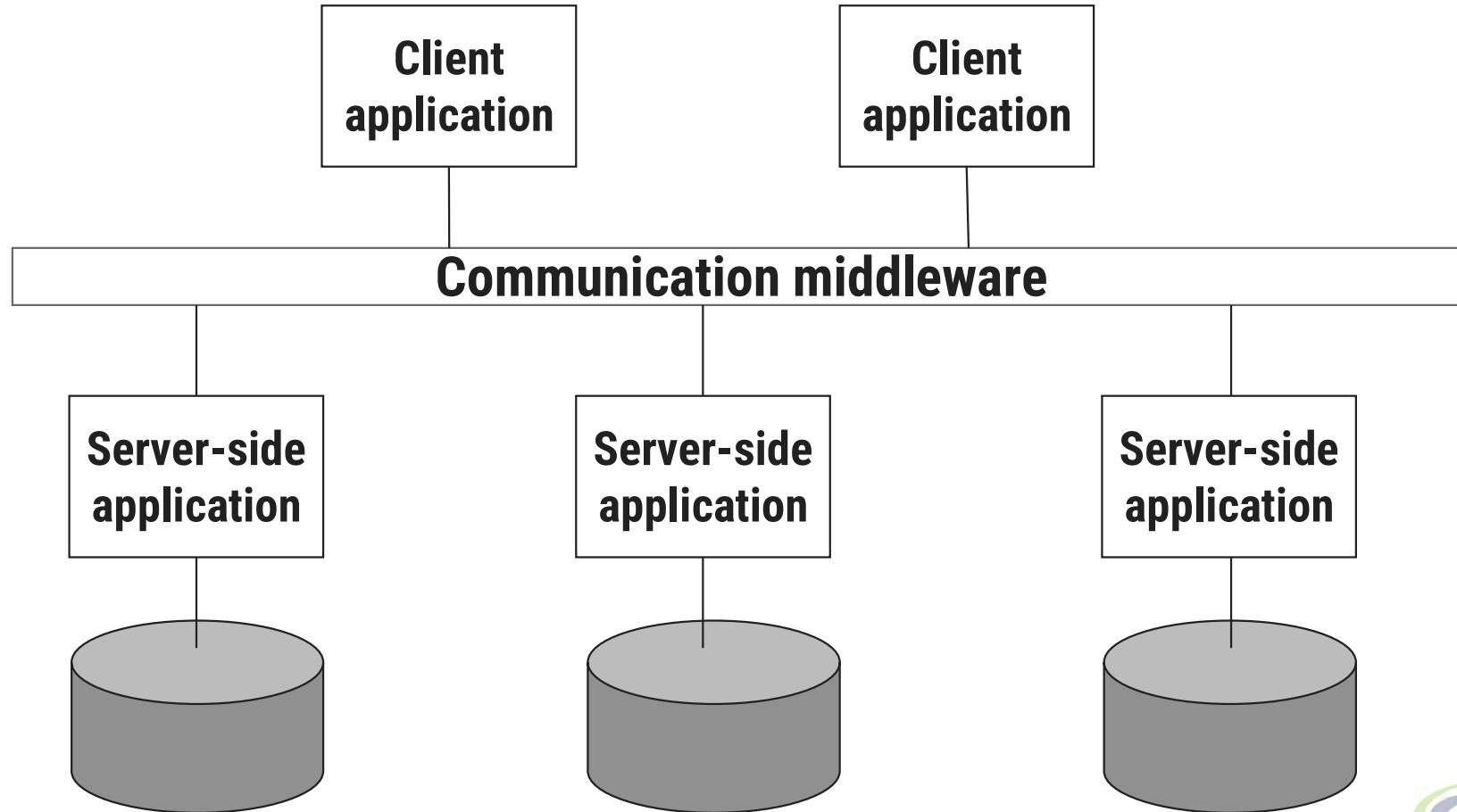


Enterprise Application Integration

- ▶ 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**



Enterprise Application Integration



Communication Middleware

Several types of communication middleware exist.

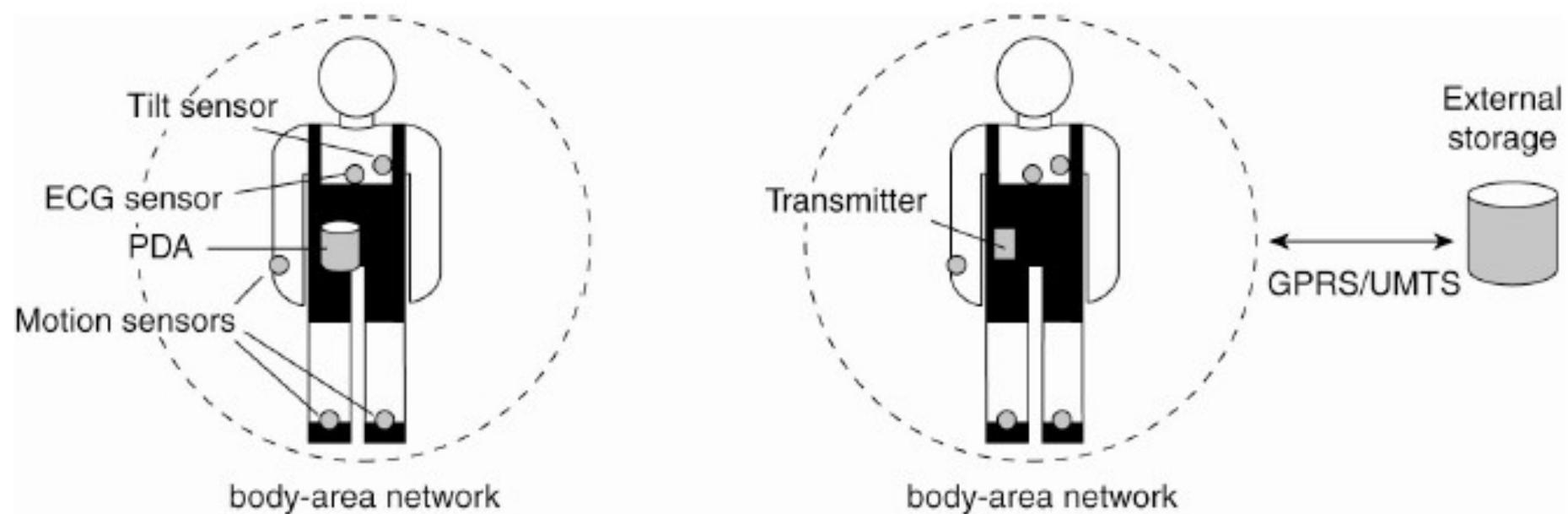
- ▶ **Remote procedure calls (RPC)** - an application component can send a request to another application component by doing a local procedure call, which results in the request being packaged as a message and sent to the callee.
- ▶ **Remote method invocations (RMI)** – An RMI is the same as an RPC, except that it operates on objects instead of applications.
- ▶ Problems with RPC and RMI:
 - The caller and callee both need to be up and running at the time of communication.
 - They need to know exactly how to refer to each other.
- ▶ **Results: Message-oriented middleware (MOM)** - applications send messages to logical contact points, often described by means of a subject.

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:

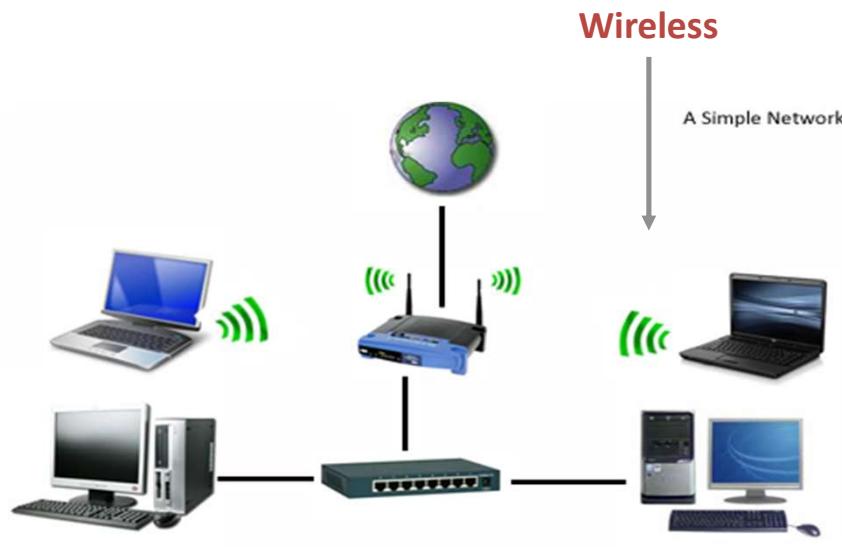
- General lack of human administrative control
- Devices can be configured by their owners
- They need to automatically discover their environment and fit in as best as possible

► Requirements for pervasive applications

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

Computer Network

- ▶ A computer network or data network is a telecommunications network which allows computers to exchange data.
- ▶ In computer networks, networked computing devices exchange data with each other using a data link.
- ▶ The connections between nodes are established using either cable media or wireless media.
- ▶ The best-known computer network is the Internet.



Computer Network in DOS

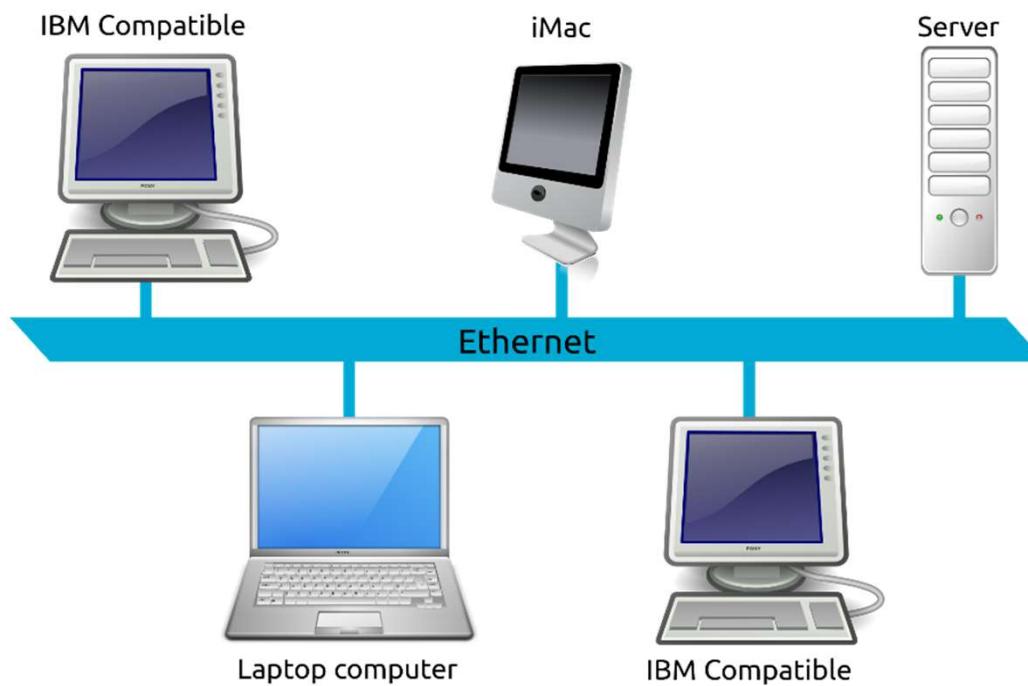
- ▶ A Distributed System is basically a computer network whose nodes have their own local memory and may also have other hardware and software resources.
- ▶ A Distributed System relies on the underlying computer network for the communication of data and control the information between nodes.
- ▶ The performance and reliability of a Distributed system depends on the underlying computer network.

Classification of Network

- ▶ Network types depends on **how large they are** and how much of an **area they cover geographically**.
- ▶ Networks are classified based on
 - Size
 - Capabilities
 - Geographical distance

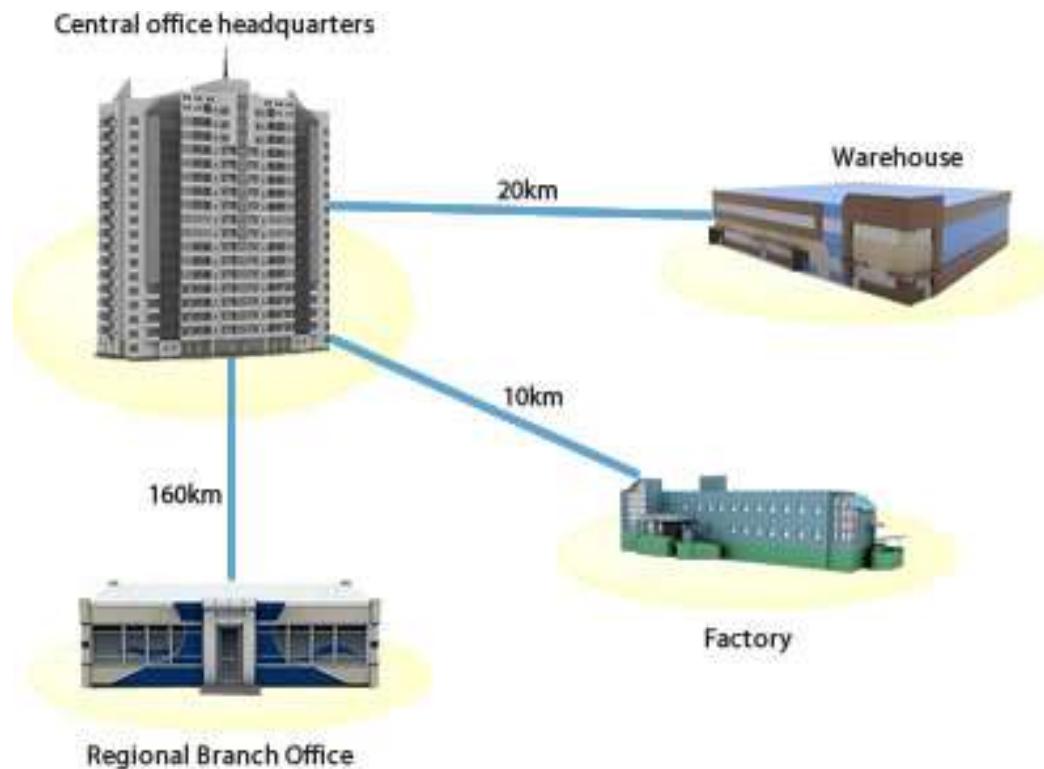
Local Area Network(LAN)

- ▶ A group of devices(computers, servers, switches, and printers) that are located in the same building.



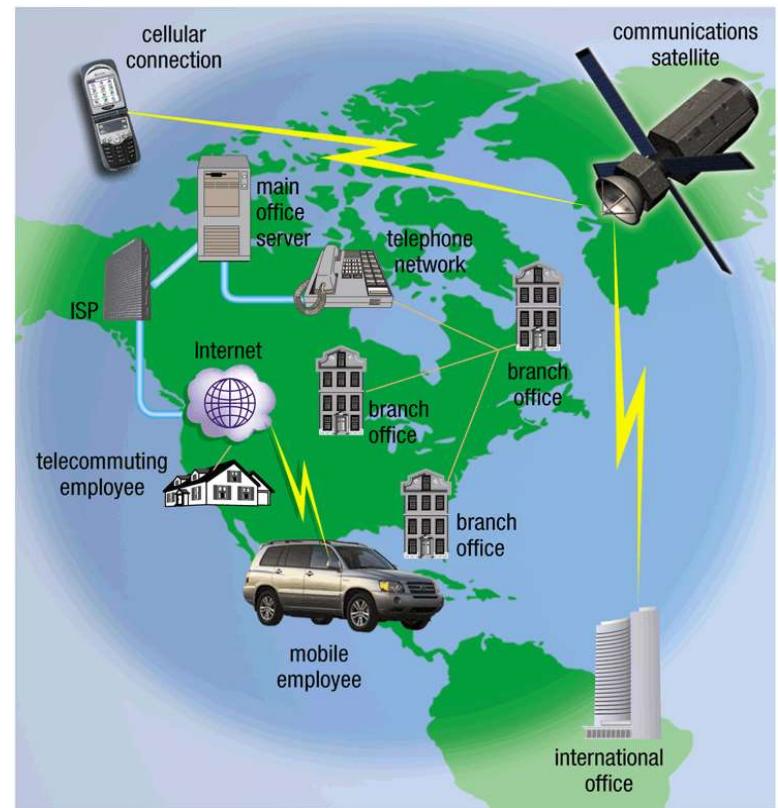
Metropolitan Area Network(MAN)

- ▶ Larger than LAN.
- ▶ Spans over several buildings in a city or town.



Wide Area Network(WAN)

- ▶ Largest type of network.
- ▶ Spans over large geographic area (e.g., country).
- ▶ The **internet** is example of WAN.



Classification of Network

Basis of Comparison	LAN	MAN	WAN
Expands to	Local Area Network	Metropolitan Area Network	Wide Area Network
Meaning	A network that connects a group of computers in a small geographical area.	It covers relatively large region such as cities, towns.	It spans large locality & connects countries together.
Ownership of Network	Private	Private or Public	Private or Public
Design and Maintenance	Easy	Difficult	Difficult
Propagation Delay	Short	Moderate	Long
Speed	High	Moderate	Low

Classification of Network

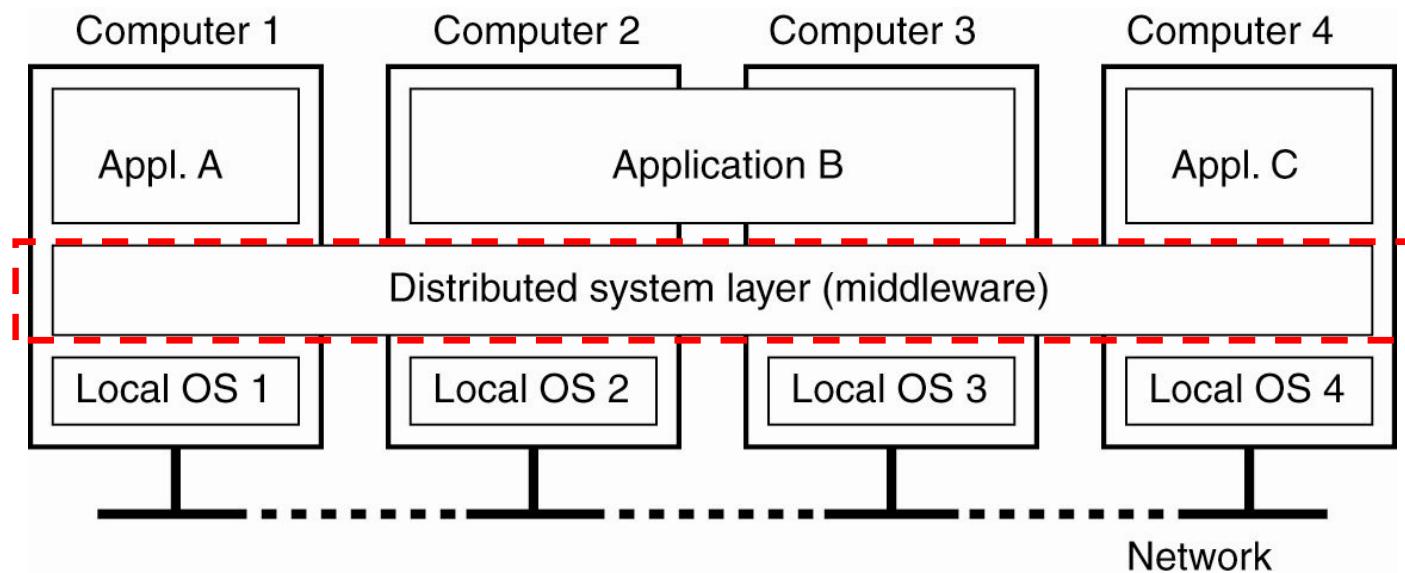
Basis of Comparison	LAN	MAN	WAN
Equipment Needed	Switch, Hub	Modem, Router	Microwave, Radio Transmitters & Receivers
Range(Approx.)	1 to 10 km	In 100 km	Beyond 100 km
Used for	College, School, Hospital	Small towns, City	Country/Continent

Wireless Network

- ▶ A wireless LAN uses wireless transmission medium.
- ▶ Wireless Network used to have
 - high prices
 - low data rates
 - licensing requirements
- ▶ Popularity of wireless LANs has grown rapidly.



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)

Unit-2

Basics of Architectures, Processes and Communication



Prof. Umesh H. Thoriya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
✉ umesh.thoriya@darshan.ac.in
📞 9714233355



Topics to be covered

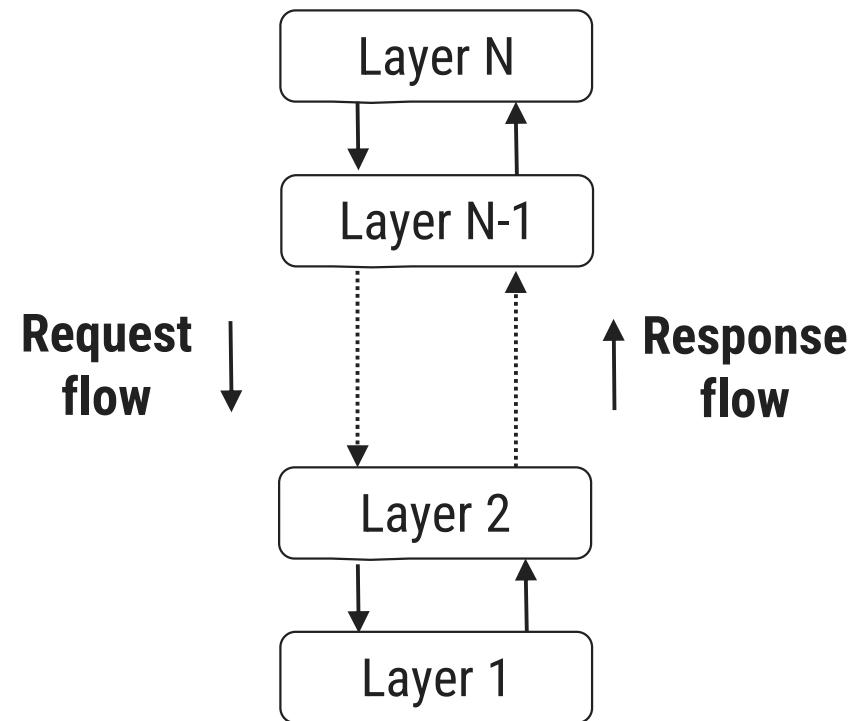
- Architectures - Types of System Architectures
- Self Management in Distributed Systems
- Processes - Basics of Threads
- Virtualization
- Roles of Client and Server
- Code Migration
- Communication - Types of Communications
- Remote Procedure Calls
- Message-Oriented Communication
- Stream-Oriented Communication
- Multicasting

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

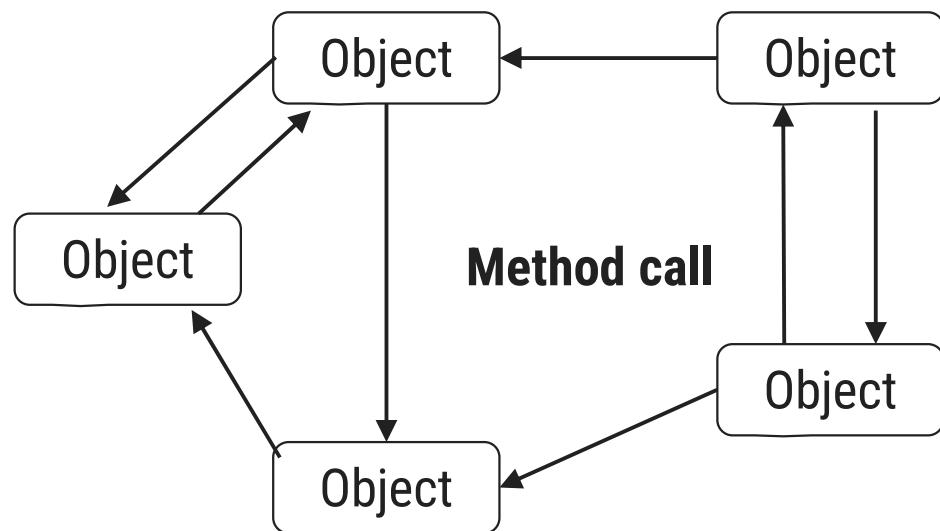
Layered architectures

- ▶ 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.



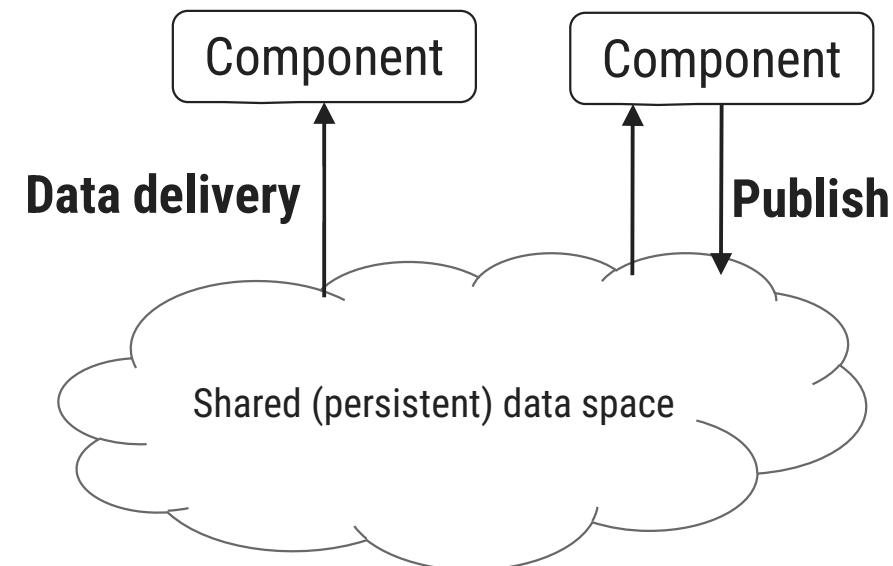
Object-based architectures

- ▶ Each object corresponds a component
- ▶ Components are connected through a (remote) procedure call mechanism.
- ▶ The layered and object-based architectures still form the most important styles for large software systems



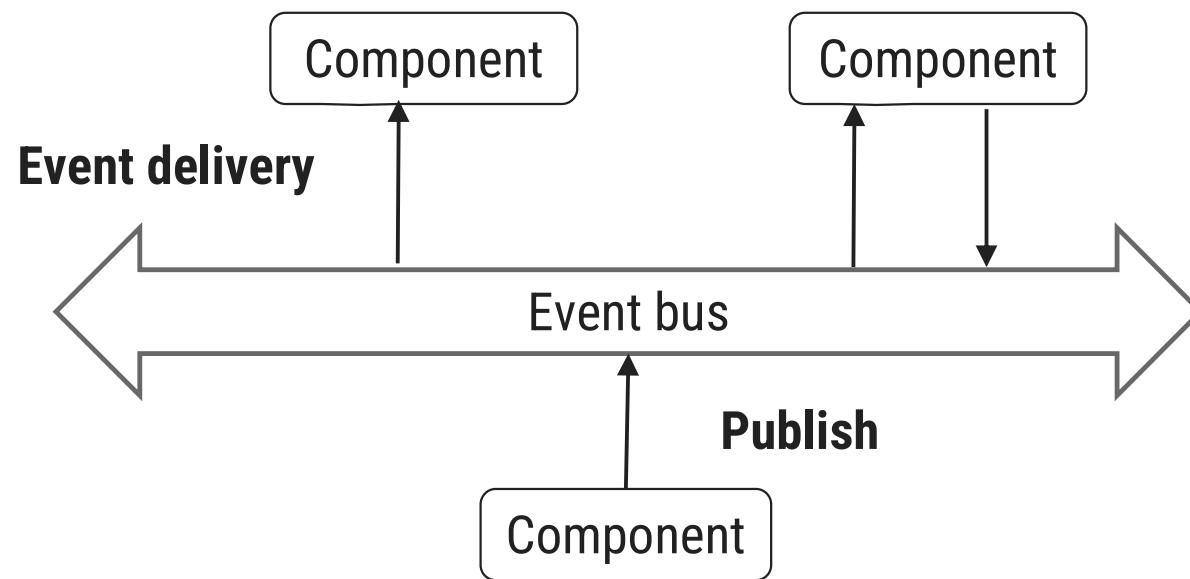
Data-centered architectures

- ▶ It evolve around the idea that processes communicate through a common (passive or active) repository.
- ▶ 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.
- ▶ 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.



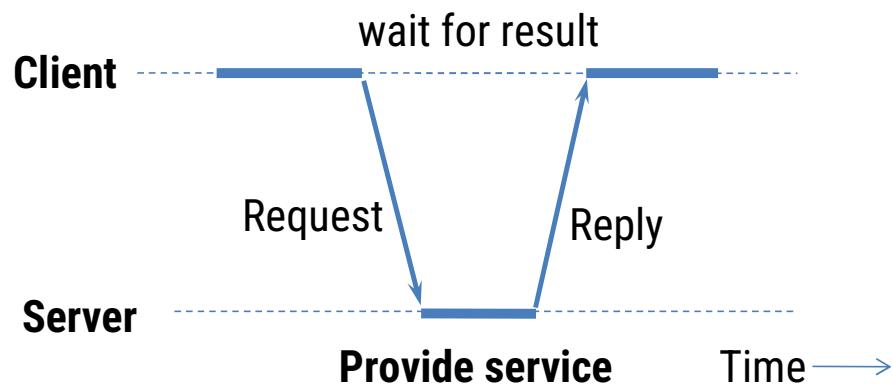
System Architecture

There are three views toward system architectures:

- ▶ Centralized Architectures
- ▶ Decentralized architectures
- ▶ Distributed 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

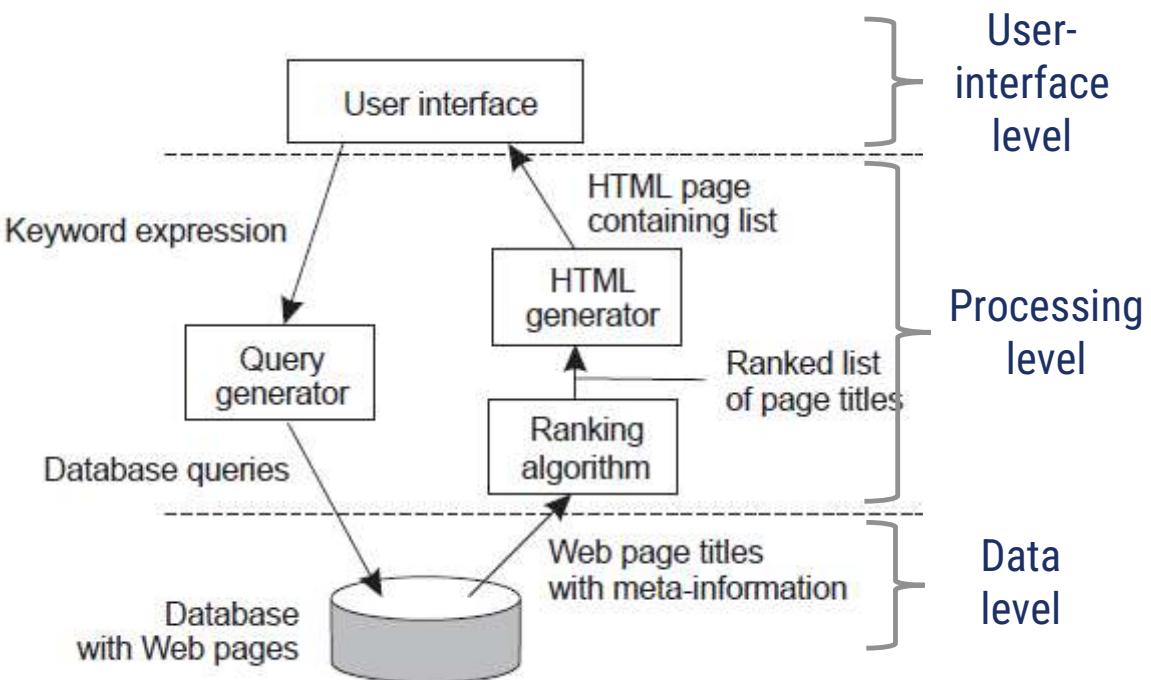
3. The data level

- 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:

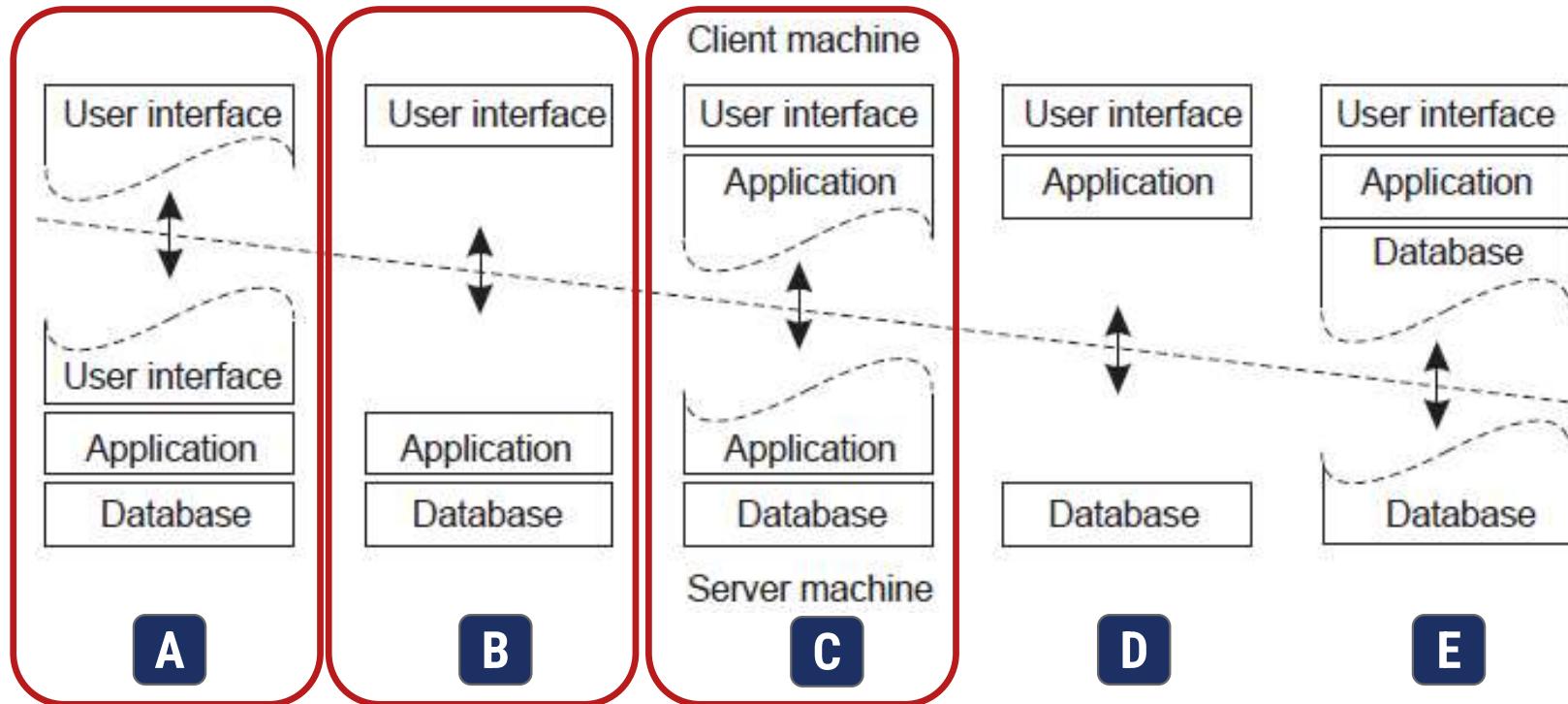
1. A client machine containing only the programs implementing (part of) the user-interface level
2. A server machine containing the rest, that is the programs implementing the processing and data level

► **Single-tiered:** dumb terminal/mainframe configuration

► **Two-tiered:** client/single server configuration

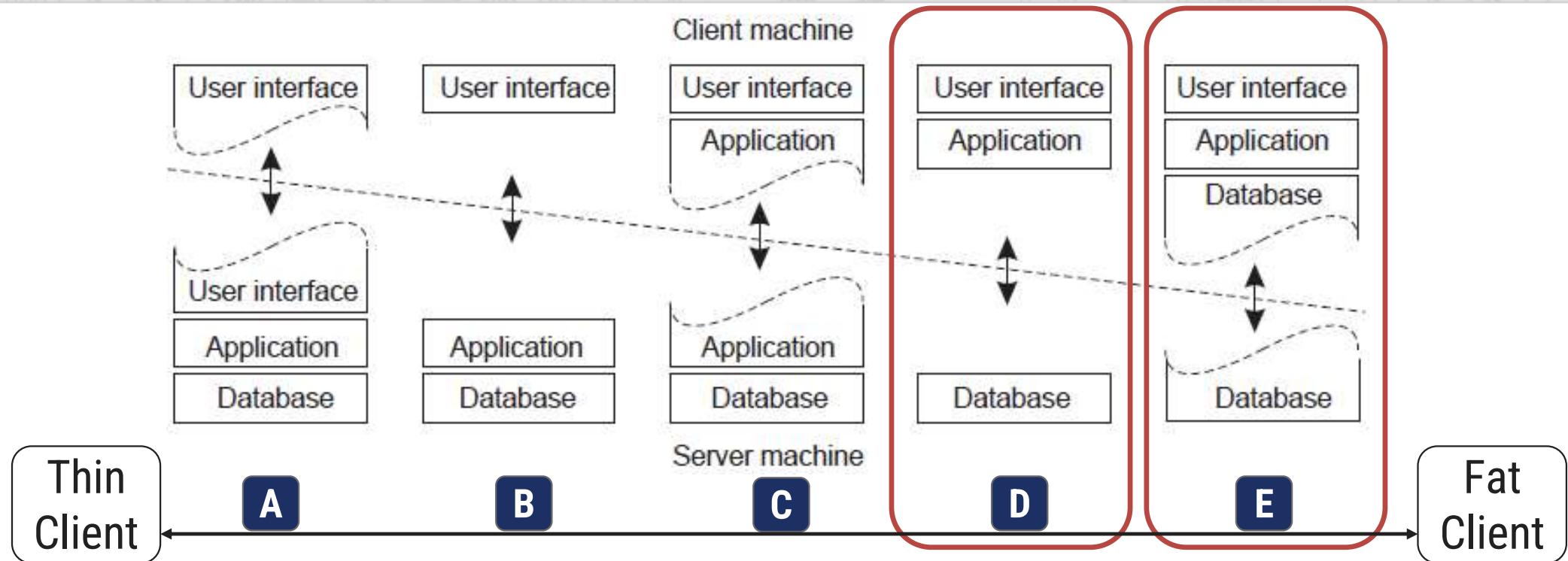
► **Three-tiered:** each layer on separate machine

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



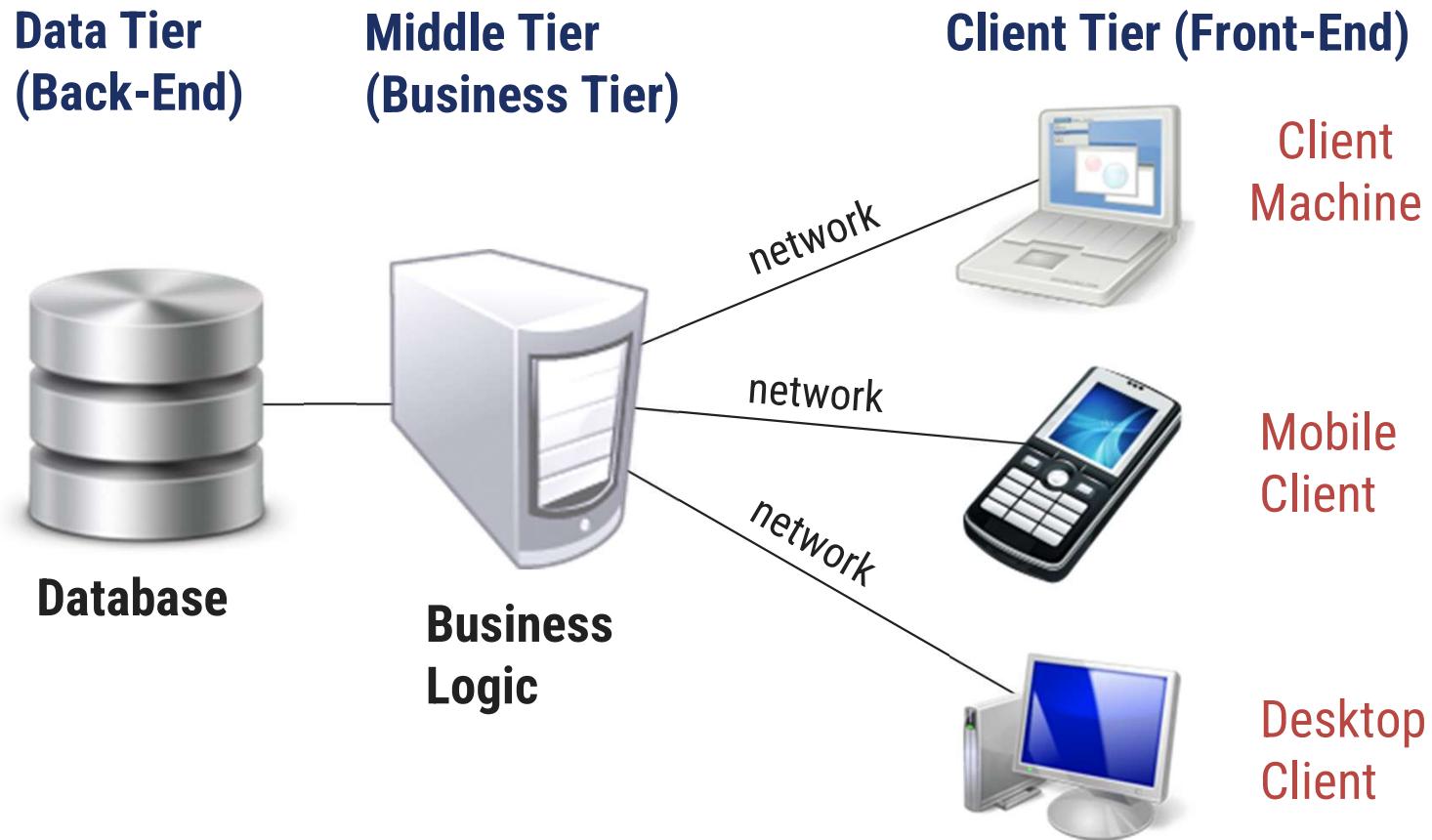
- ▶ Case-A: Only the terminal-dependent part of the user interface
- ▶ Case- B: Place the entire user-interface software on the client side
- ▶ Case- C: Move part of the application to the front end

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



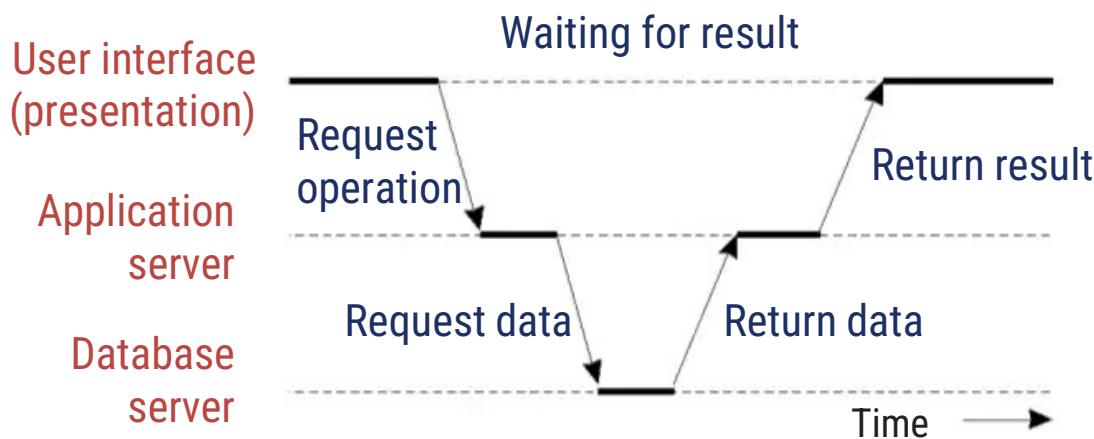
- ▶ **Case-D:** Used where the client machine is a PC or workstation, connected through a network to a distributed file system or database
- ▶ **Case- E:** Used where the client machine is a PC or workstation, connected through a network to a distributed file system or database

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

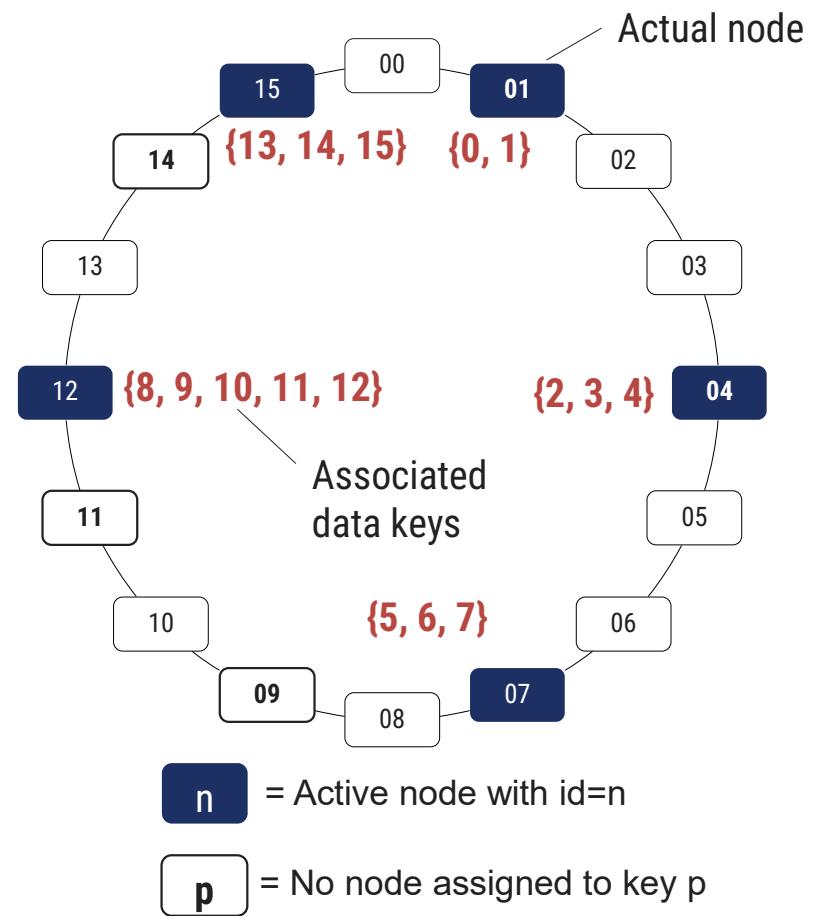
- ▶ Multi-tiered architectures can be considered as vertical distribution
 - Placing logically different components on different machines
- ▶ An alternative is horizontal distribution (peer-to-peer systems)
 - A collection of logically equivalent parts
 - Each part operates on its own share of the complete 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

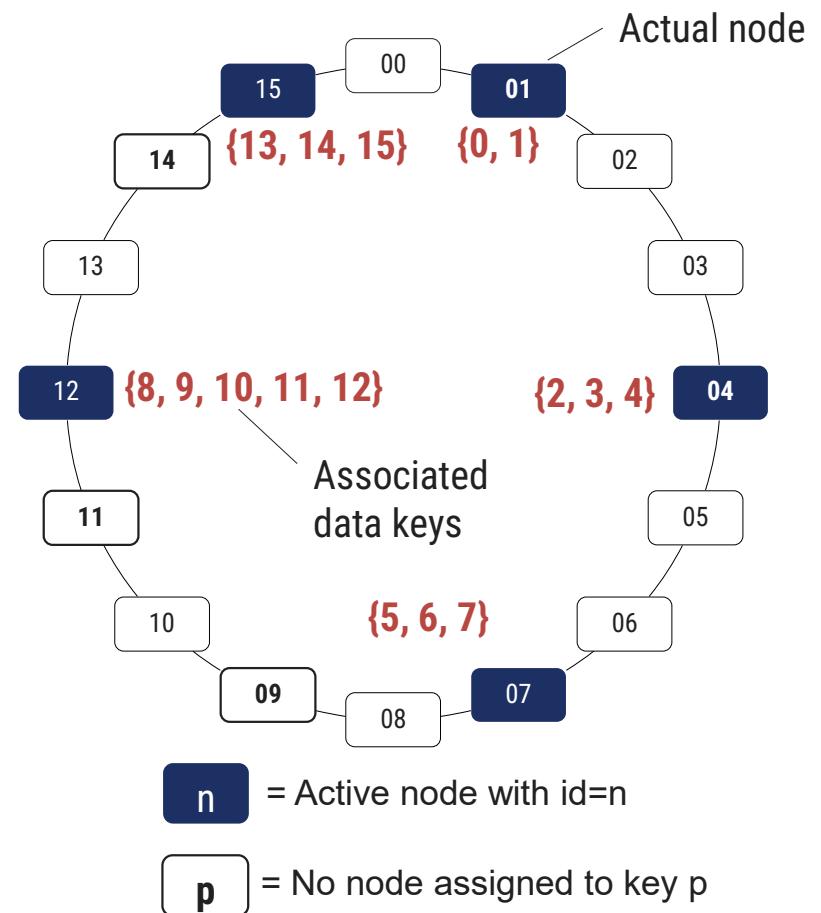
- Generate the node's random identifier, id, using the distributed hash function
- Use the lookup function to locate $\text{succ}(id)$
- Contact $\text{succ}(id)$ and its predecessor to insert self into ring.
- Assume data items from $\text{succ}(id)$

▶ Leave (normally)

- Notify predecessor & successor;
- Shift data to $\text{succ}(id)$

▶ Leave (due to failure)

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

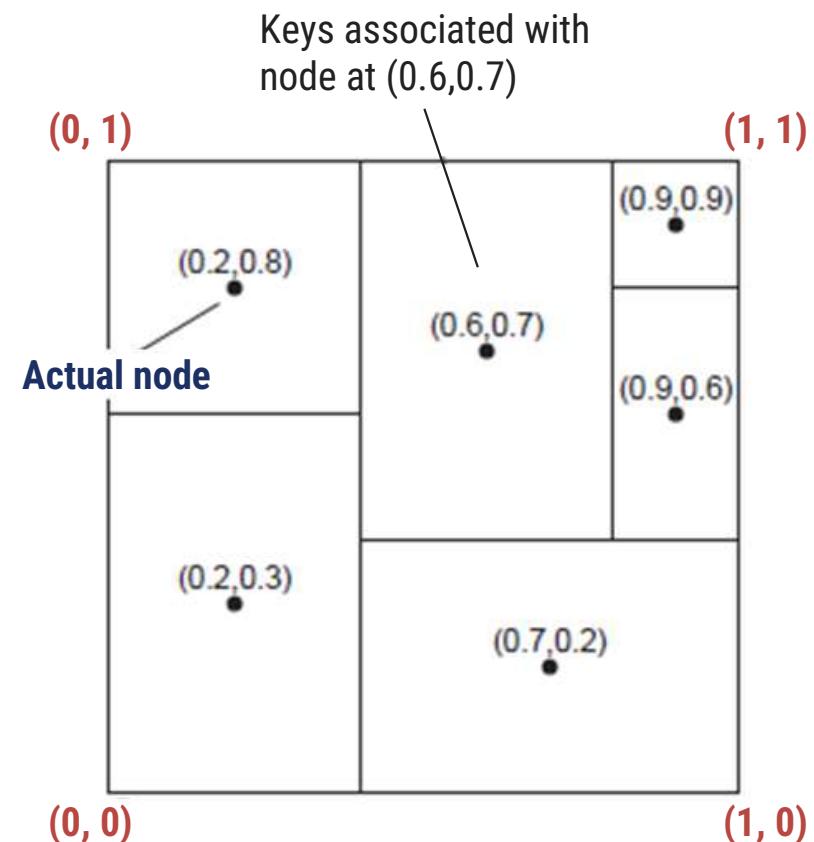


Content Addressable Network (CAN)

► 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**

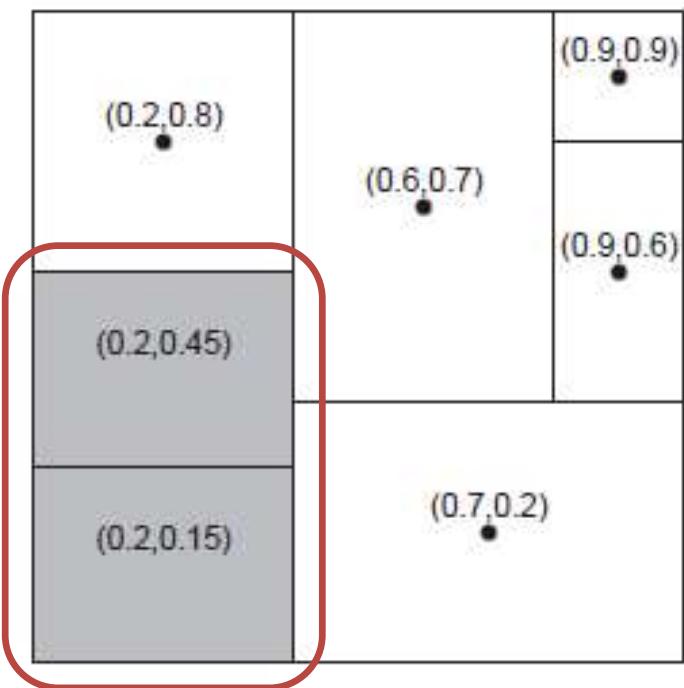
- Two-dimensional space $[0,1] \times [0,1]$ is divided among six nodes
- Each node has an associated region
- Every data item in CAN will be assigned a unique point in this space, after which it is also clear which node is responsible for that data



Content Addressable Network (CAN)

Joining CAN

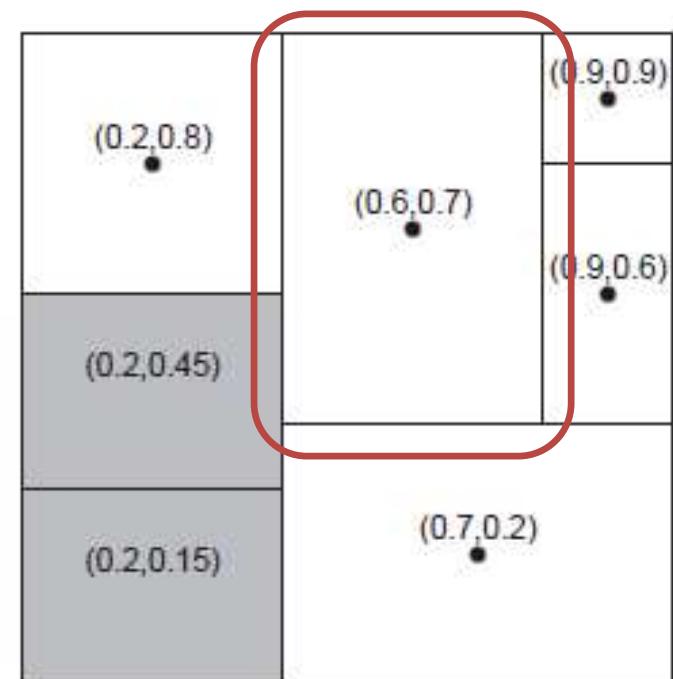
- When a node P wants to join a CAN system, it picks an arbitrary point from the coordinate space and subsequently looks up the node Q in whose region that point falls.
- Node Q then splits its region into two halves and one half is assigned to the node P.
- Nodes keep track of their neighbors, that 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 P.
- 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

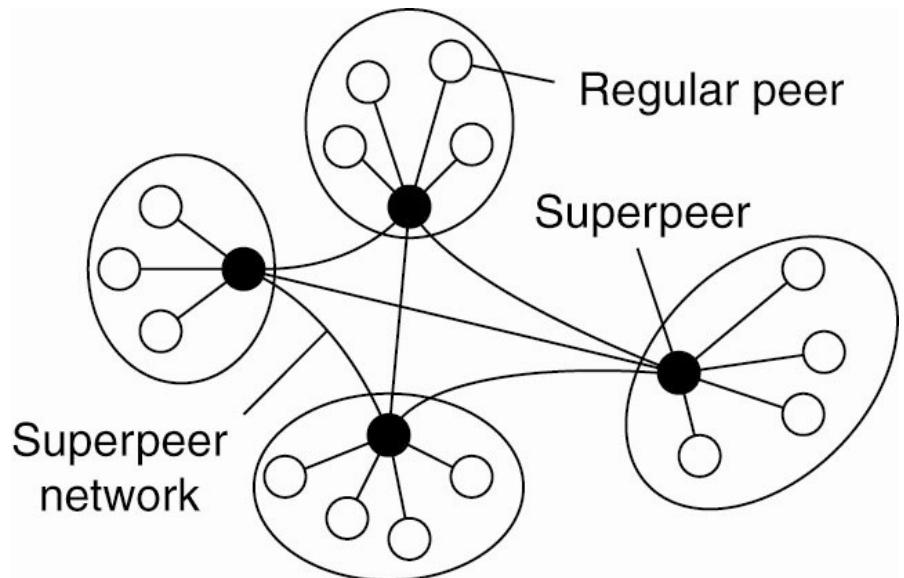
- ▶ Nodes act as both client and server; interaction is symmetric
- ▶ Each node acts as a server for part of the total system data
- ▶ Overlay networks **connect nodes in the P2P system.**
- ▶ An overlay network can be thought of as a computer network on top of another network. All nodes in an overlay 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

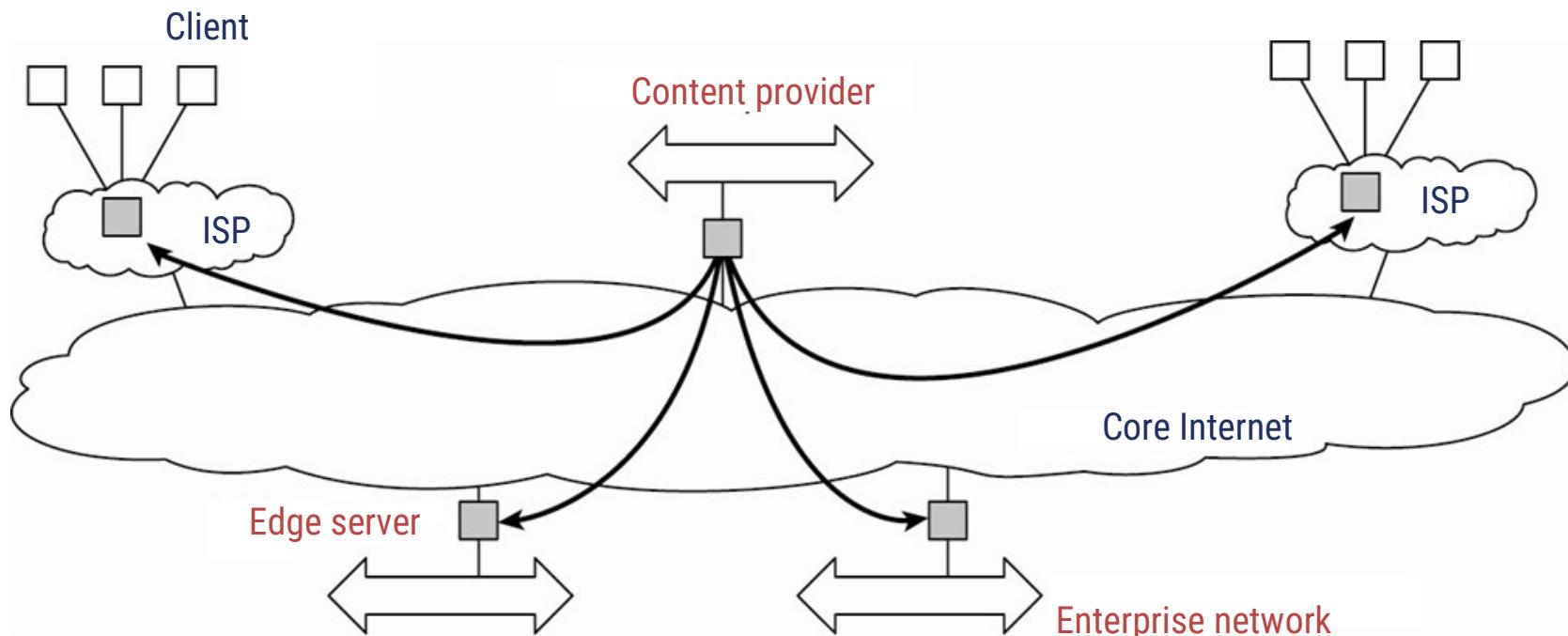
- ▶ This is quite challenging in unstructured P2P systems
 - Assume a data item is randomly placed
- ▶ Solution 1: Flood the network with a search query
- ▶ Solution 2: A randomized algorithm
 - Let us first assume that
 - Each node knows the IDs of k other randomly selected nodes
 - The ID of the hosting 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.



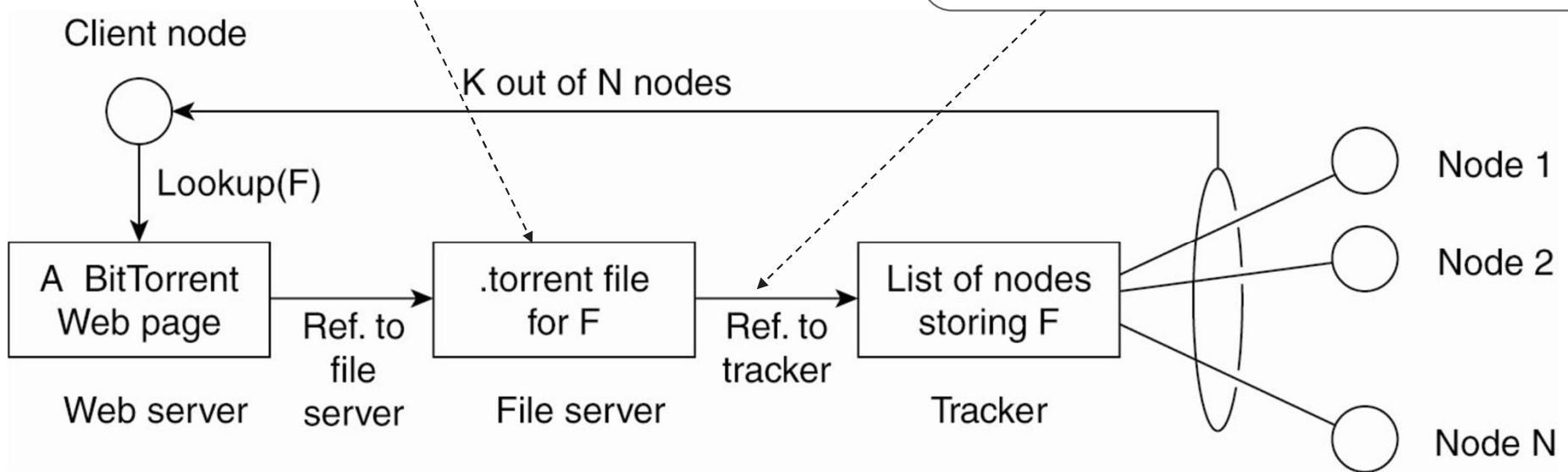
Collaborative Distributed Systems

- ▶ 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

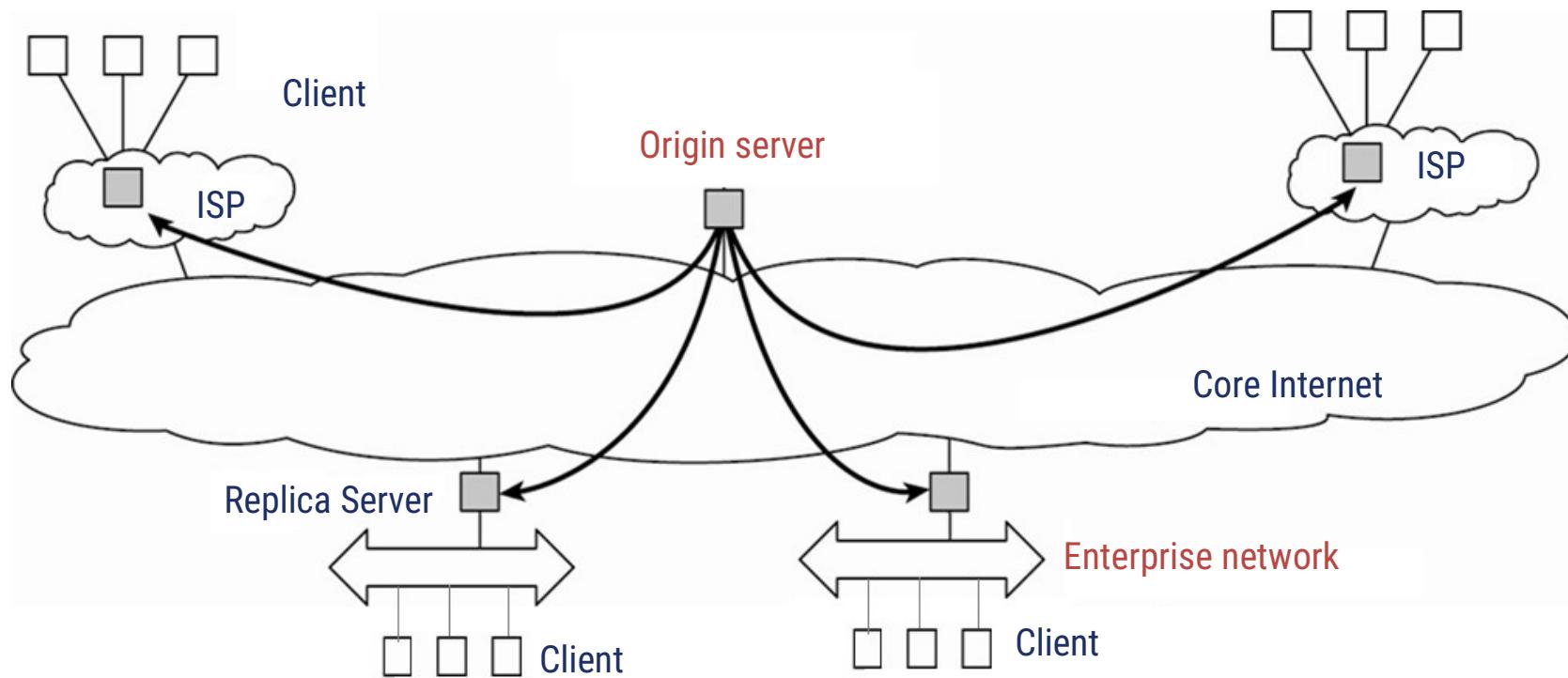


The principal working of BitTorrent (Pouwelse et al. 2004).

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



Architectures Versus Middleware

- ▶ Middleware forms a layer between applications and distributed platforms
- ▶ Provide a degree of distribution transparency, hiding the distribution of data, processing, and control from applications.
- ▶ Many middleware follows a specific architecture style
 - Object-based style, event-based style
 - **Benefits:** simpler to design application
 - **Limitations:** the solution may not be **optimal**
- ▶ In many cases, distributed systems/applications are developed according to a specific architectural style .
- ▶ Middleware systems follow a specific architectural style
 - Object-based architectural style - CORBA
 - Event-based architectural style - TIB/Rendezvous
- ▶ **Interceptors :** Intercept the usual flow of control when invoking a remote object

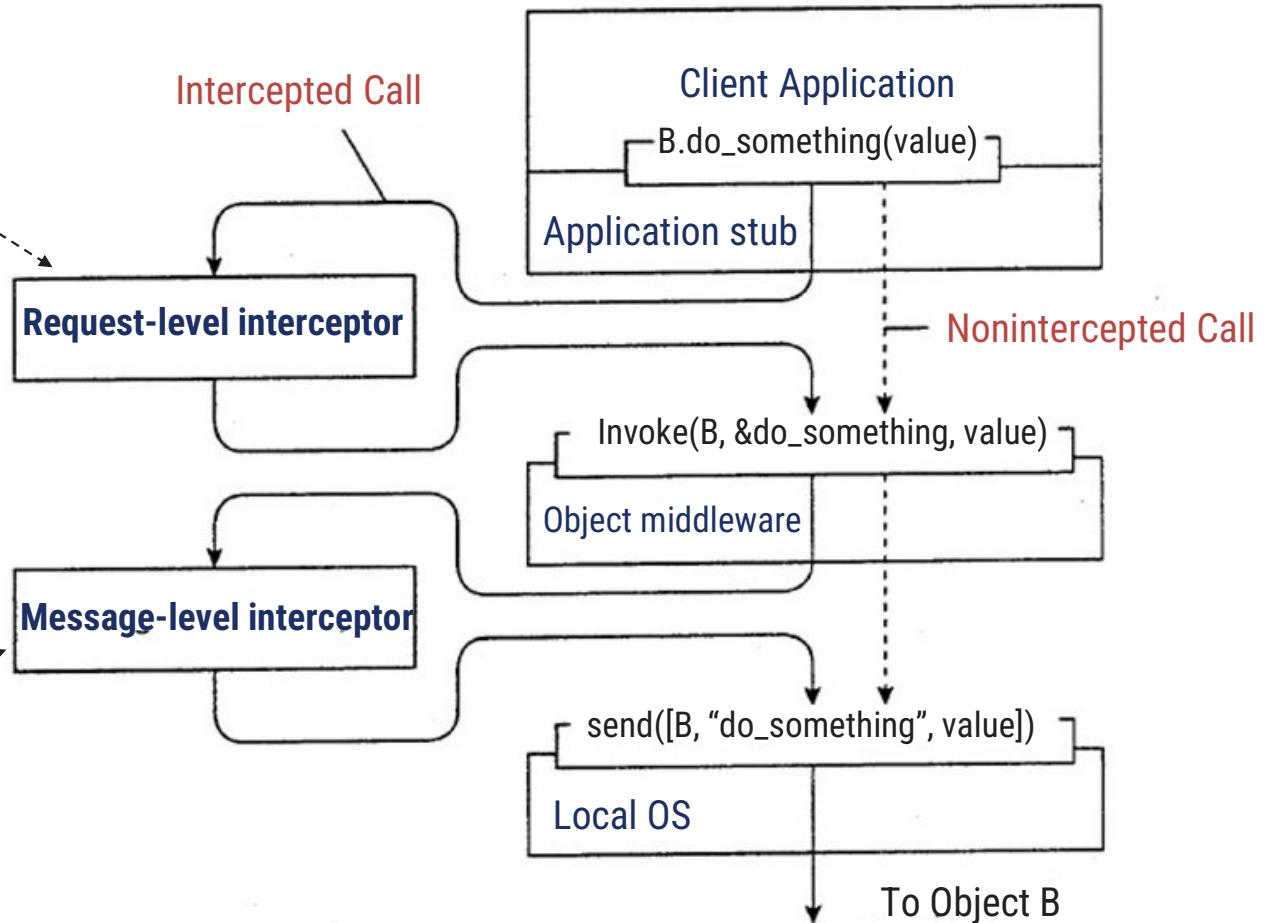
Supporting Technology: Interceptors

- ▶ An Interceptor is a software that Breaks the usual flow of control and Allows other (application specific) code to be executed
- ▶ It makes middleware more adaptable to application requirements and changing environment
- ▶ Interceptors are good for
 - Providing transparent replication and
 - Improving performance

Interceptors

May want to send to many other B's (i.e. replicated)

May want to break a large message for better performance



General Approaches for Adaptability (Adaptive Middleware)

► Separation of concerns

- Modularizing the system and separate security from functionality
- However, the problem is that a lot of things you cannot easily separate, e.g., security

► Computational reflection

- Ability to inspect itself, and if necessary, adapt its behavior
- Reflective middleware has yet to proof itself as a powerful tool to manage the complexity of distribute systems

► Component-based design (stand-alone)

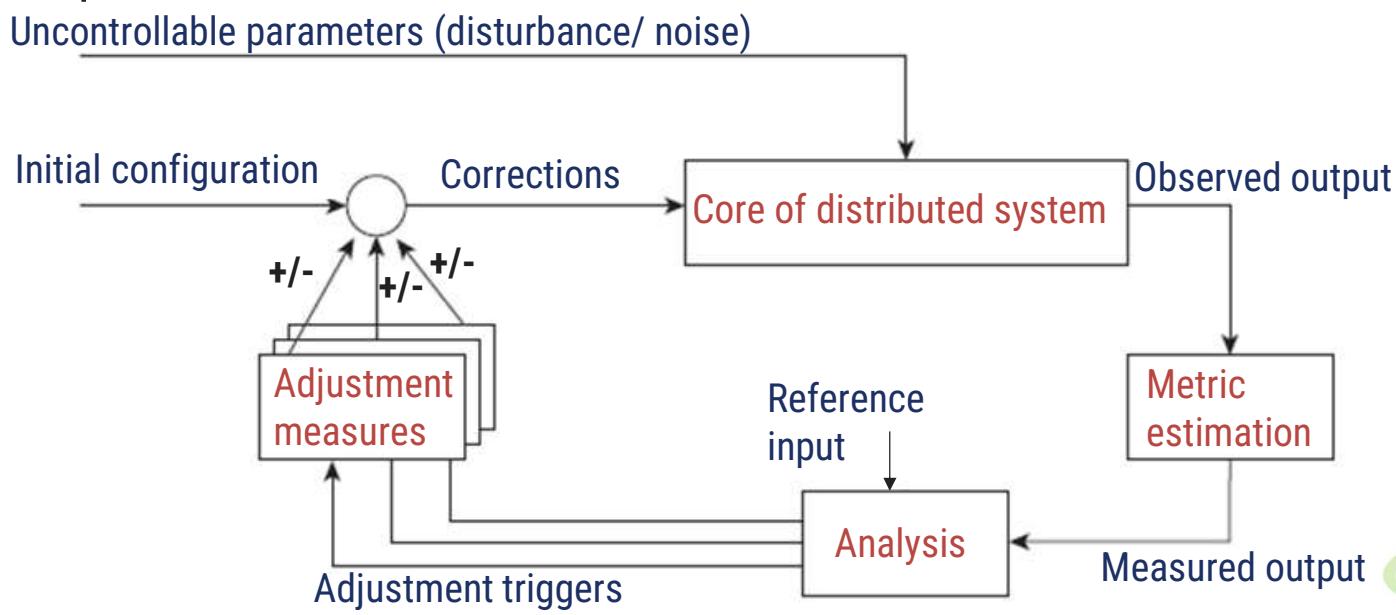
- However, components are less independent than one may think
- Replacement of one component may have huge impact on others

Self-Management in Distributed Systems

- ▶ Distributed systems are often required to adapt to environmental changes by
 - Switching policies for allocation resources
- ▶ The algorithms to make the changes are often already in the components
 - But the challenge is how to make such change without human intervention
- ▶ A strong interplay between software architectures and system architectures
 - Organize components in a way that monitoring and adjustment can be done easily
 - Decide where the processes to be executed to do the adaption

The Feedback Control Model

- ▶ Allow automatic adaption to changes by means of one or more feedback control loops
 - self-managing
 - self-healing
 - self-configuration
 - self-optimization, etc

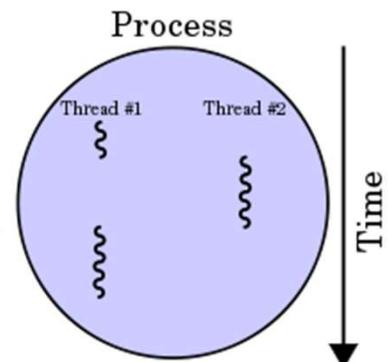


Introduction to Threads

- ▶ **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

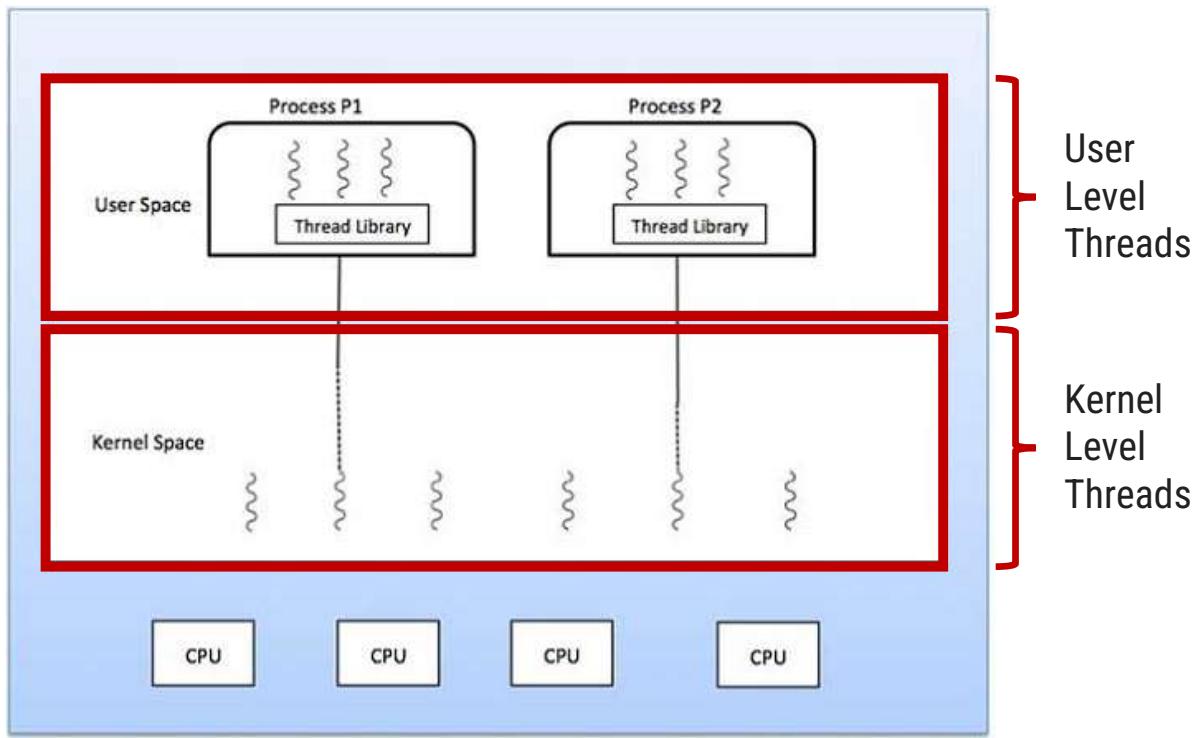
- Like processes threads **share CPU** and only **one thread is running at a time**.
- Like processes threads within a process **execute sequentially**.
- Like processes thread can **create children**.
- Like a traditional process, a thread can be in any one of several states: **running, blocked, ready 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

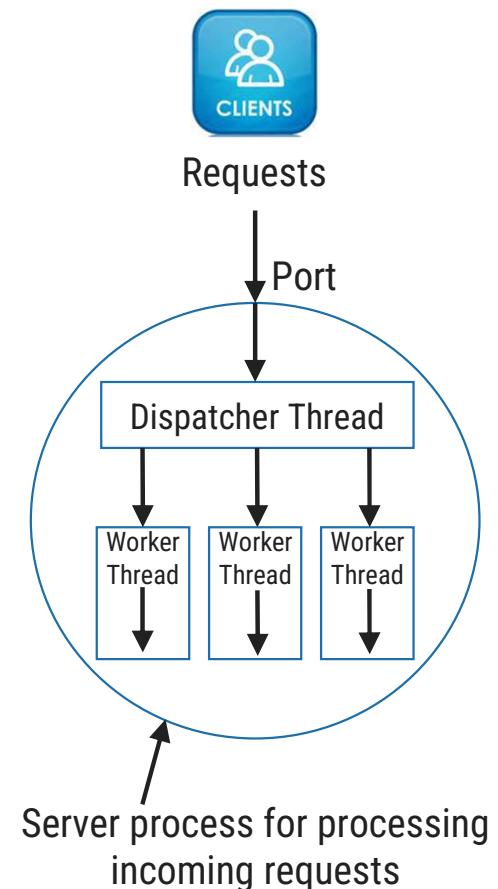
USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	Kernel threads are implemented by OS.
OS doesn't recognize user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complex.
Context switch time is less.	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 : Window 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

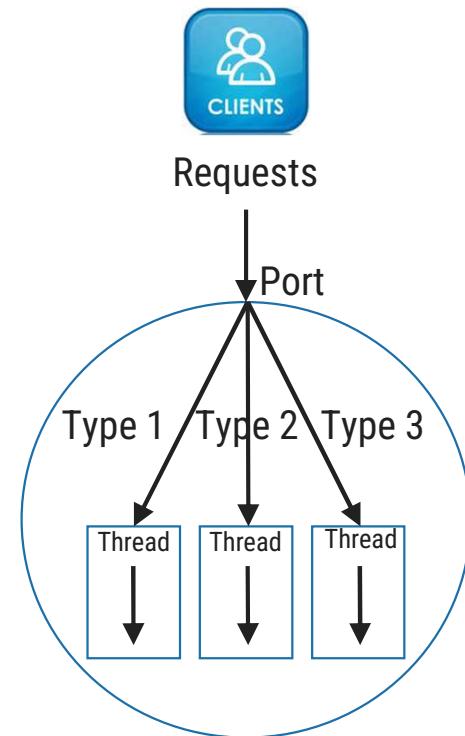
Dispatcher/Worker Model

- ▶ 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.



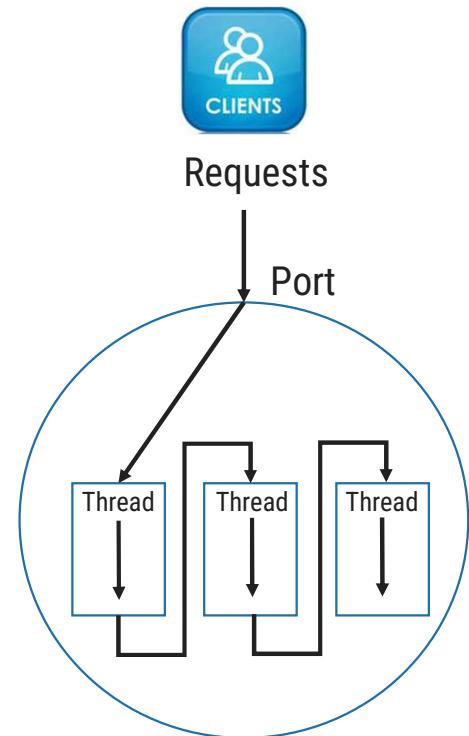
Team Model

- ▶ 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.



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:

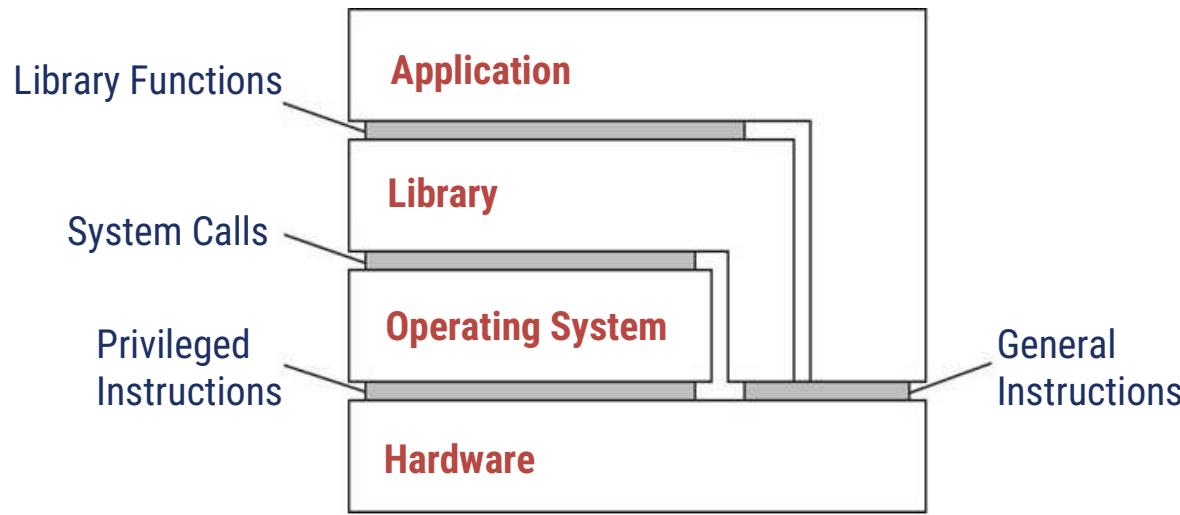
- It involves the simulation of virtual machines.
- Platform virtualization is performed on a given hardware platform by "host" software (a control program), which creates a simulated computer environment (a virtual machine) for its "guest" software.
- The "guest" software, which is often itself a complete 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.

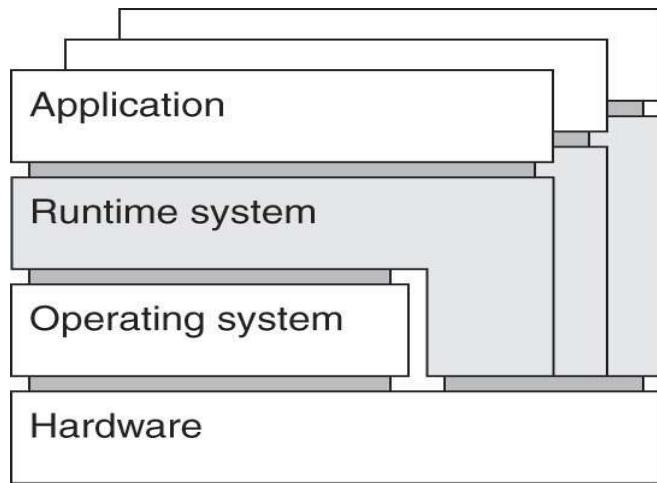
Architectures of Virtual Machines

- ▶ Four distinct levels of interfaces to computers the behavior of which that virtualization can mimic
 - Unprivileged machine instructions: available to any program
 - Privileged instructions: hardware interface for the OS/other privileged software
 - System calls: interface to the operating system for applications & library functions
 - API: An OS interface through library function calls from applications.

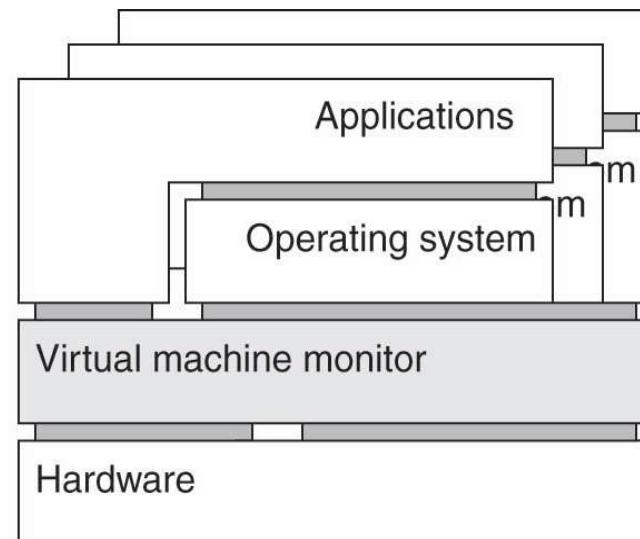


Two Ways to Virtualize

Process Virtual Machine



Virtual Machine Monitor



- program is compiled to intermediate code, executed by a runtime system

- software layer mimics the instruction set; supports an OS and its applications

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.

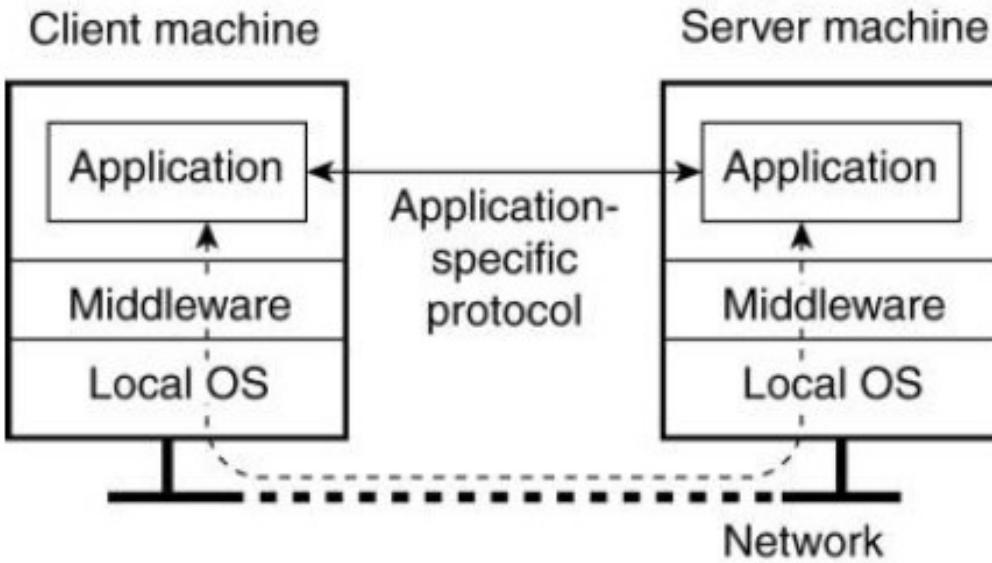
Networked User Interfaces

Two ways to support client-server interaction:

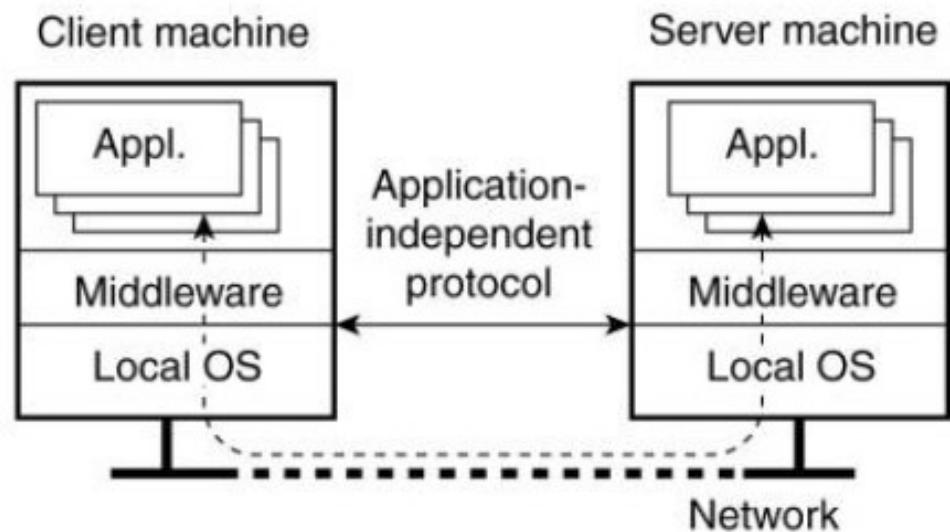
1. For each remote service - the client machine will have a separate counterpart that can contact the service over the network.
 - Example: an agenda running on a user's PDA that needs to synchronize with a remote, possibly shared agenda.
 - In this case, an application-level protocol will handle the 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 as shown in

Networked User Interfaces

Networked application with its own protocol

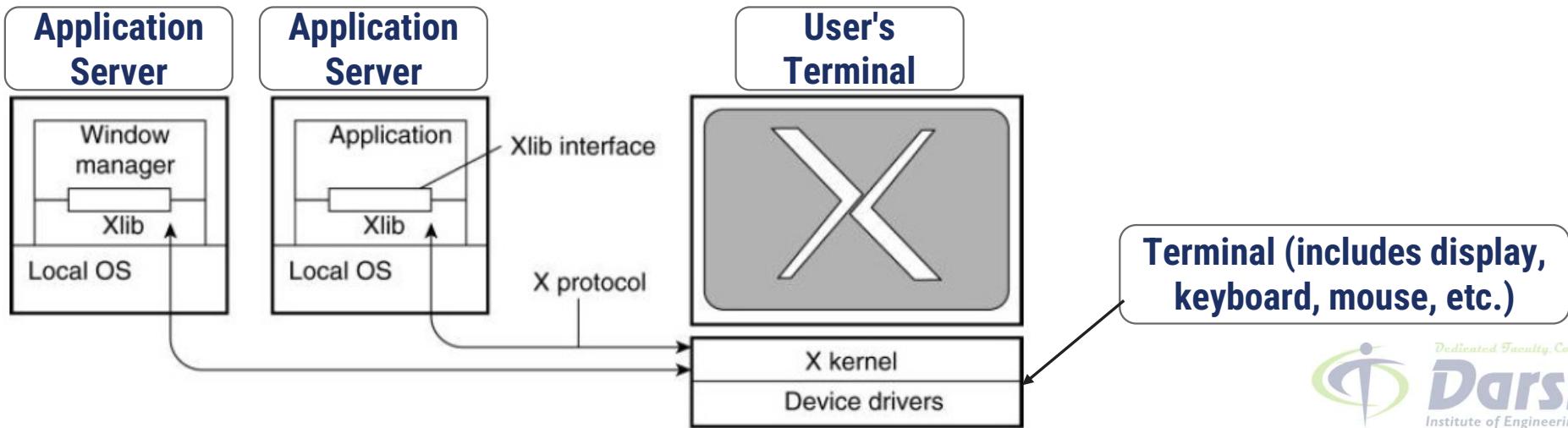


General solution to allow access to remote applications.



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.



Client-Side Software for Distribution Transparency

▶ Access transparency

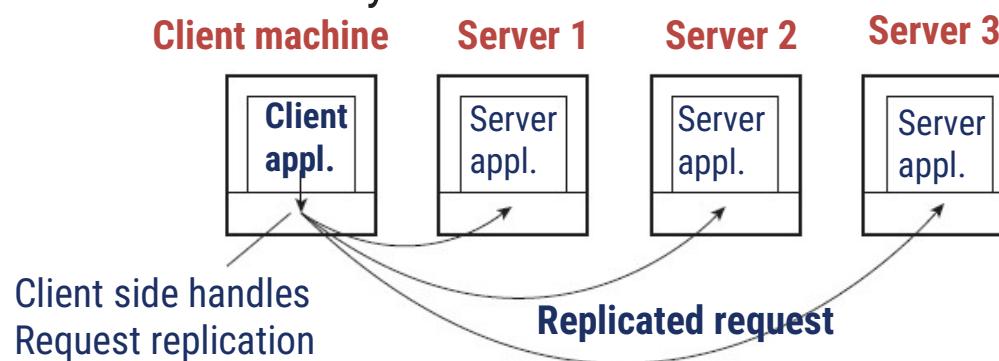
- Handled through client-side stubs for RPCs: provides
- Same interface as at the server, hides different machine architectures

▶ 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

▶ Persistence transparency

- Completely handled at the server.

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

- 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.

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.

Stateful server

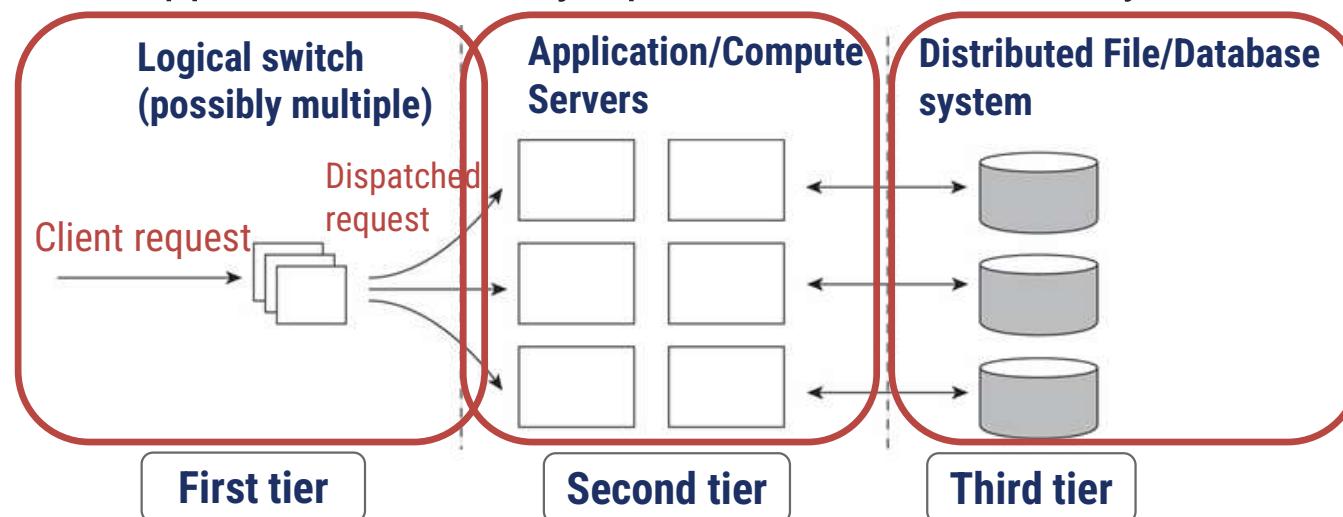
- ▶ A stateful server remembers client data (state) from one request to the next.
- ▶ Information needs to be explicitly deleted by the server.
- ▶ Example:
 - A file server that allows a client to keep a local copy of a file, even for performing update operations.
 - The server maintains a table containing (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



Code Migration

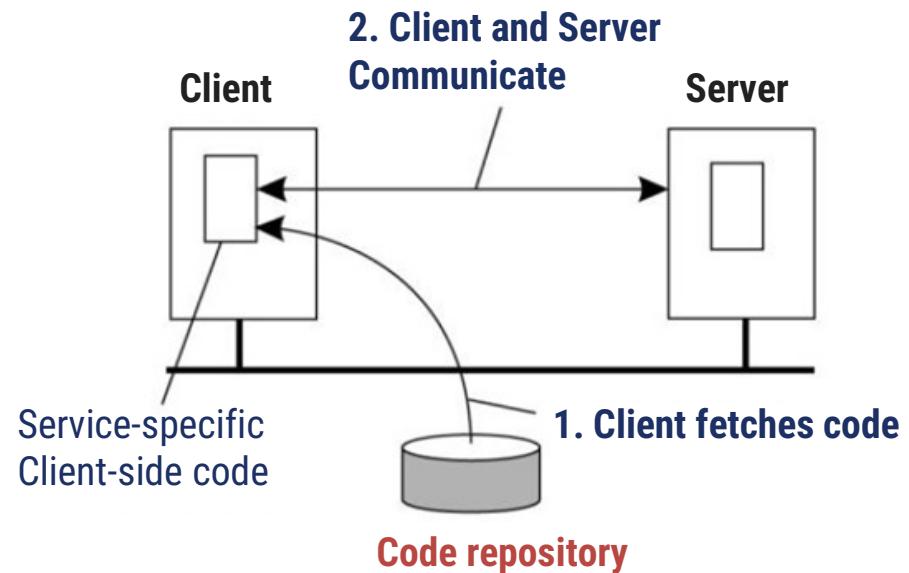
- ▶ Traditionally, communication in distributed systems is concerned with exchanging data between processes.
- ▶ Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target
- ▶ process migration in which an entire 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 only the code segment, along with perhaps some initialization data.	In contrast to weak mobility, in systems that support strong mobility the execution segment can be transferred as 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 execution from the beginning	D'Agents.
Benefit	Simplicity	Much more general than weak mobility

Migration Initiation

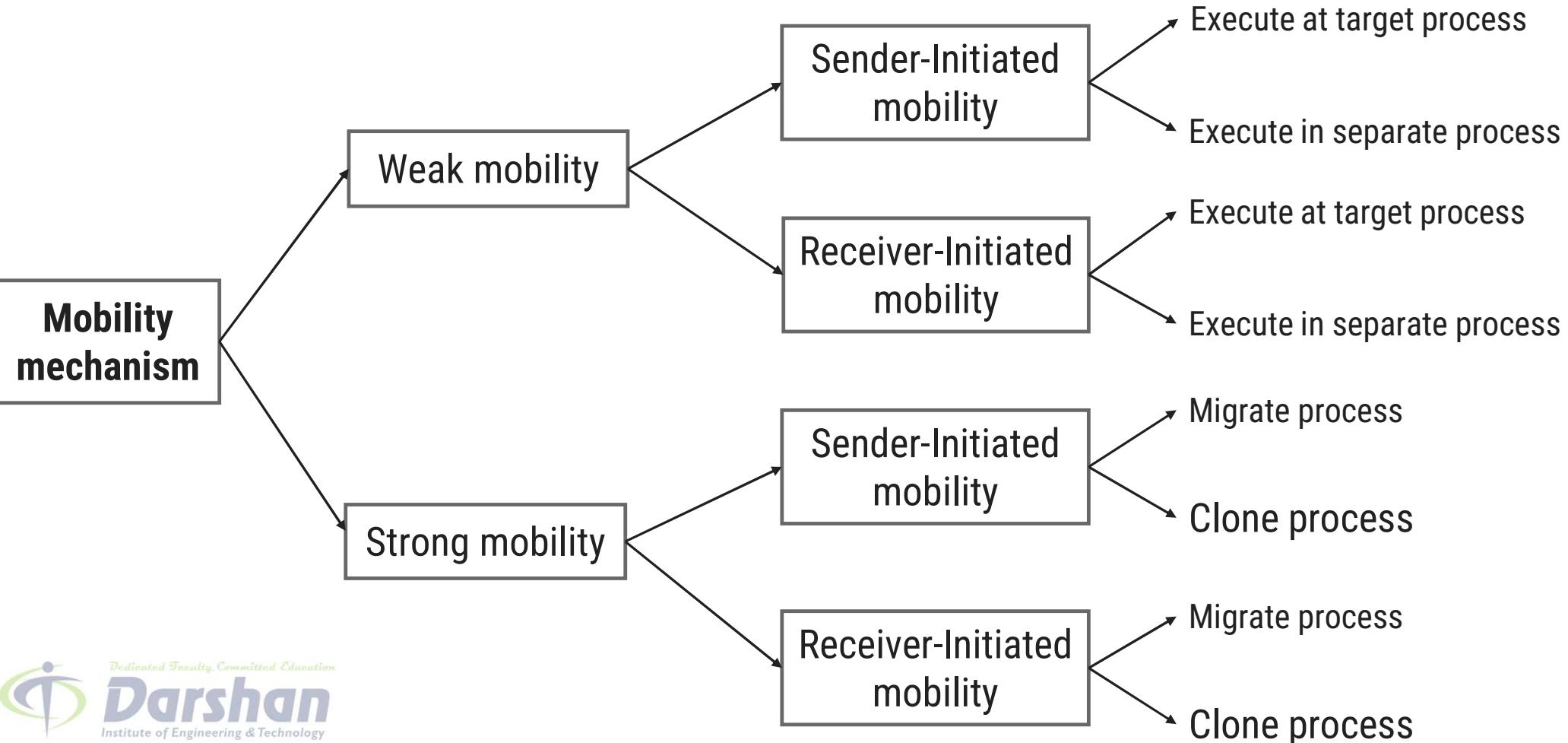
Sender-Initiated Migration

- ▶ 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
	CP (or MV, GR)	GR (or CP)	GR
	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

Communication in a Distributed System

- ▶ In a distributed system, processes run on different machines.
- ▶ Processes can only exchange information through message passing.
 - Harder to program than shared memory communication
- ▶ Successful distributed systems depend on communication models that hide or simplify message passing
- ▶ Communication in distributed systems is always based on low-level message passing as offered by the underlying network.

Layered Network Communication Protocols

Physical layer:

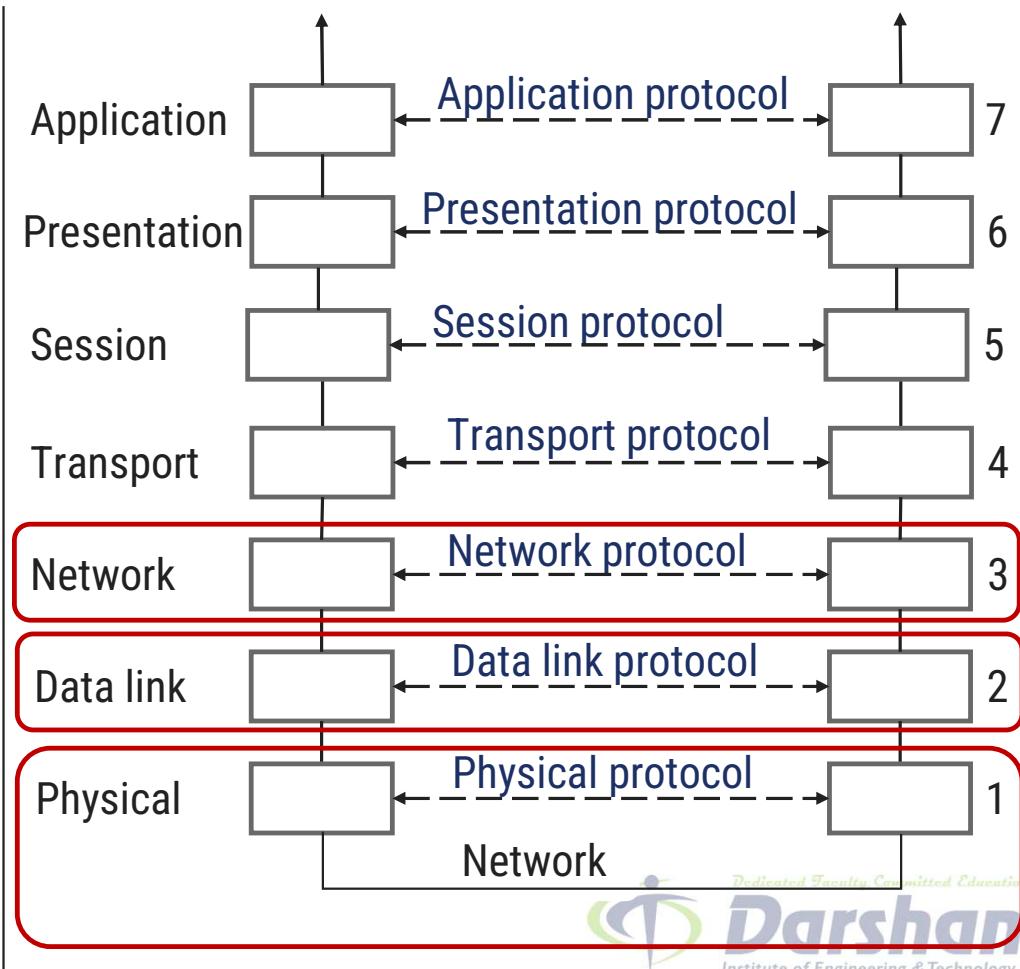
- ▶ Transmitting bits between sender and receiver
- ▶ Functions of a Physical layer: Line Configuration, Data Transmission, Topology, Signals

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



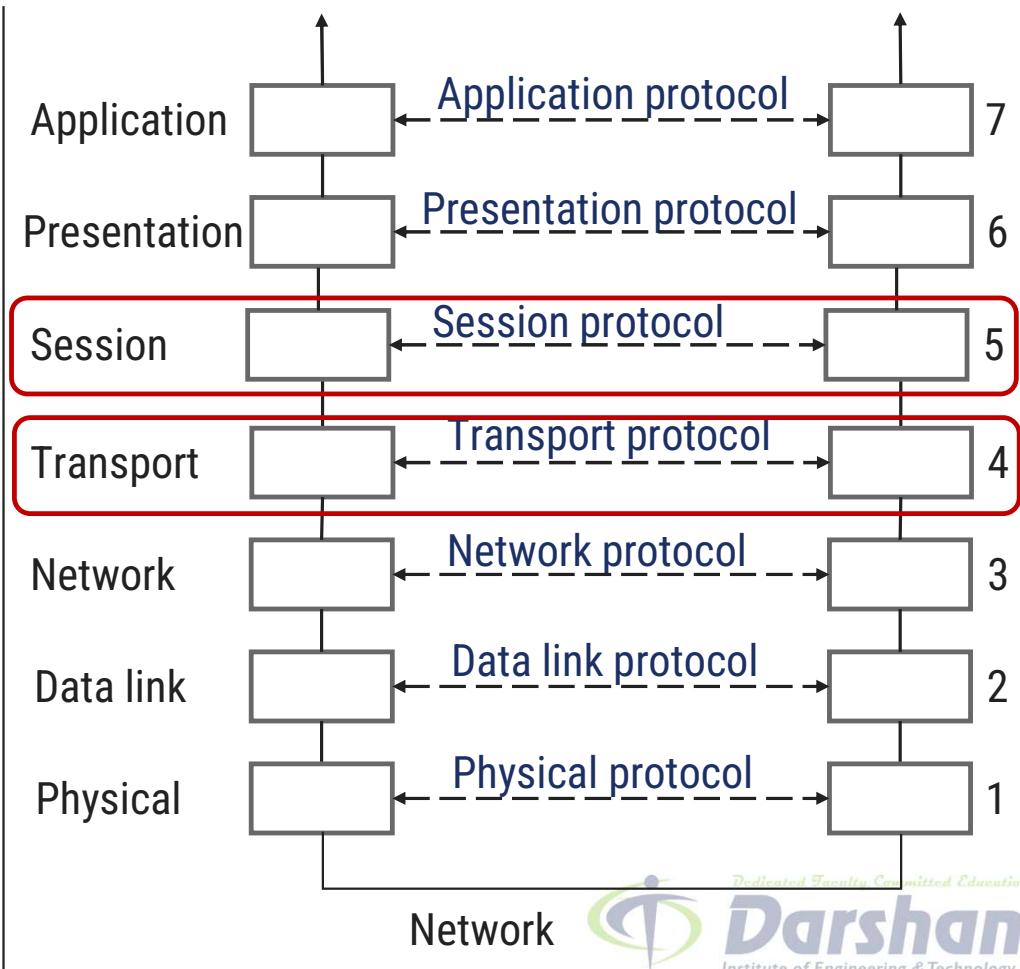
Layered Network Communication Protocols

Transport layer:

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

Session layer:

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



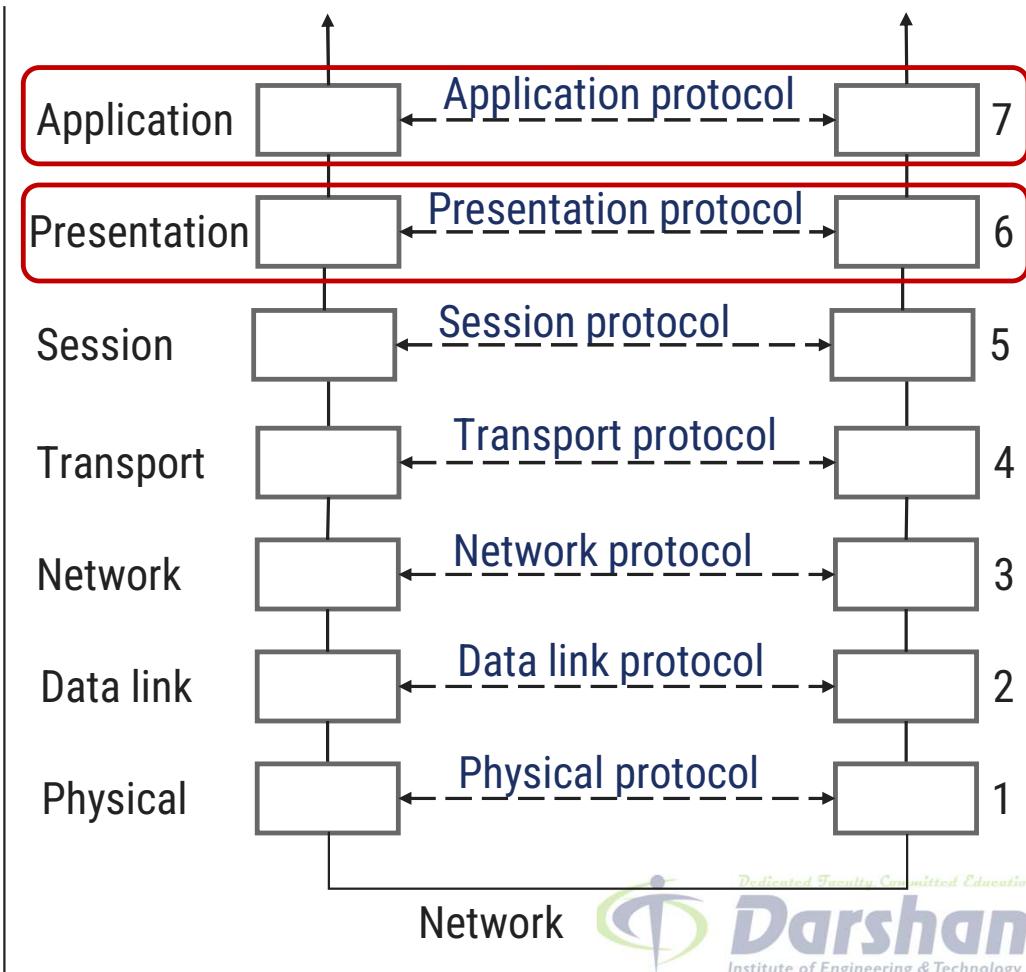
Layered Network Communication Protocols

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:

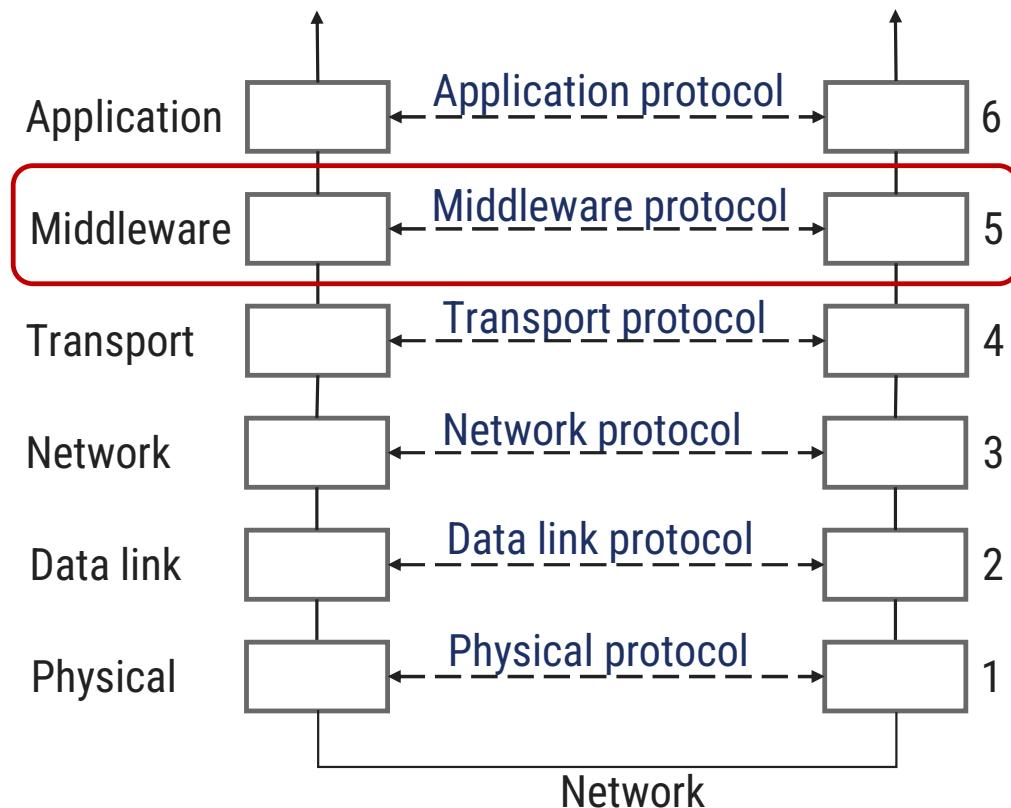
- ▶ Provides means for the user to access information on the network through an application
- ▶ It serves as a window for users and application processes to access network service.



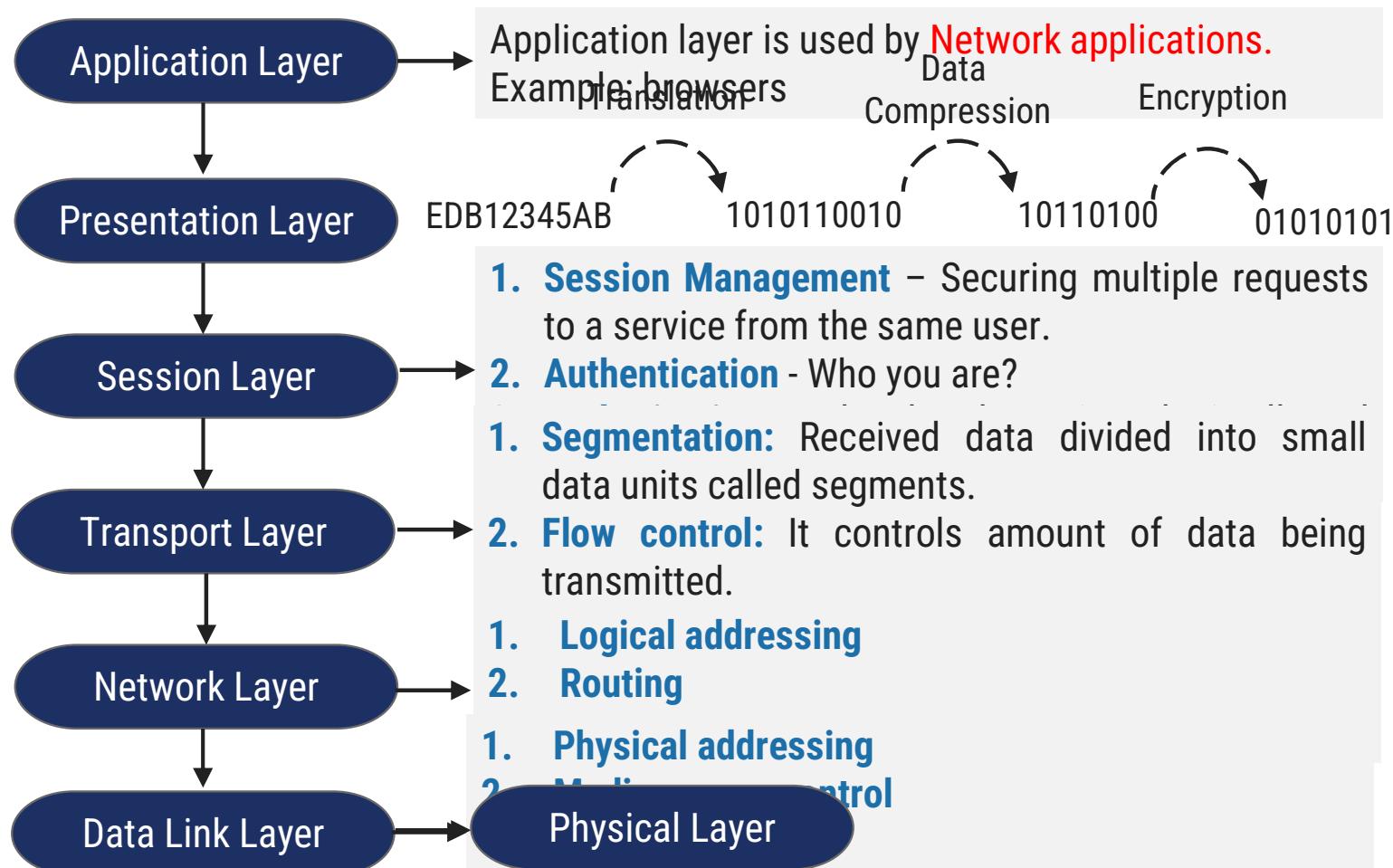
Middleware Layer

► **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



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.

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.

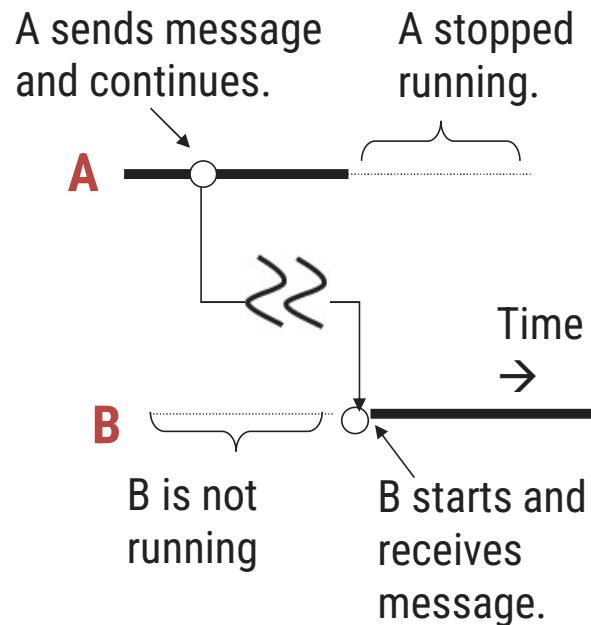
Asynchronous Communication

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

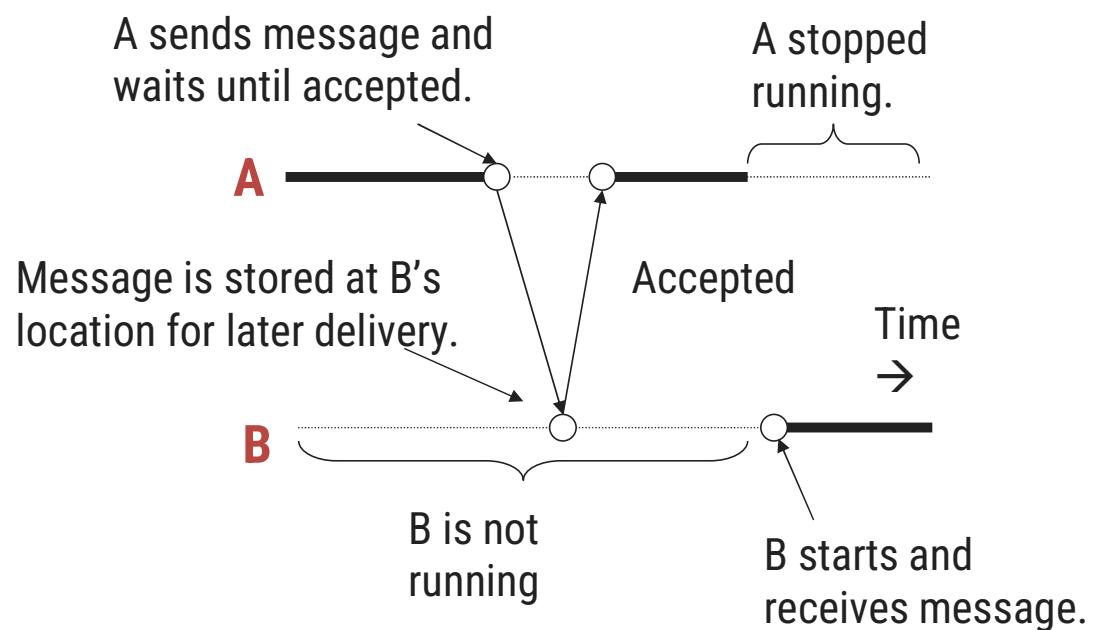
Distributed Communications Classifications

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



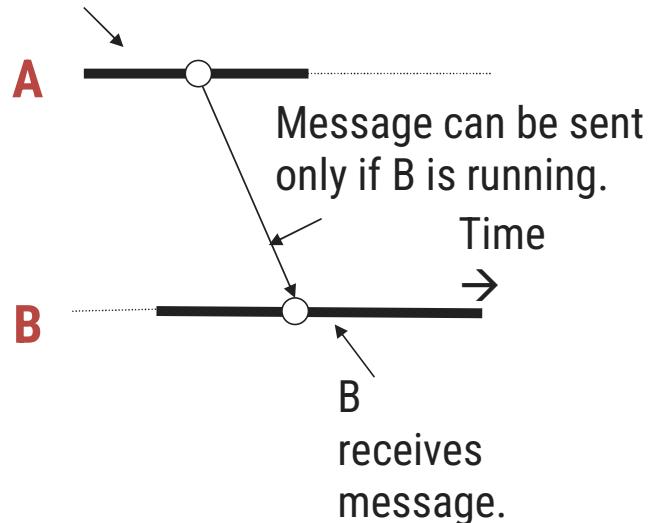
Persistent asynchronous communication



Persistent synchronous communication

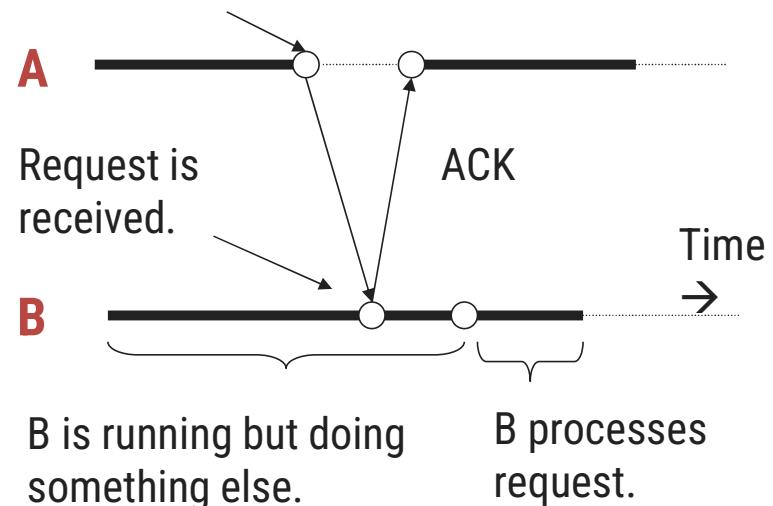
Distributed Communications Classifications

A sends message and continues.



Transient asynchronous communication

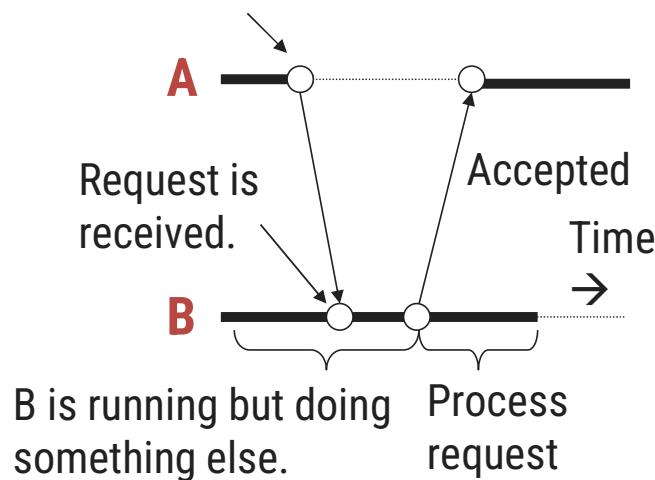
A sends message and waits until received.



Receipt-based synchronous communication

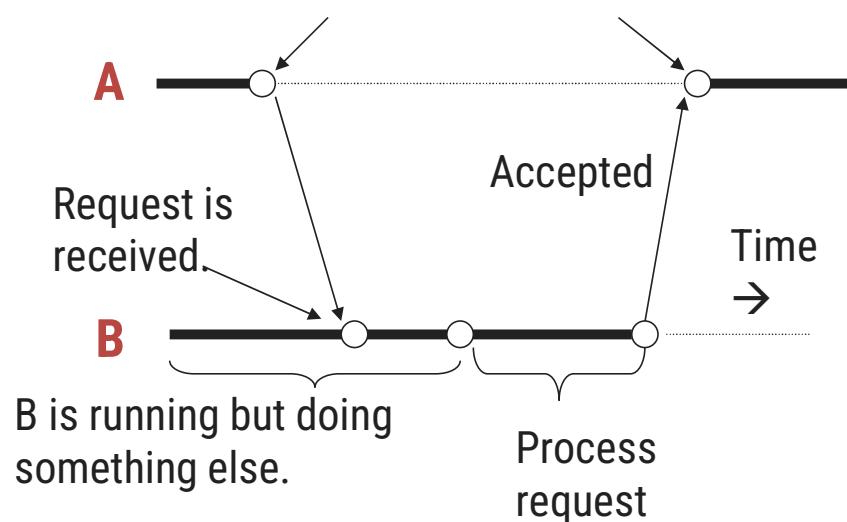
Distributed Communications Classifications

A sends request and waits until accepted.



Delivery-based synchronous communication

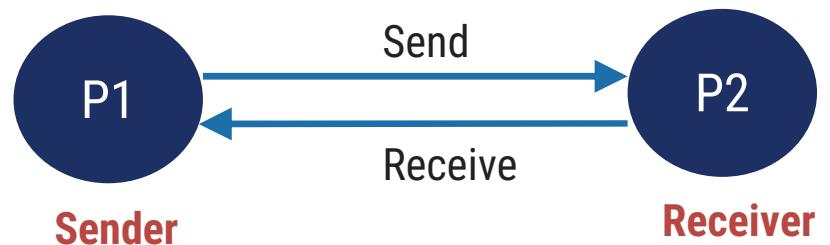
A sends request and waits for reply.



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 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

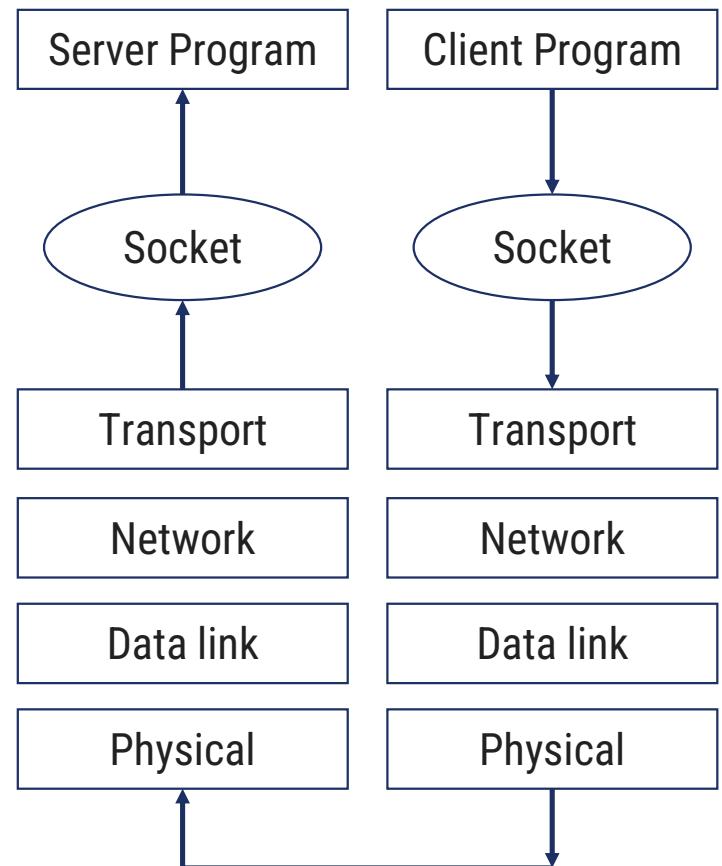
- ▶ Following are different types of packets transmitted across the network.
 - **REQ:** Request packet is used to **send the request** from the client to the server.
 - **Reply:** This message is used to **carry the result** from the server to the client.
 - **ACK:** Acknowledgement packet is used to 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)

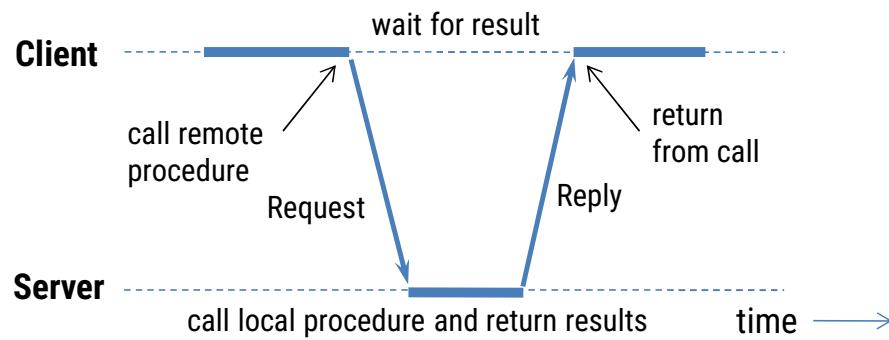
Socket

- ▶ 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.

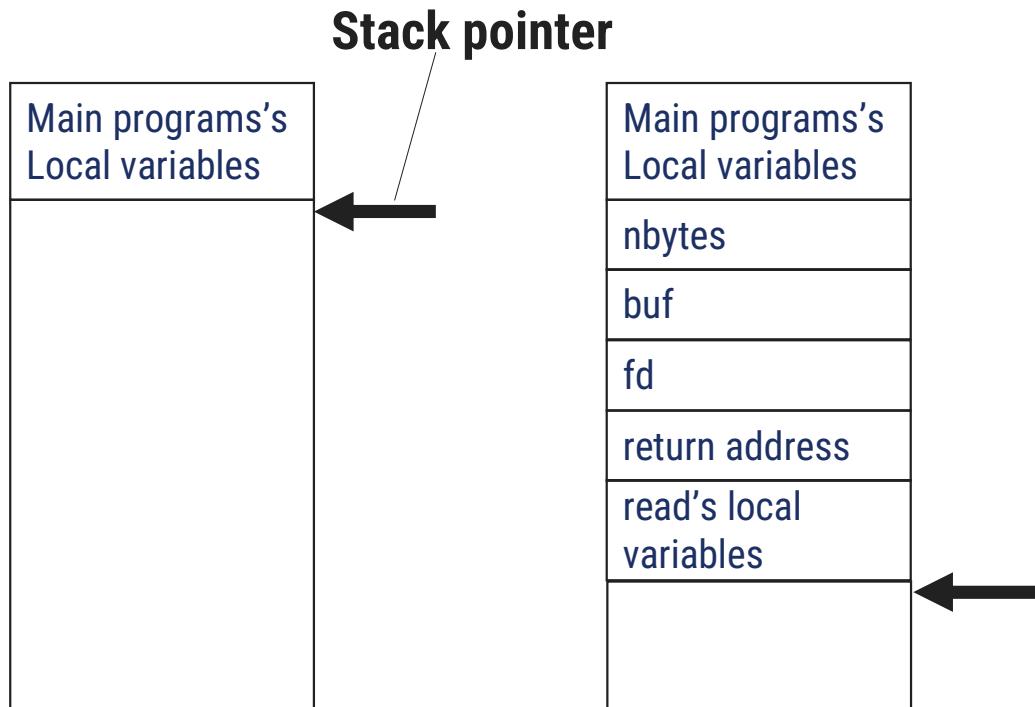
Conventional Procedure Call

- ▶ 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

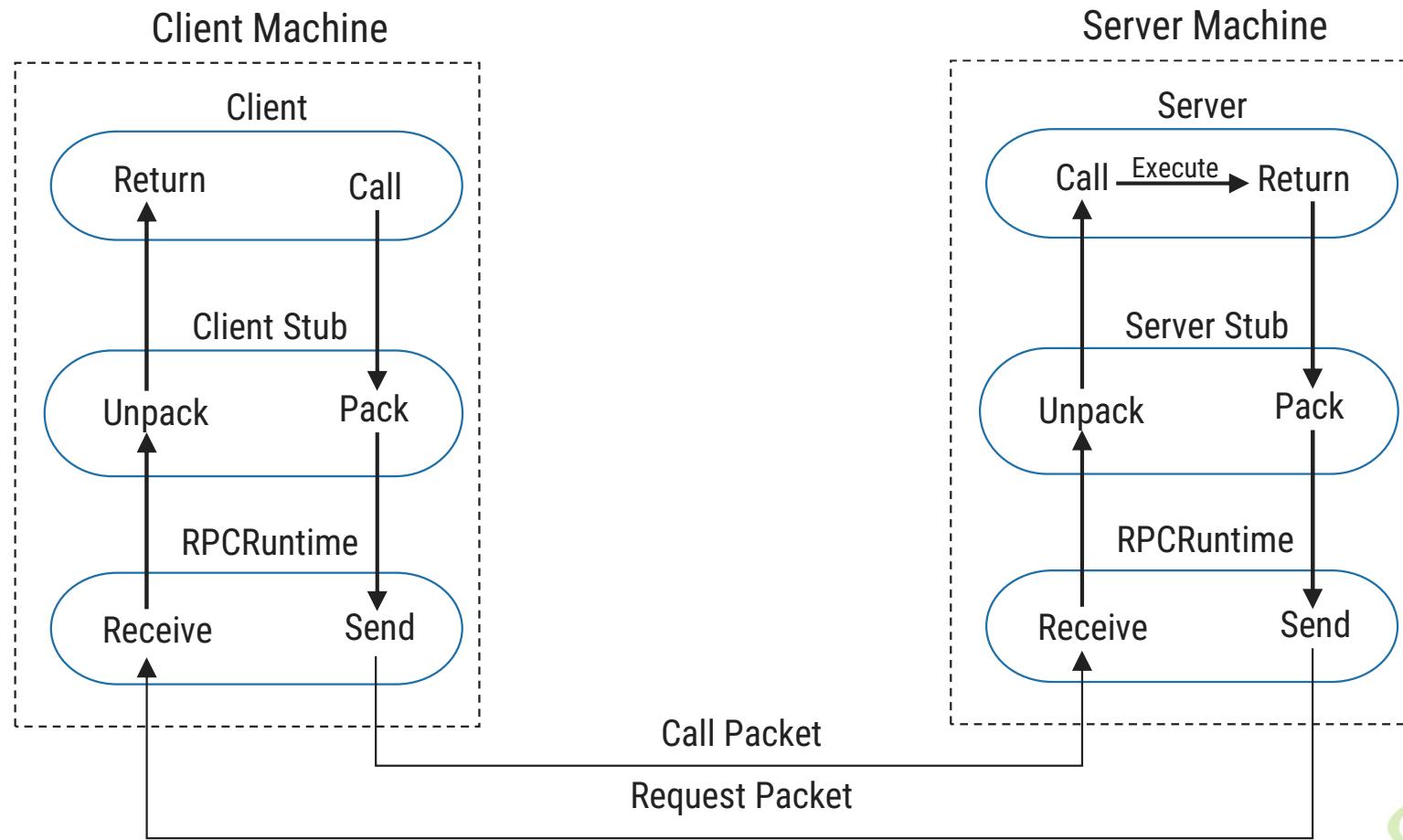
The Server stub

- ▶ 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

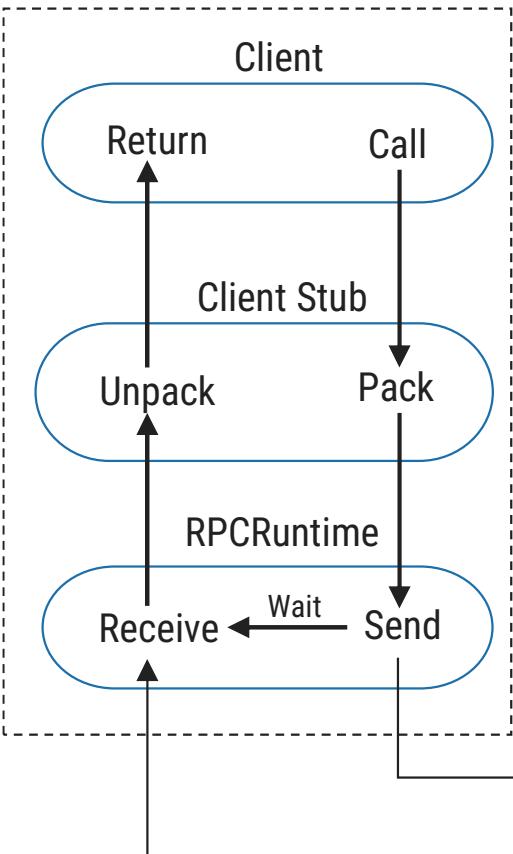
- ▶ It executes a appropriate procedure and returns the result from a server stub.

RPC Mechanism



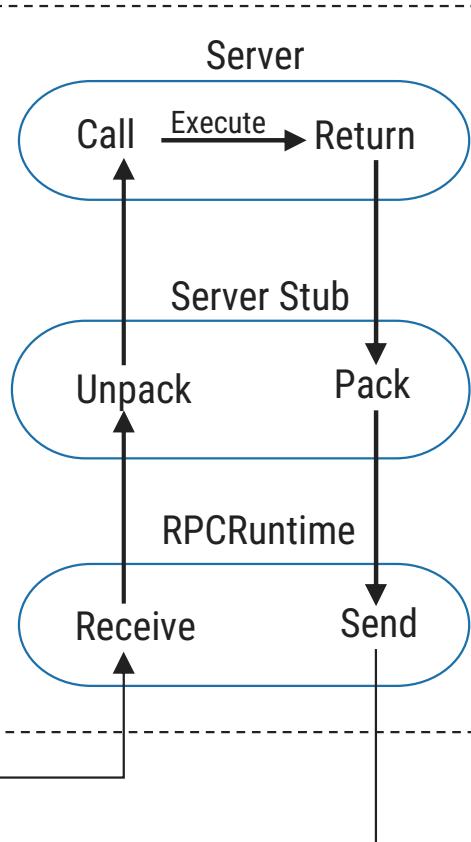
RPC Mechanism

Client Machine



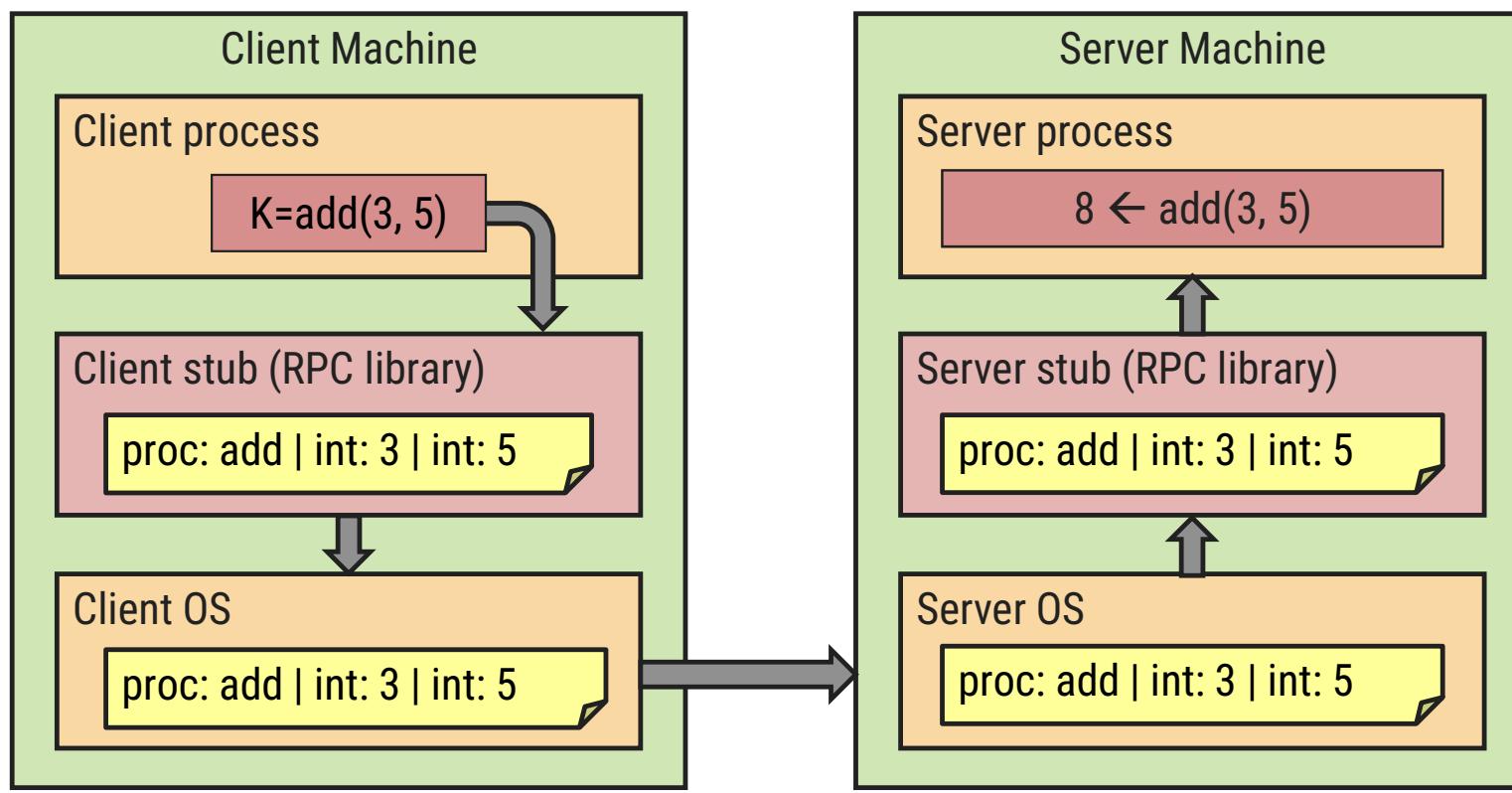
- ④ The client sends a call that contains the arguments for the method to all the runtime components standing by performing a local procedure call. The runtime performs the requested operations and returns the arguments.
- The client sends the arguments packed into the arguments message to the server.
- The server receives the arguments message and performs the requested operations.
- The server sends the results back to the client through the server stub, and has no idea about the client process which parameter it asked for how, because it has no idea about the other process continues its execution.

Client Machine



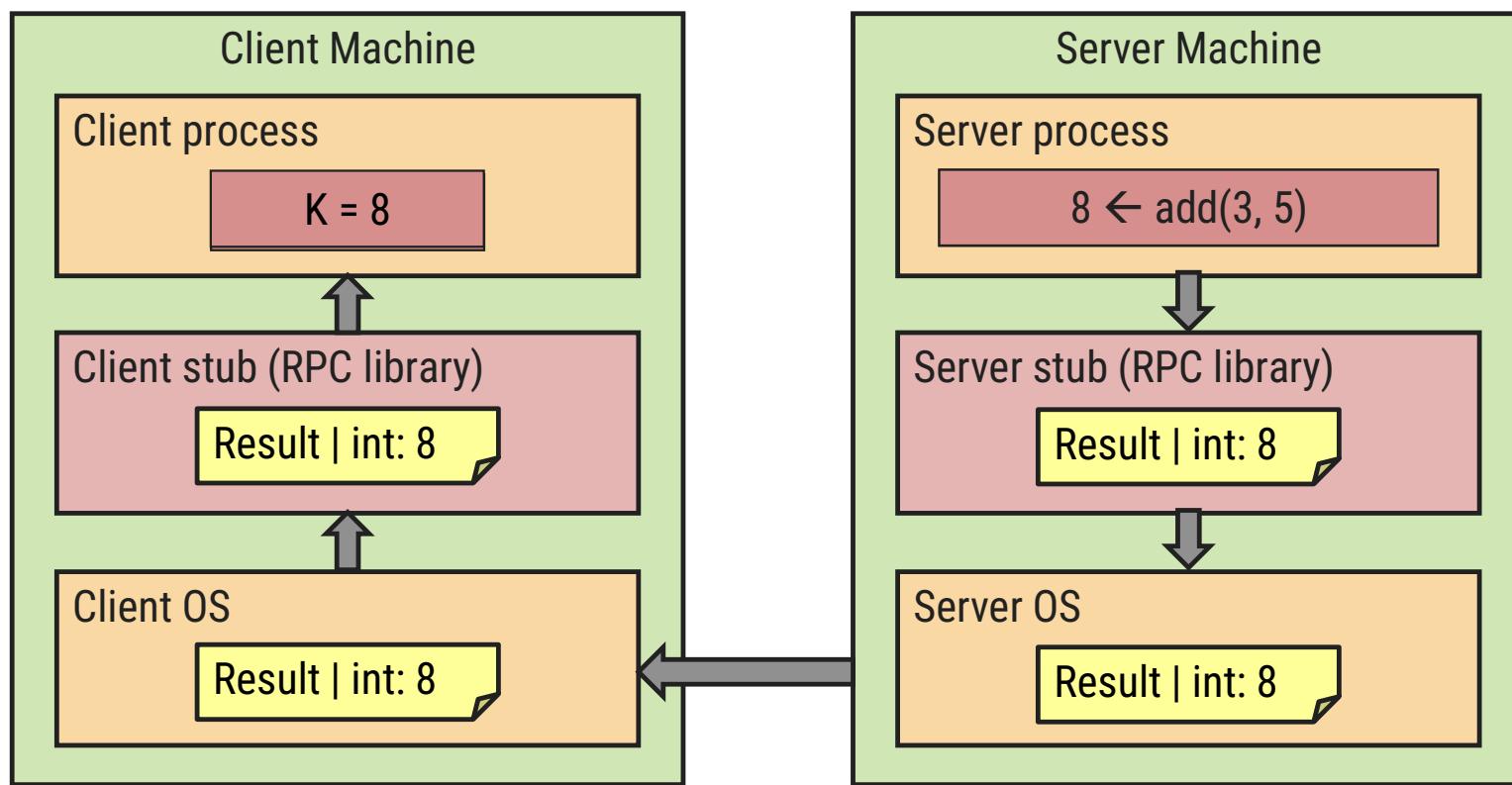
RPC Mechanism

3. Client OS sends the client stub to the server OS.



RPC Mechanism

Q. State's OS generates message to the client DSS.



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

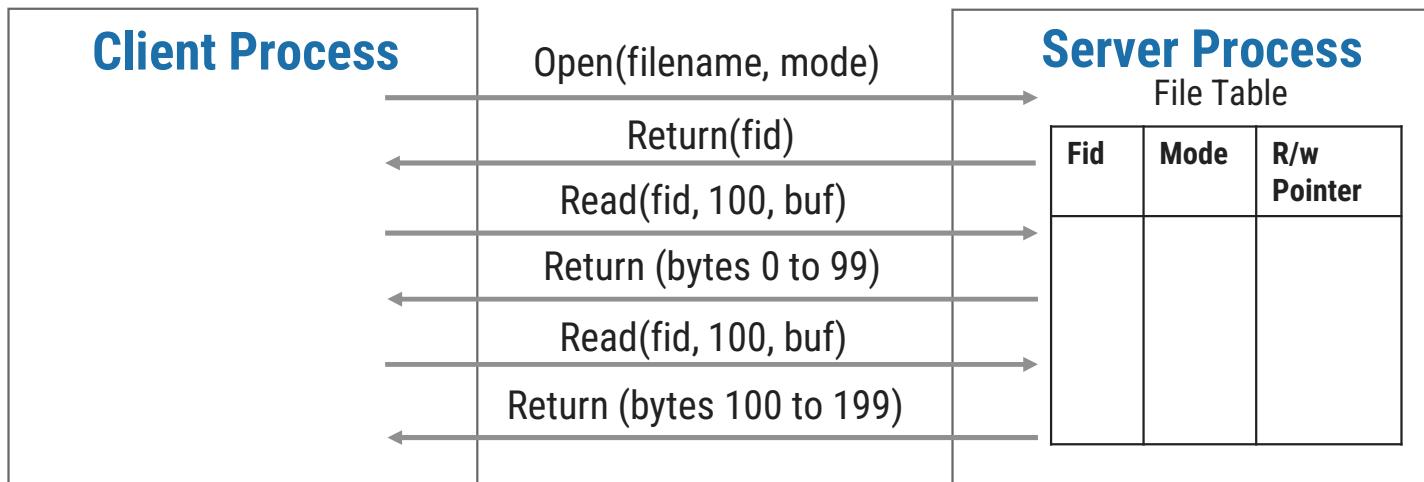
8. Messages get sent back across the network to the client stub.
9. The client stub reads the messages from the local kernel.
10. The client stub then returns the results to the client function, converting them from the network representation to a local one if necessary.

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

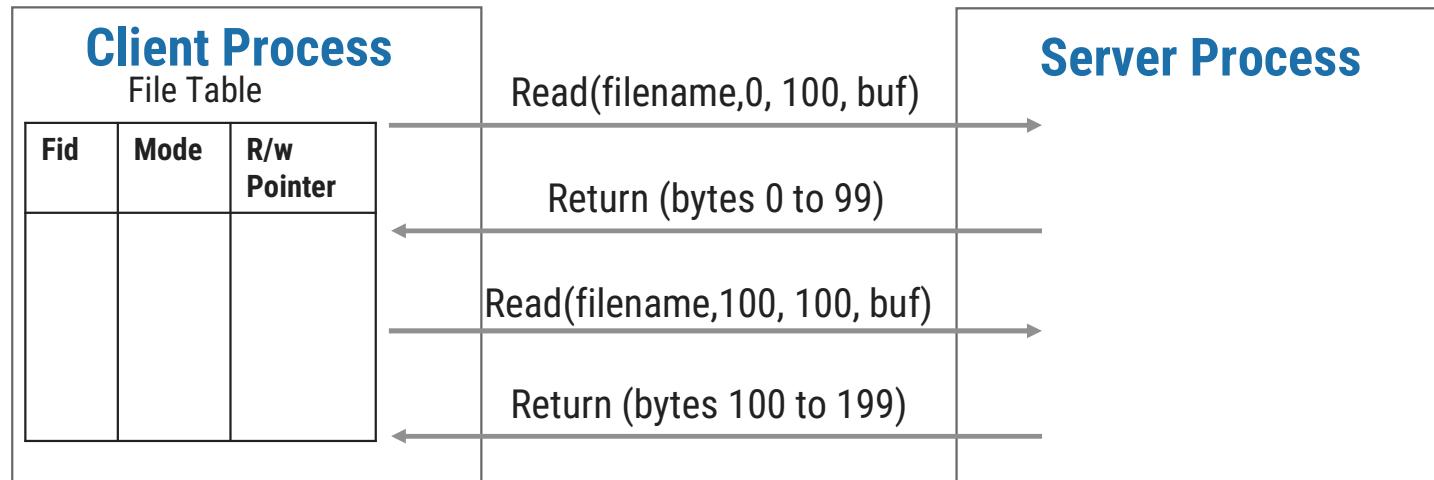


- After opening a file, if a client makes two subsequent Read (fid, 100, buf) requests, for the first request the first 100 bytes (bytes 0 to 99) will be read and for the second request the next 100 bytes (bytes 100 to 199) will be read.

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 remember 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

0	3	2	1	0
L	L	I	J	

Original message on
the Pentium

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

The message after
receipt on the SPARC

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

The message after
being inverted

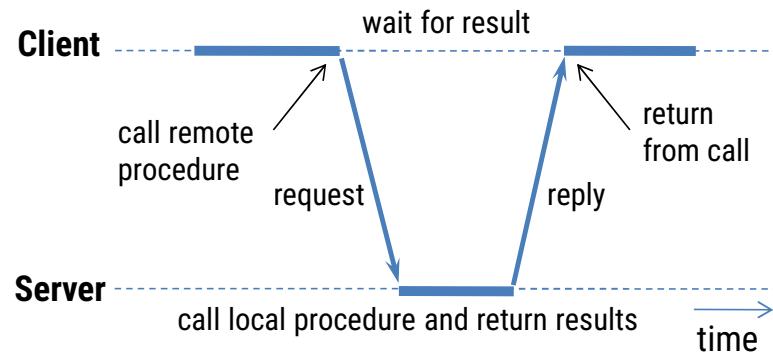
Asynchronous RPC

- ▶ 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

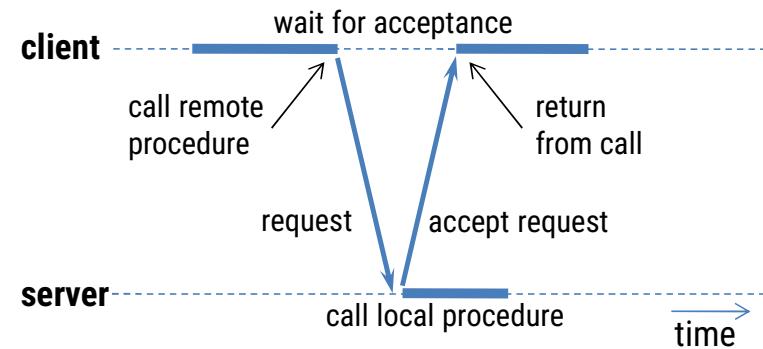
Asynchronous RPC

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

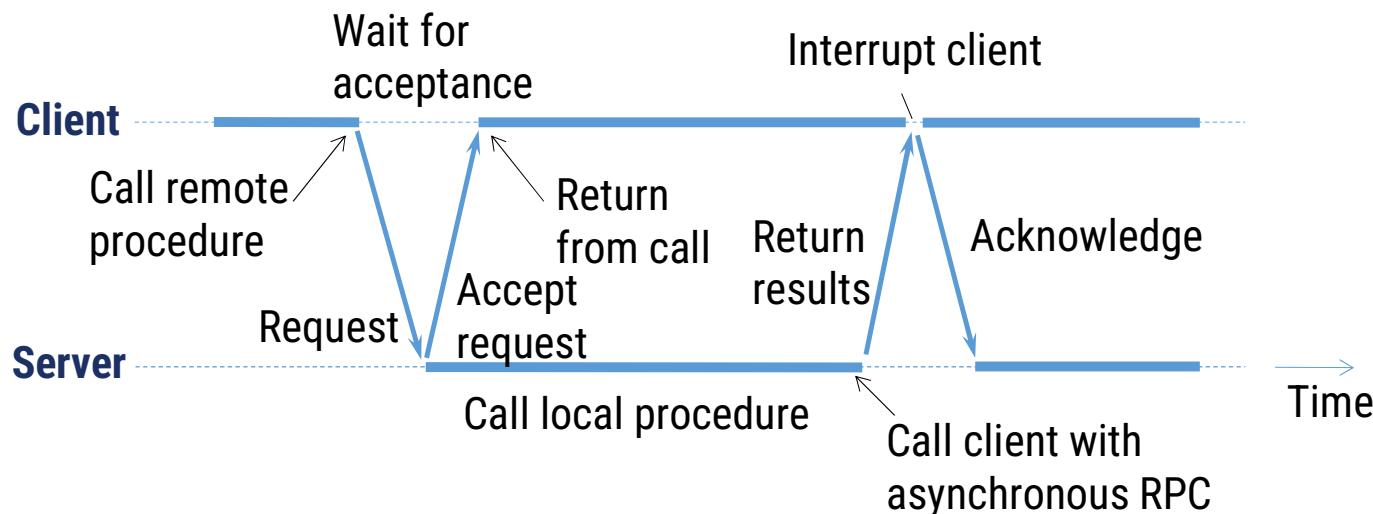


The interaction using **asynchronous RPC**

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.

Berkeley Sockets

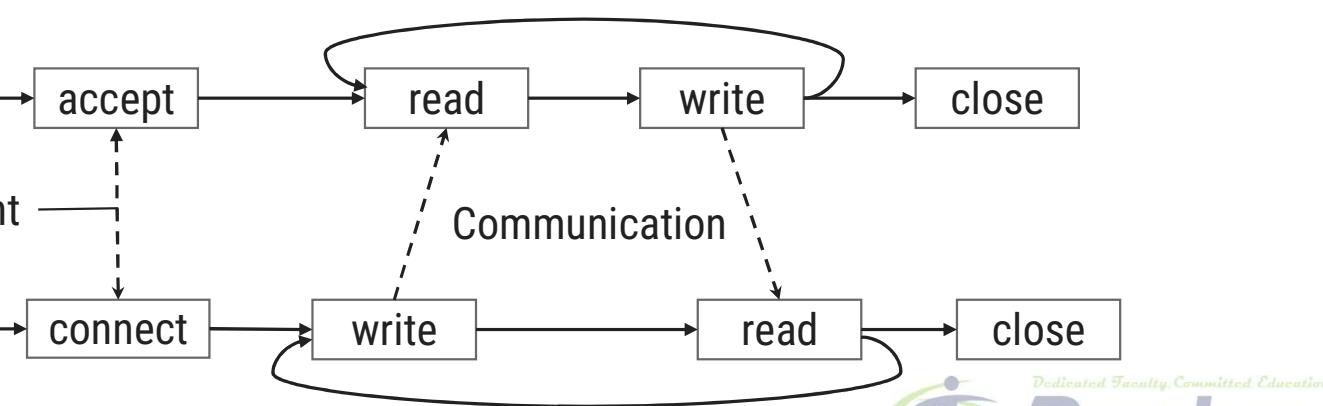
Primitive	Meaning	Primitive	Meaning
Socket	Create a new communication end point	Connect	Actively attempt to establish a connection
Bind	Attach a local address to a socket	Send	Send some data over the connection
Listen	Announce willingness to accept connections	Receive	Receive some data over the connection
Accept	Block caller until a connection request arrives	Close	Release the connection
		Write	Send data on the connection
		Read	Get data that was sent on the connection

Server



Synchronization point

Client



The Message-Passing Interface (MPI)

- ▶ The Message Passing Interface (MPI) is a library specification for message passing.
- ▶ 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

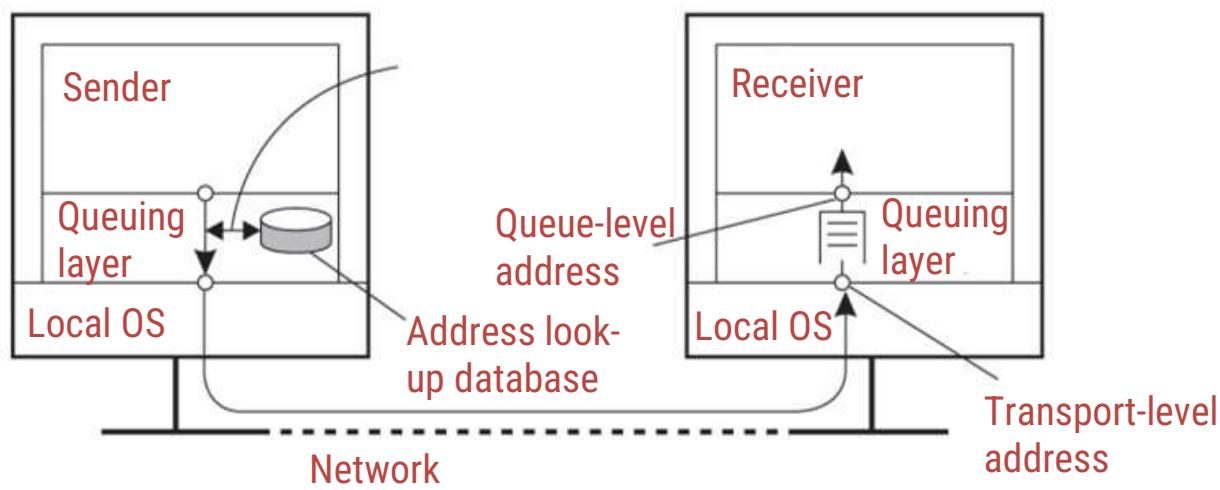
Message-passing primitives of MPI

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

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

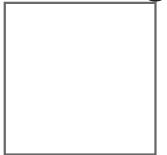


Message-Queuing Model

- ▶ 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
- ▶ Loosely-coupled communication
- ▶ Systemwide unique name of the destination queue is used for addressing

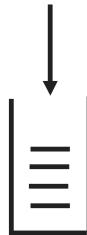
Loosely-Coupled Communication

**Sender
running**



**Receiver
running**

**Sender
running**



**Receiver
passive**

**Sender
passive**



**Receiver
running**

**Sender
passive**



**Receiver
passive**

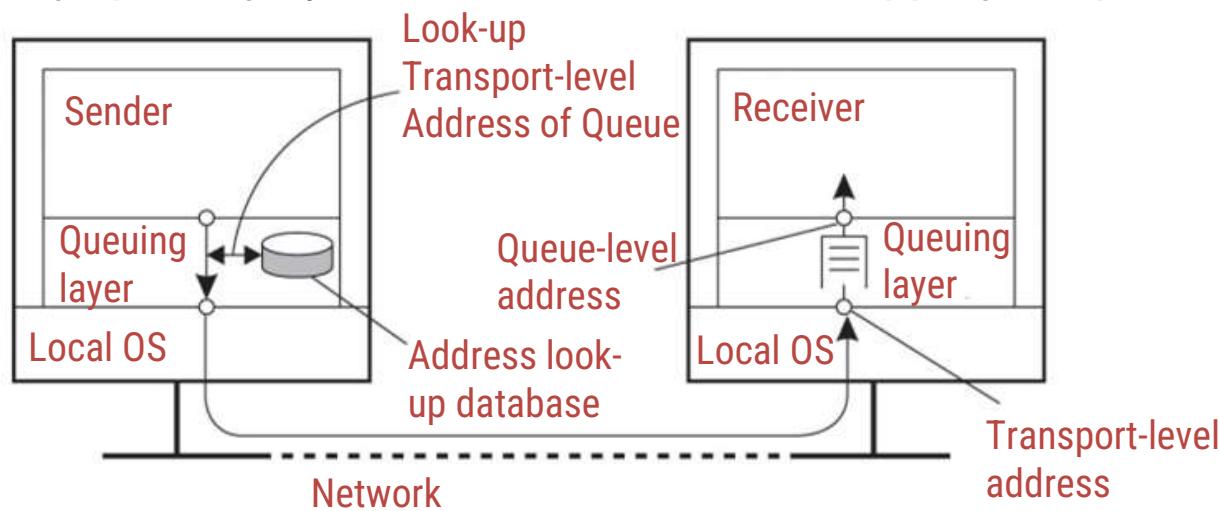
Message-Oriented Persistent Communication

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.

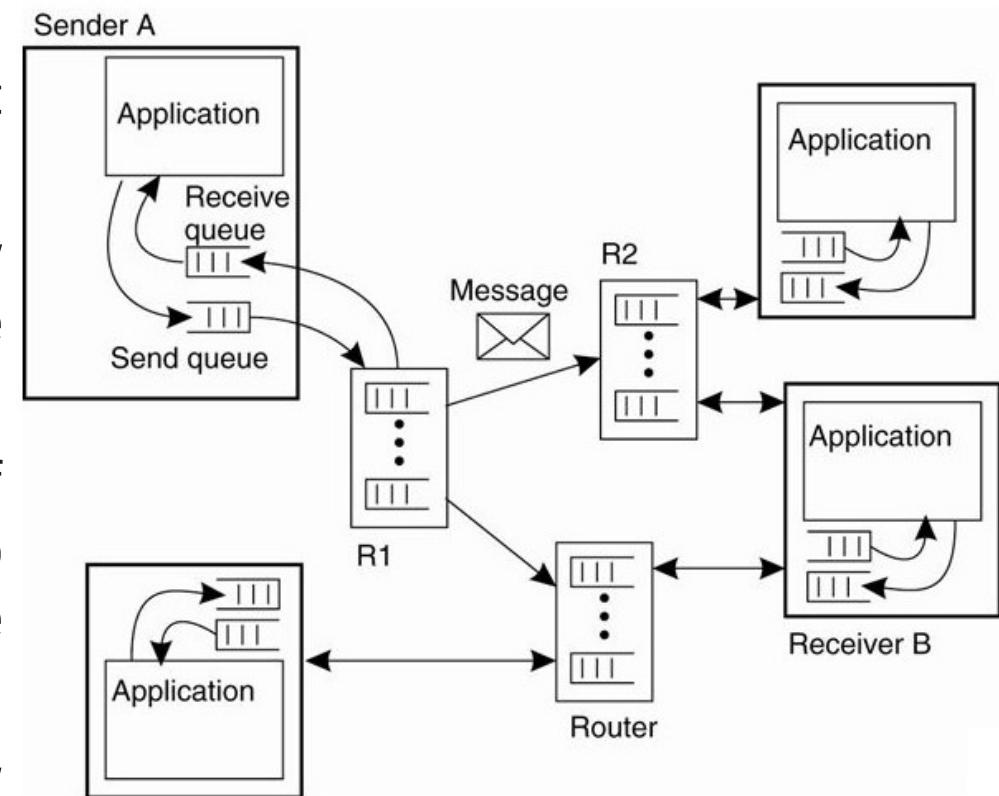
General Architecture of a Message-Queuing System

- ▶ 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

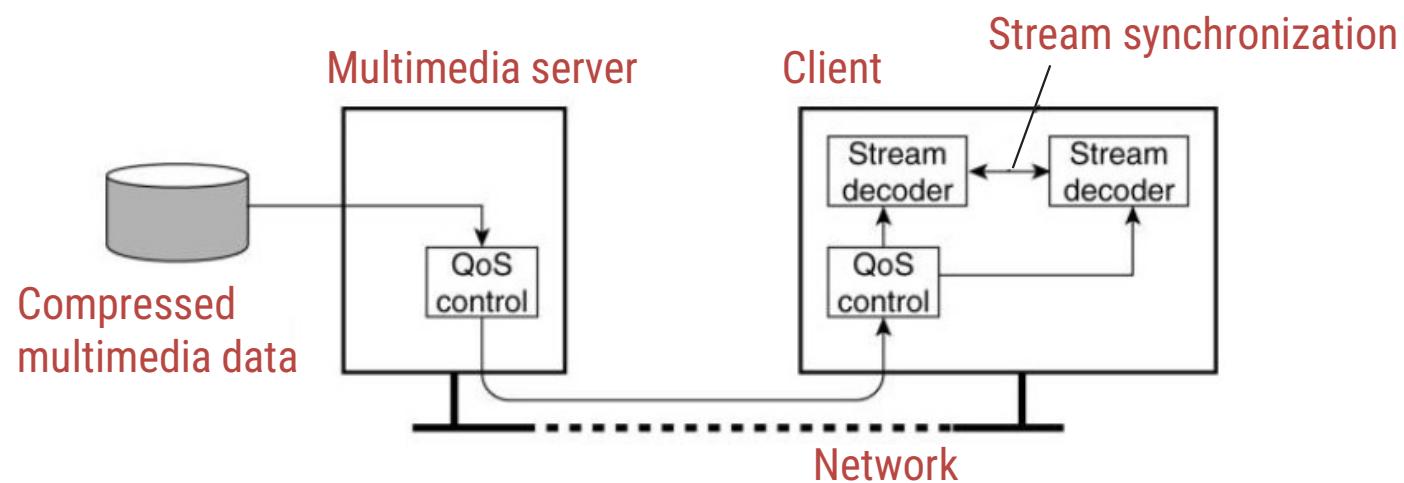
- ▶ RPC, RMI, message-oriented communication are based on the exchange of discrete messages
 - Timing might affect performance, but not correctness
- ▶ In stream-oriented communication the message content must be delivered at a certain rate, as well as correctly.
 - e.g., music or video
- ▶ Audio and 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 (DS)
GTU #3170719



Unit-3 Naming



Prof. Umesh H. Thoriya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot

✉ umesh.thoriya@darshan.ac.in
📞 9714233355



Topics to be covered

- Names
- Identifiers
- Addresses
- Flat Naming
- Structured Naming
- Attribute-Based Naming



Names, identifiers, and addresses

- ▶ **Name:** set of bits/characters used to identify/refer to an entity, a collective of entities, etc. in a context
 - 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

Broadcasting

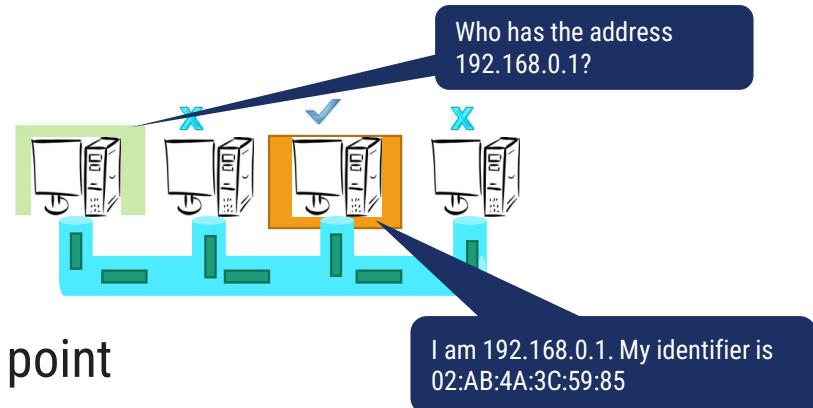
► Approach: Broadcast the identifier to the complete network. The entity associated with the identifier responds with its current address

► Example: Address Resolution Protocol (ARP)

- Resolve an IP address to a MAC address.
- In this application
 - IP address is the identifier of the entity
 - MAC address is the address of the access point

► Challenges:

- Not scalable in large networks
 - This technique leads to flooding the network with broadcast messages
- Requires all entities to listen to all requests

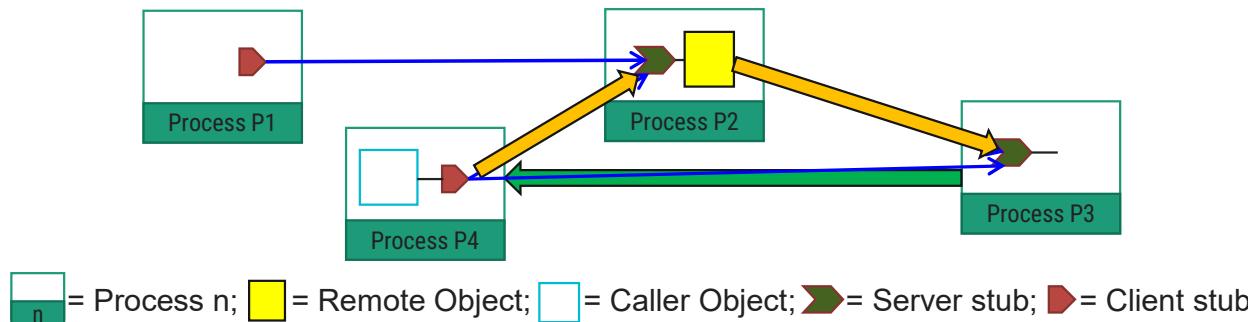


Forwarding Pointers

- ▶ Forwarding Pointers enables locating mobile entities
 - Mobile entities move from one access point to another
 - When an entity moves from location A to location B, it leaves behind (in A) a reference to its new location at B
- ▶ Name resolution mechanism
 - Follow the chain of pointers to reach the entity
 - Update the entity's reference when the present location is found
- ▶ Challenges:
 - Long chains lead to longer resolution delays
 - Long chains are prone to failure due to broken links

Forwarding Pointers – An Example

- ▶ Stub-Scion Pair (SSP) chains implement remote invocations for mobile entities using forwarding pointers
 - Server stub is referred to as Scion in the original paper
- ▶ Each forwarding pointer is implemented as a pair: (client stub, server stub)
 - The server stub contains a local reference to the actual object or a local reference to another client stub
- ▶ When object moves from A (e.g., P2) to B (e.g., P3),
 - It leaves a client stub at A (i.e., P2)
 - It installs a server stub at B (i.e., P3)



Home-Based Approaches

- ▶ Each mobile entity has a fixed IP address (home address)
 - Entity's home address is registered at a naming service
 - Home node keeps track of current address of the entity (care-of address)
 - Entity updates the home about its current address (care-of address) whenever it moves
- ▶ Name resolution
 - Client contacts the home to obtain the current address
 - Client then contacts the entity at the current location
- ▶ Challenges:
 - Home address is permanent for an entity's lifetime
 - If the entity permanently moves, then a simple home-based approach incurs higher communication overhead
 - Poor geographical scalability
 - Consider the scenario where the clients are nearer to the mobile entity than the home entity

Home-Based Approaches

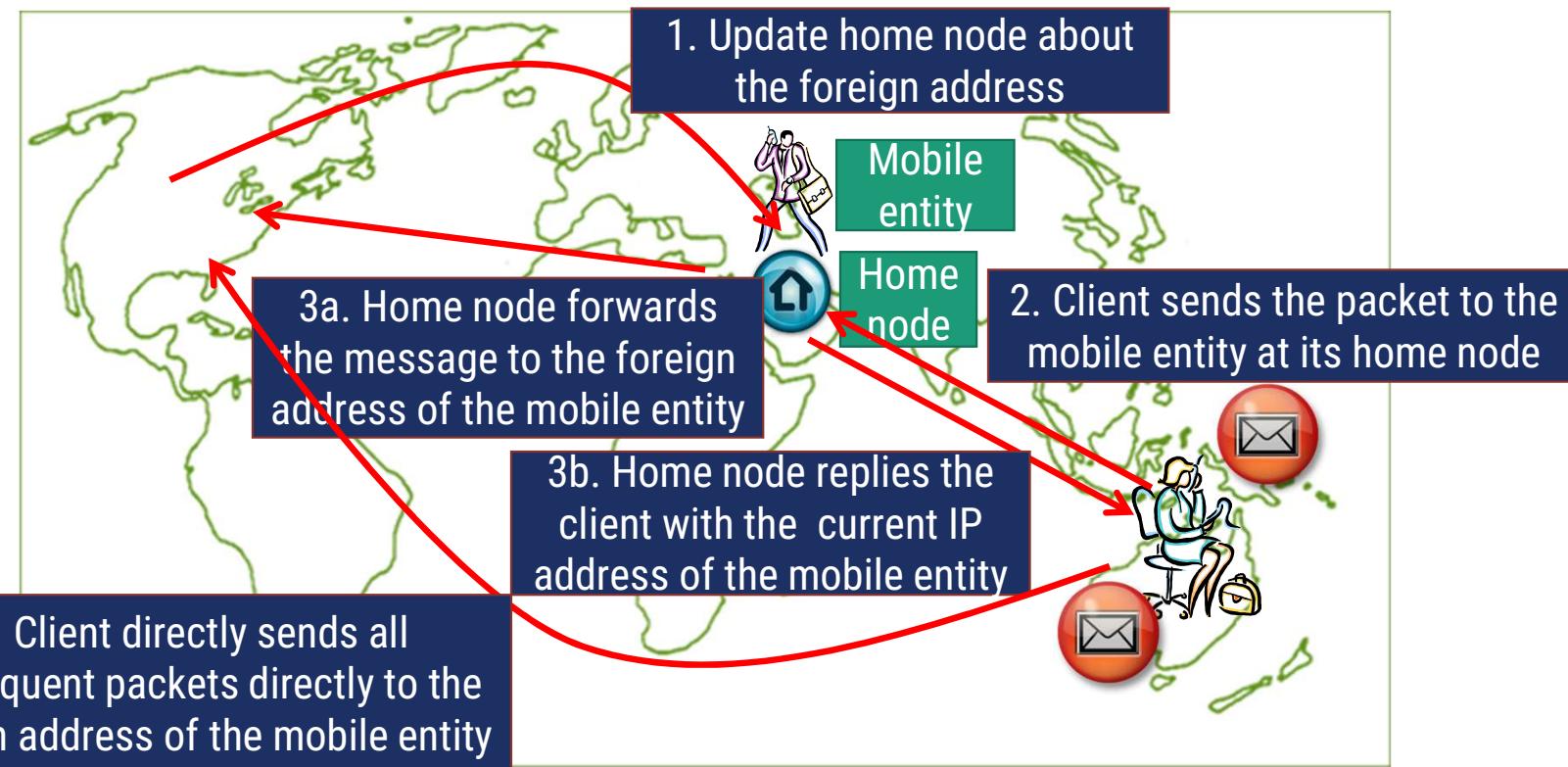
Single-tiered scheme:

- ▶ Let a home keep track of where the entity is
 - An entity's home address is registered at a naming service
 - The home registers the foreign address of the entity
 - Clients always contact the home first, and then continues with the foreign location

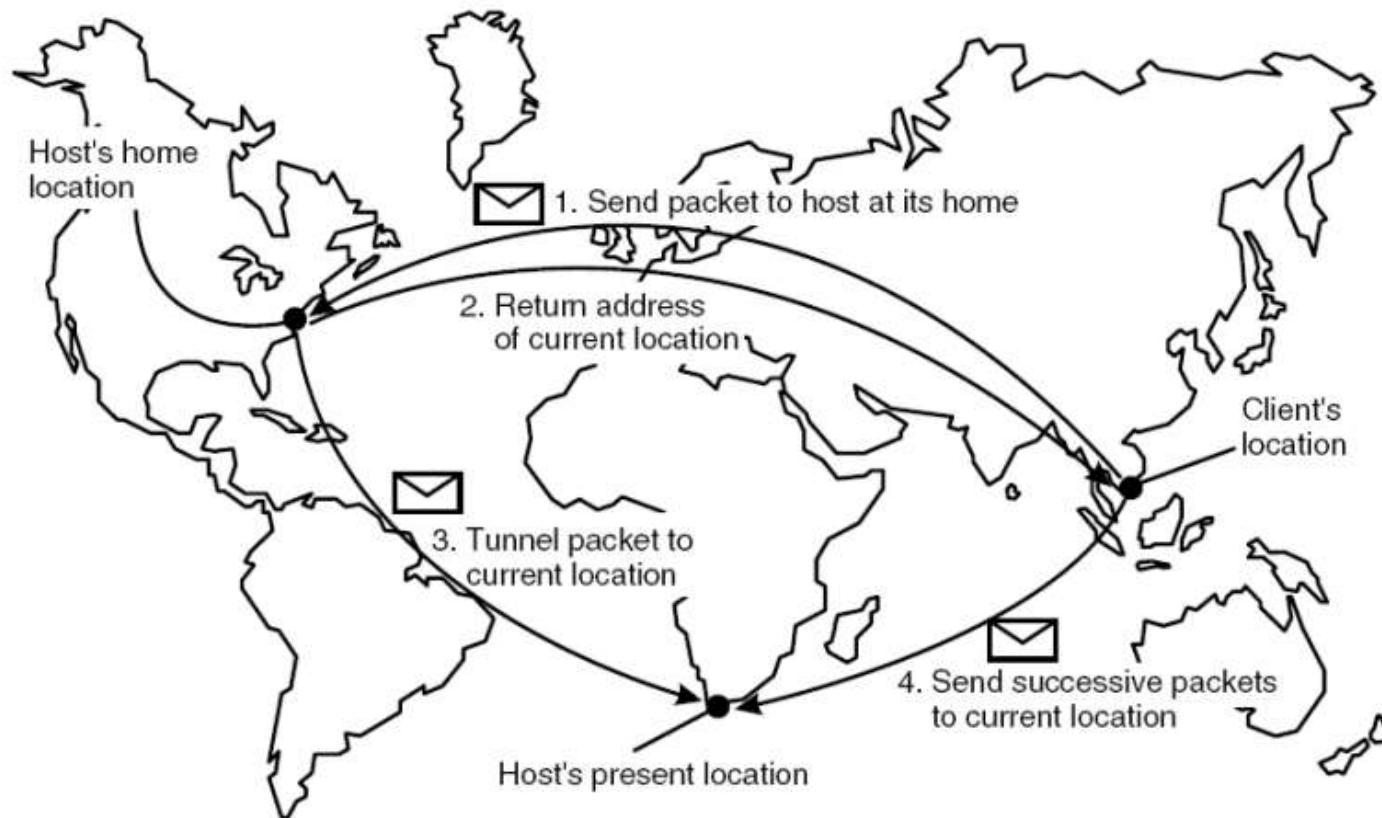
Two-tiered scheme:

- ▶ Keep track of visiting entities:
 - Check local visitor register first
 - Fall back to home location if local lookup fails

Home-based approaches – An example



Home-based approaches – An example



Distributed Hash Tables/Chord

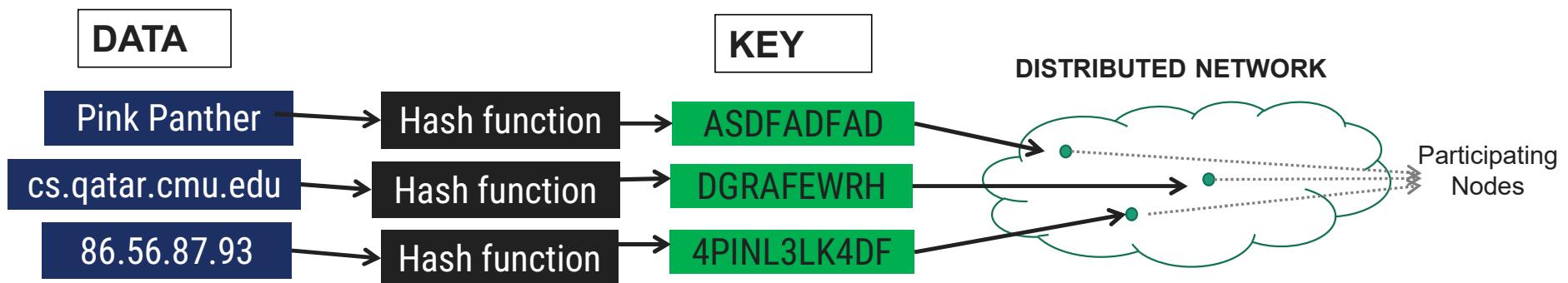
- ▶ Chord is representative of other DHT approaches
- ▶ It is based on an m-bit identifier space: both host node and entities are assigned identifiers from the name space.
 - Entity identifiers are also called keys.
 - Entities can be anything at all
- ▶ An m-bit identifier space = 2^m identifiers.
 - m is usually 128 or 160 bits, depending on hash function used.
- ▶ Each node has an m-bit id, obtained by hashing some node identifier (IP address?)
- ▶ Each entity has a key value, determined by the application (not Chord) which is hashed to get its m-bit identifier k
- ▶ Nodes are ordered in a virtual circle based on their identifiers.
- ▶ An entity with key k is assigned to the node with the smallest identifier id such that $id \geq k$. (the successor of k)

Chord

- ▶ Each node p knows its immediate neighbors, its immediate successor, $\text{succ}(p + 1)$ and its predecessor, denoted $\text{pred}(p)$.
- ▶ When given a request for key k , a node checks to see if it has the object whose id is k . If so, return the entity; if not, forward request to one of its two neighbors.
- ▶ Requests hop through the network one node at a time.

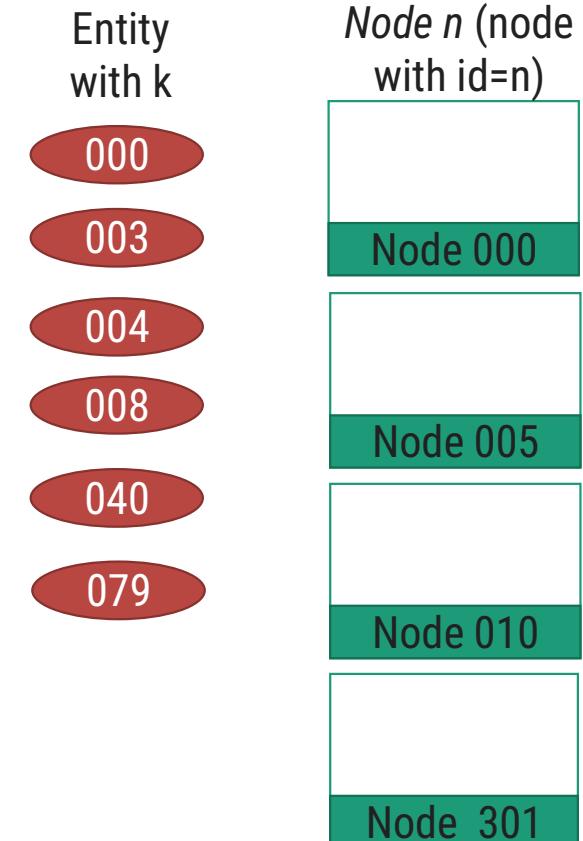
Chord

- ▶ DHT is a distributed system that provides a lookup service similar to a hash table
 - (key, value) pair is stored in the nodes participating in the DHT
 - The responsibility for maintaining the mapping from keys to values is distributed among the nodes
 - Any participating node can serve in retrieving the value for a given key



Chord

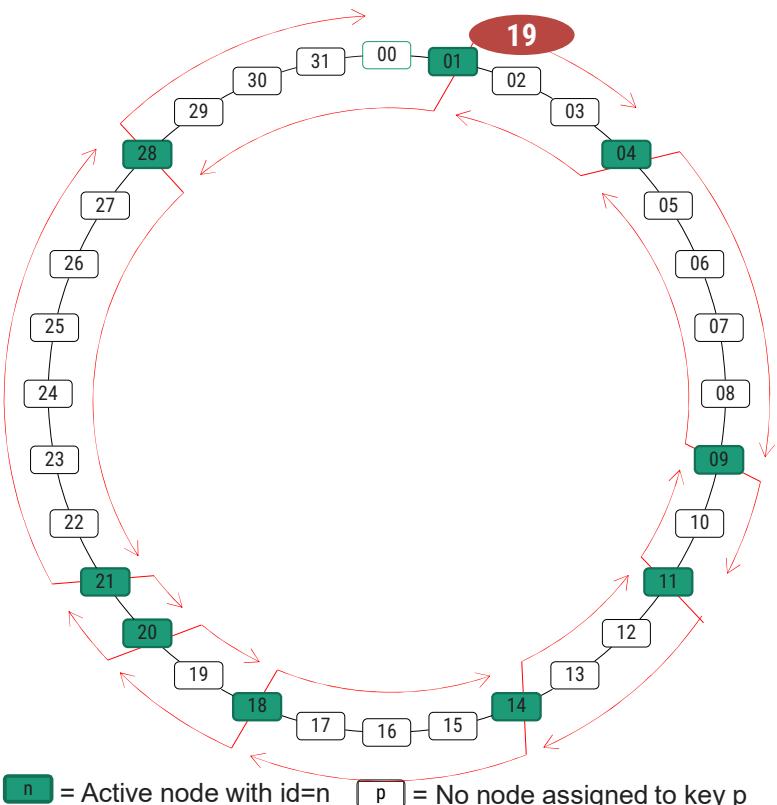
- ▶ Chord assigns an m-bit identifier (randomly chosen) to each node
 - A node can be contacted through its network address
- ▶ Alongside, it maps each entity to a node
 - Entities can be processes, files, etc.,
- ▶ Mapping of entities to nodes
 - Each node is responsible for a set of entities
 - An entity with key k falls under the jurisdiction of the node with the smallest identifier $id \geq k$. This node is known as the successor of k, and is denoted by $\text{succ}(k)$



Map each entity with key k to node
 $\text{succ}(k)$

A Naive Key Resolution Algorithm

- ▶ The main issue in DHT is to efficiently resolve a key k to the network location of $\text{succ}(k)$
- ▶ Given an entity with key k , how to find the node $\text{succ}(k)$?

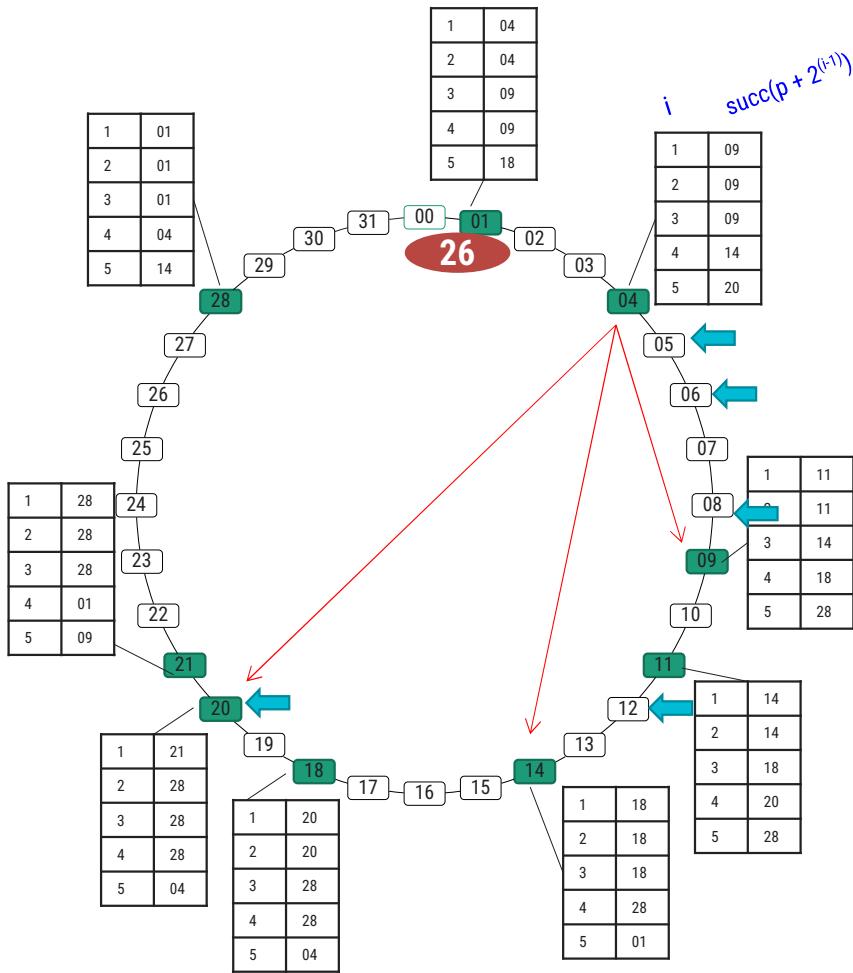


1. All nodes are arranged in a logical ring according to their IDs
2. Each node 'p' keeps track of its immediate neighbors: $\text{succ}(p)$ and $\text{pred}(p)$
3. If 'p' receives a request to resolve key 'k':
 - If $\text{pred}(p) < k \leq p$, node p will handle it
 - Else it will forward it to $\text{succ}(p)$ or $\text{pred}(p)$

Solution is not scalable:

- As the network grows, forwarding delays increase
- Key resolution has a time complexity of $O(n)$

Key Resolution in Chord



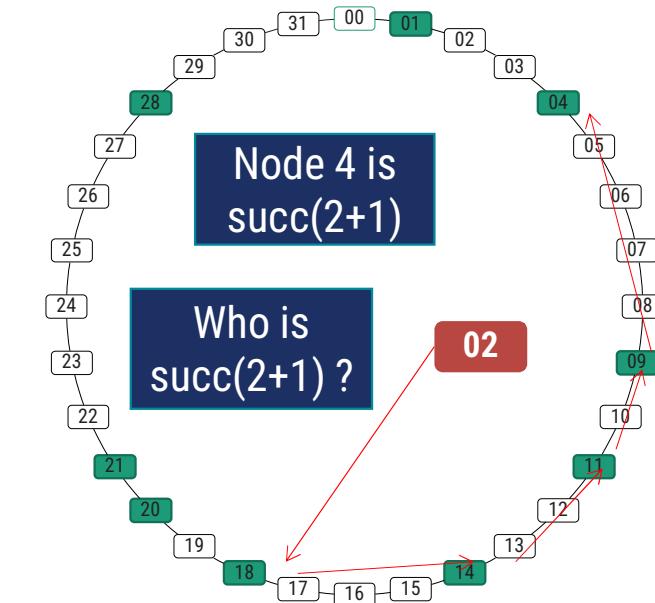
- Chord improves key resolution by reducing the time complexity to $O(\log n)$
- 1. All nodes are arranged in a logical ring according to their IDs
- 2. Each node 'p' keeps a table FT_p of at-most m entries. This table is called Finger Table

$$FT_p[i] = \text{succ}(p + 2^{(i-1)})$$

NOTE: $FT_p[i]$ increases exponentially
- 3. If node 'p' receives a request to resolve key 'k':
 - Node p will forward it to node q with index j in FT_p where $q = FT_p[j] \leq k < FT_p[j+1]$
 - If $k > FT_p[m]$, then node p will forward it to $FT_p[m]$
 - If $k < FT_p[1]$, then node p will forward it to $FT_p[1]$

Chord – Join and Leave Protocol

- ▶ In large-scale distributed Systems, nodes dynamically join and leave (voluntarily or due to failures)
- ▶ If a node p wants to join:
 - It contacts arbitrary node, looks up for $\text{succ}(p+1)$, and inserts itself into the ring
- ▶ If node p wants to leave:
 - It contacts $\text{pred}(p)$ and $\text{succ}(p+1)$ and updates them



Finger Tables – A Better Way

► Each node maintains a finger table containing at most m entries.

► For a given node p , the i th entry is

$$FT_p[i] = \text{succ}(p + 2^{i-1}), \text{ the 1}^{\text{st}} \text{ node succeeding } p \text{ by at least } 2^{i-1}.$$

► Finger table entries are short-cuts to other nodes in the network.

► As the index in the finger table increases, the distance between nodes increases exponentially.

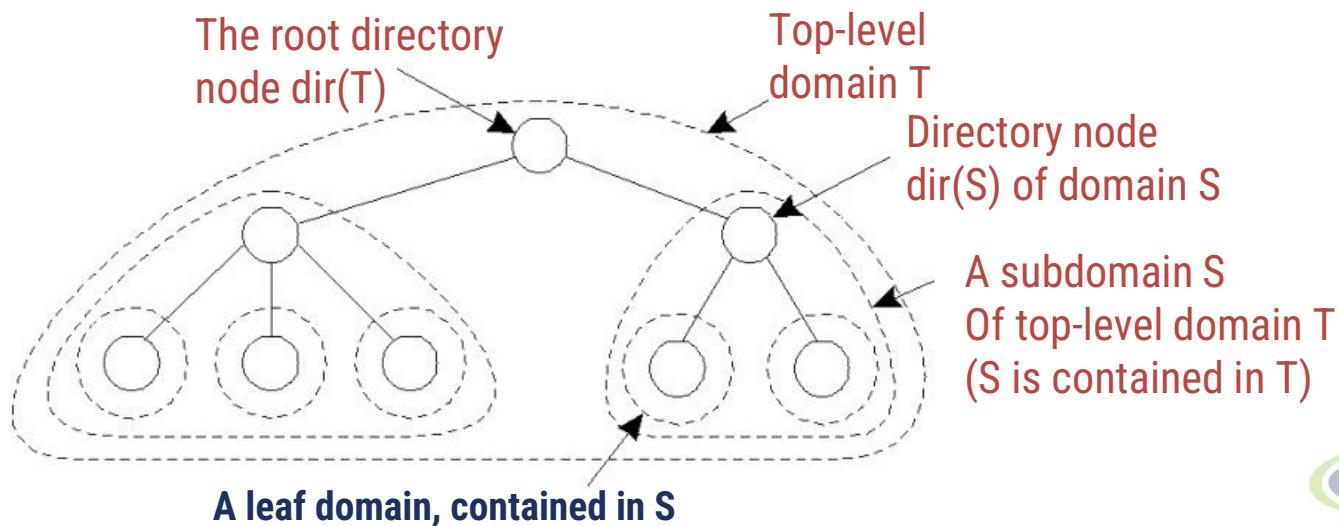
► To locate an entity with key value = k, beginning at node p

- If p stores the entity, return to requestor
- Else, forward the request to a node q in p's finger table
- Node q has index j in p's finger table; j satisfies the relation

$$q = FT_p[j] \leq k < FT_p[j + 1]$$

Hierarchical Approaches

- ▶ A flat name space with hierarchical administration
- ▶ Top level domain knows (or can find) all names
- ▶ Each sub-domain knows subset of names
- ▶ Local names resolved within own subset
- ▶ Other names cached as needed



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**

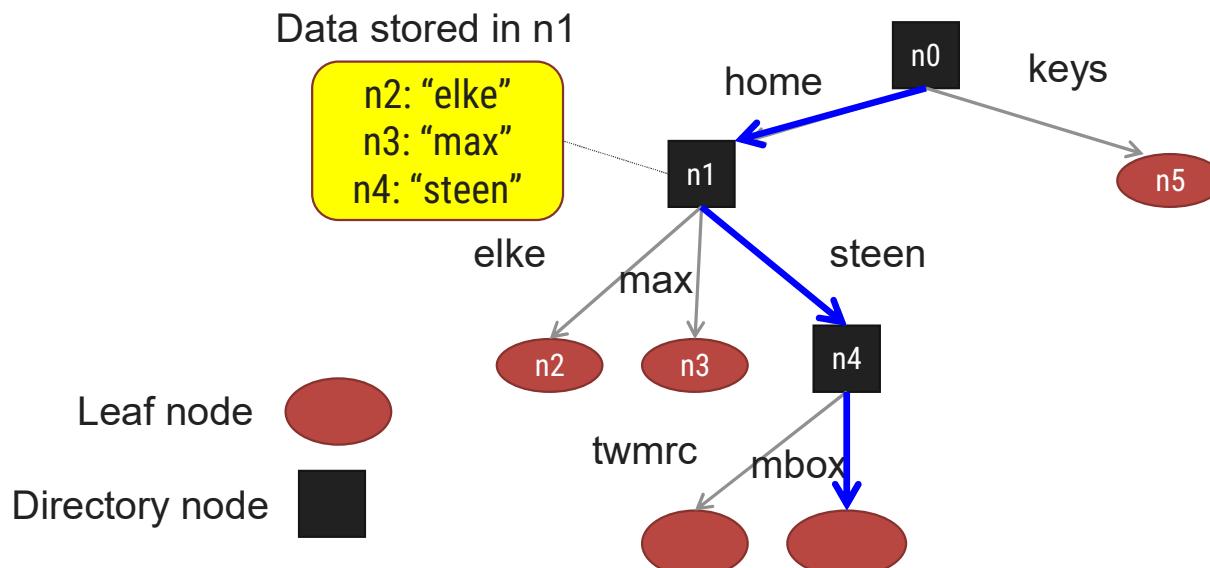
- Directory node refers to other leaf or directory nodes
- Each outgoing edge is represented by (edge label, node identifier)

- ▶ 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:
 - Name resolution cannot be accomplished without an initial directory node
 - The closure mechanism selects the implicit context from which to start name resolution
 - Examples:
 - www.qatar.cmu.edu: start at the DNS Server
 - /home/steen/mbox: start at the root of the file-system

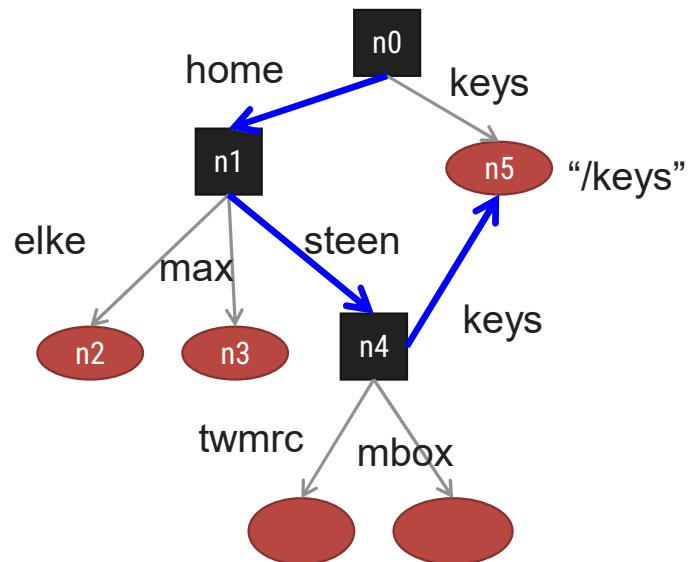
Name Linking

- ▶ 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

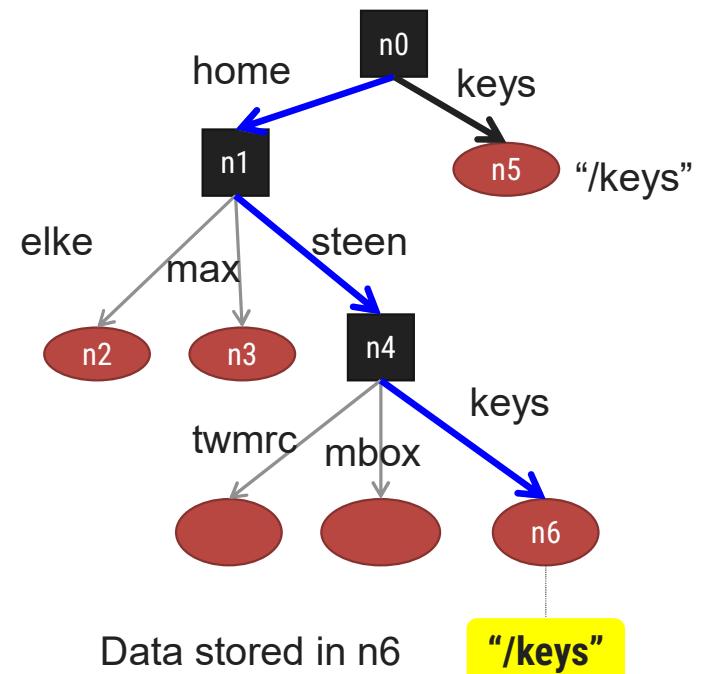
“/home/steen/keys” is a hard link to “/keys”



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

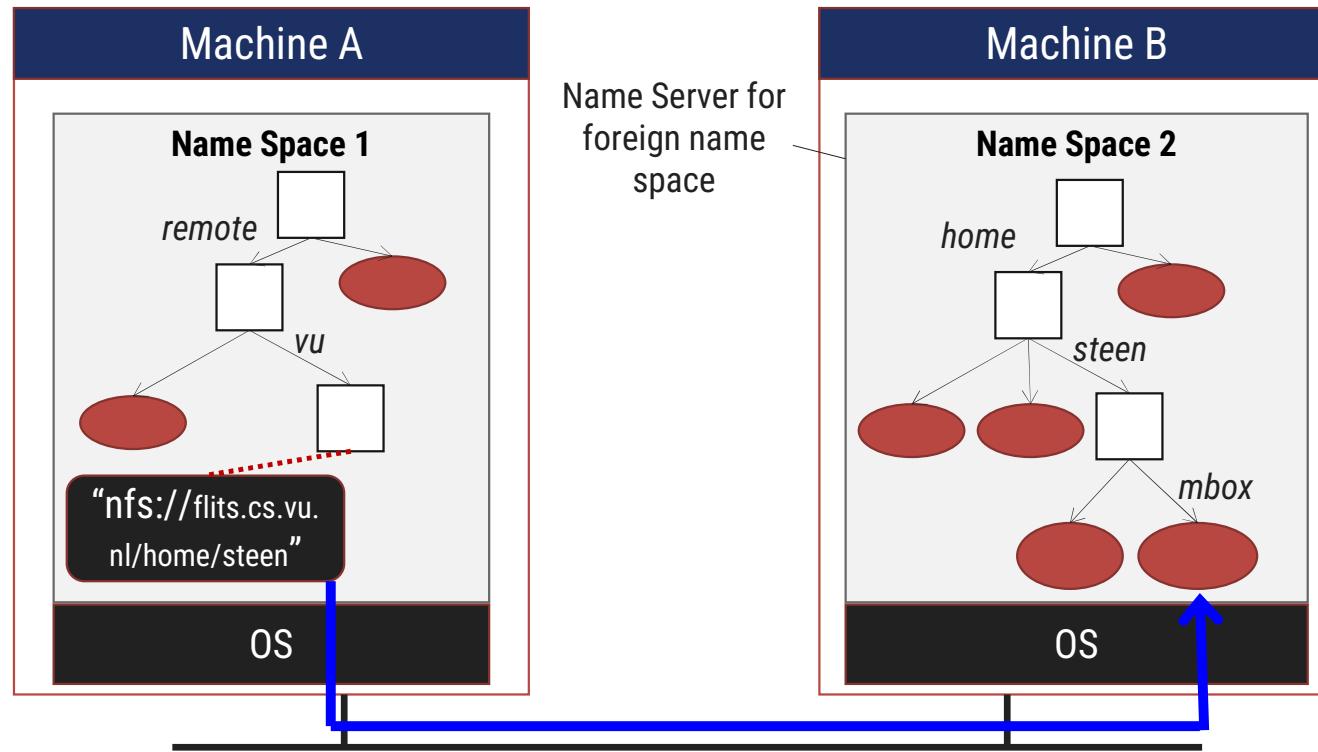
"/home/steen/keys" is a symbolic link to "/keys"



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

Global Layer

- Consists of high-level directory nodes
- Directory nodes are jointly managed by different administrations

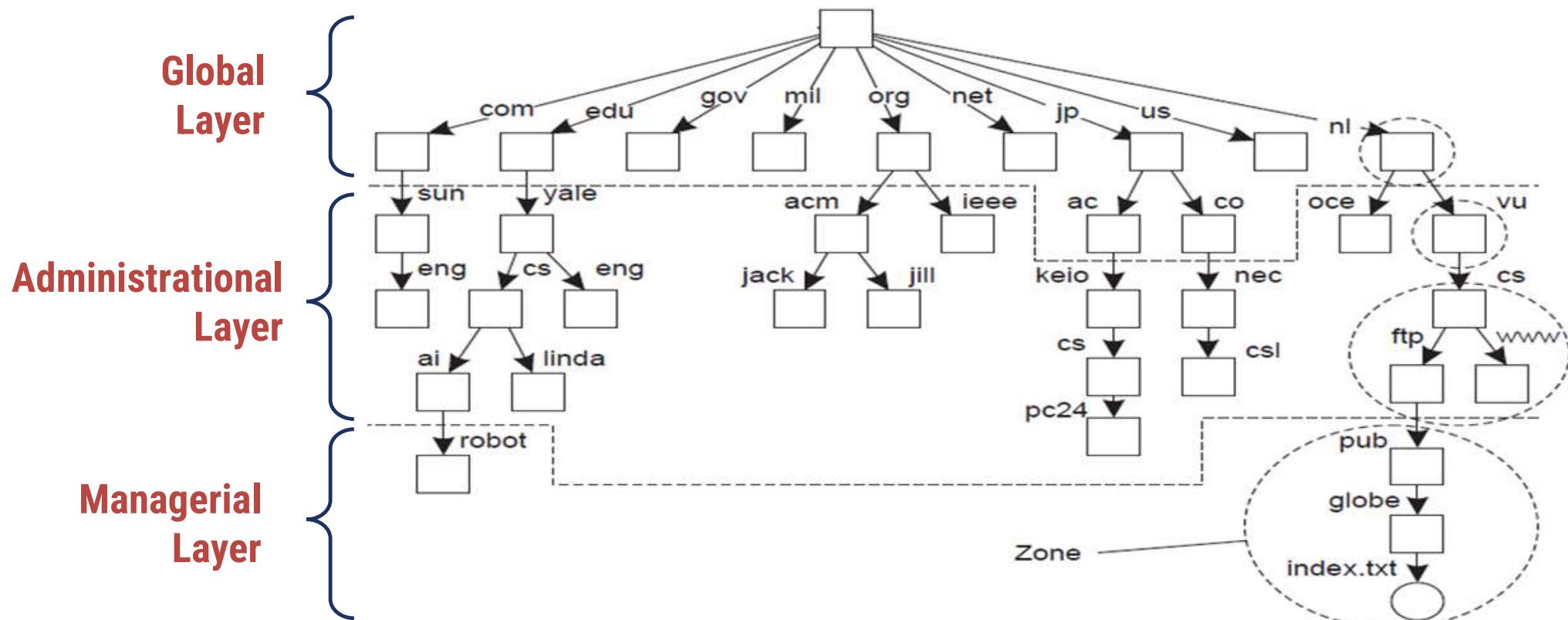
Administrative Layer

- Contains mid-level directory nodes
- Directory nodes grouped together in such a way that each group is managed by an administration

Managerial Layer

- Contains low-level directory nodes within a single administration
- The main issue is to efficiently map directory nodes to local name servers

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

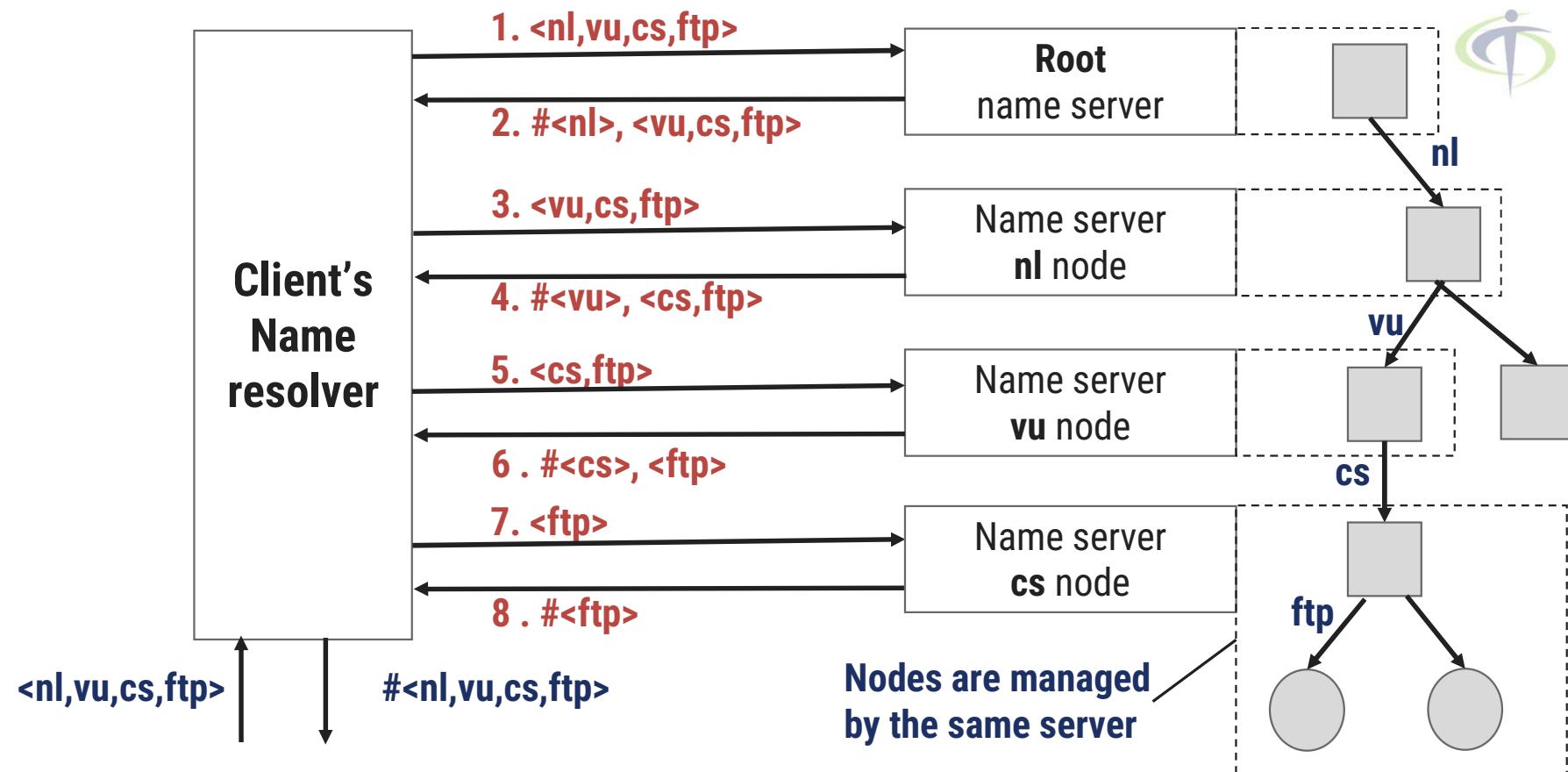
Distributed Name Resolution

- ▶ 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

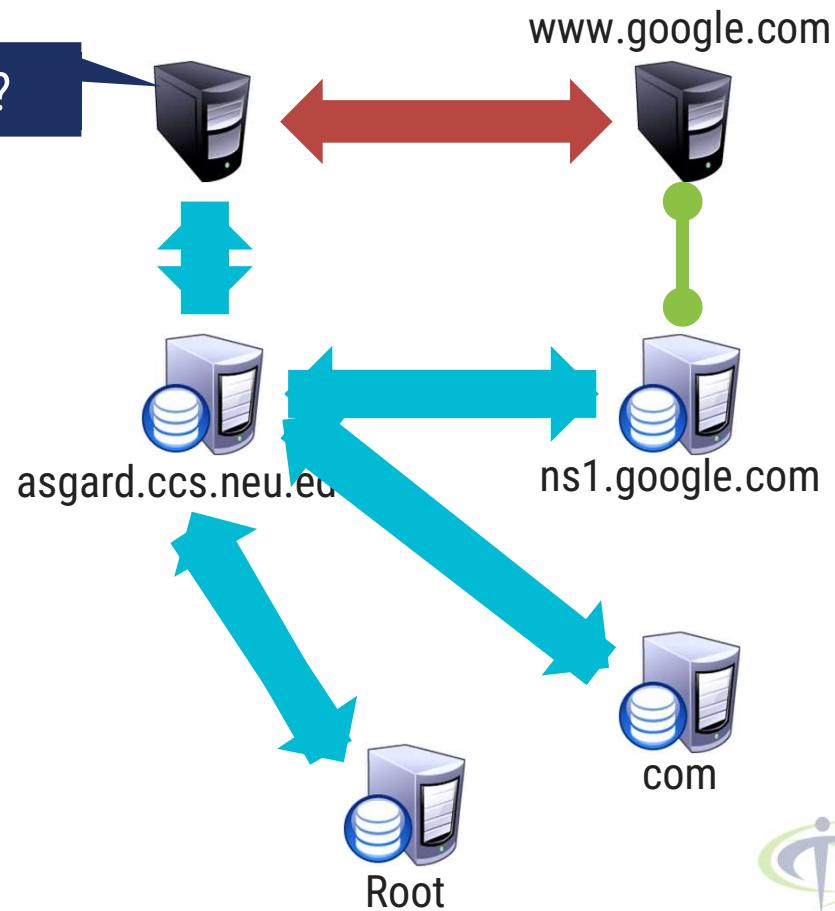


Resolving the name "*ftp.cs.vu.nl*"

<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

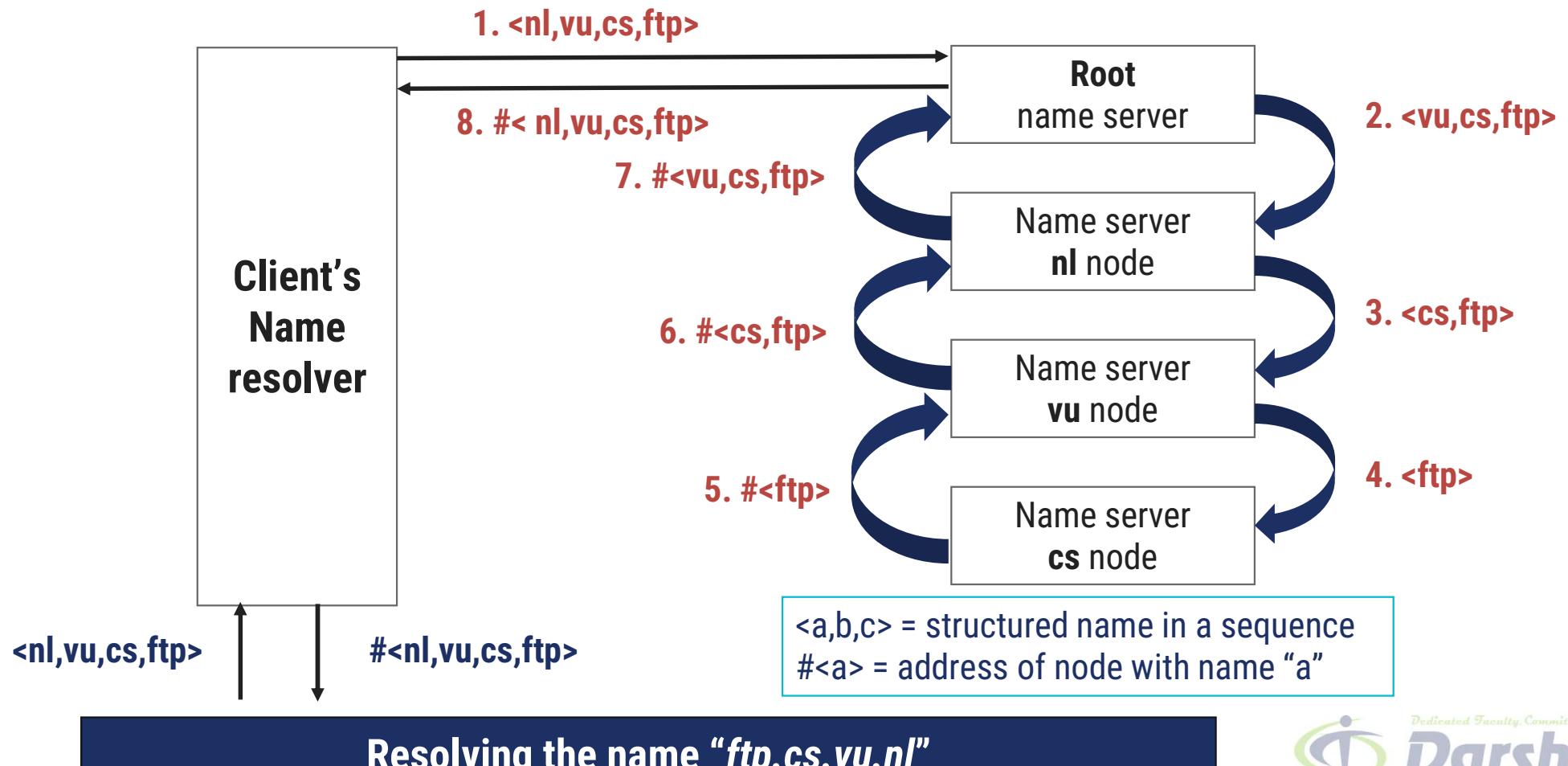
Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	<ftp>	#<ftp>	--	--	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<CS> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<CS> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,CS>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

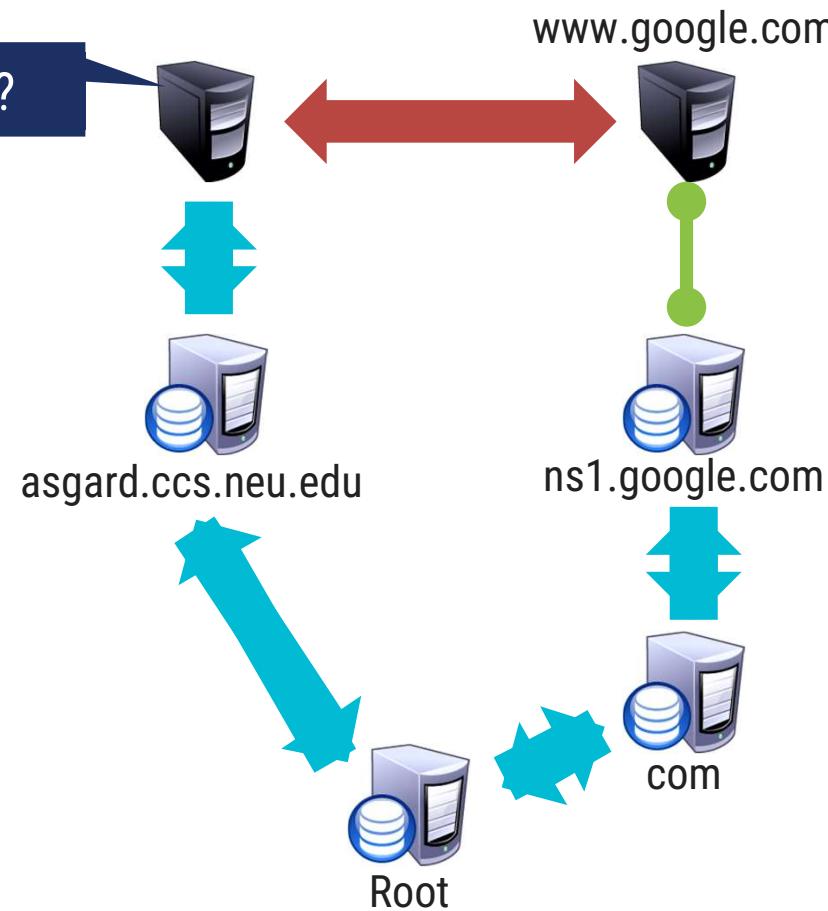
**Recursive name resolution of <nl, vu, cs, ftp>
name servers cache intermediate results for subsequent lookups**

Recursive Name Resolution - An Example



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 (e.g., yellow pages)
- ▶ 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.

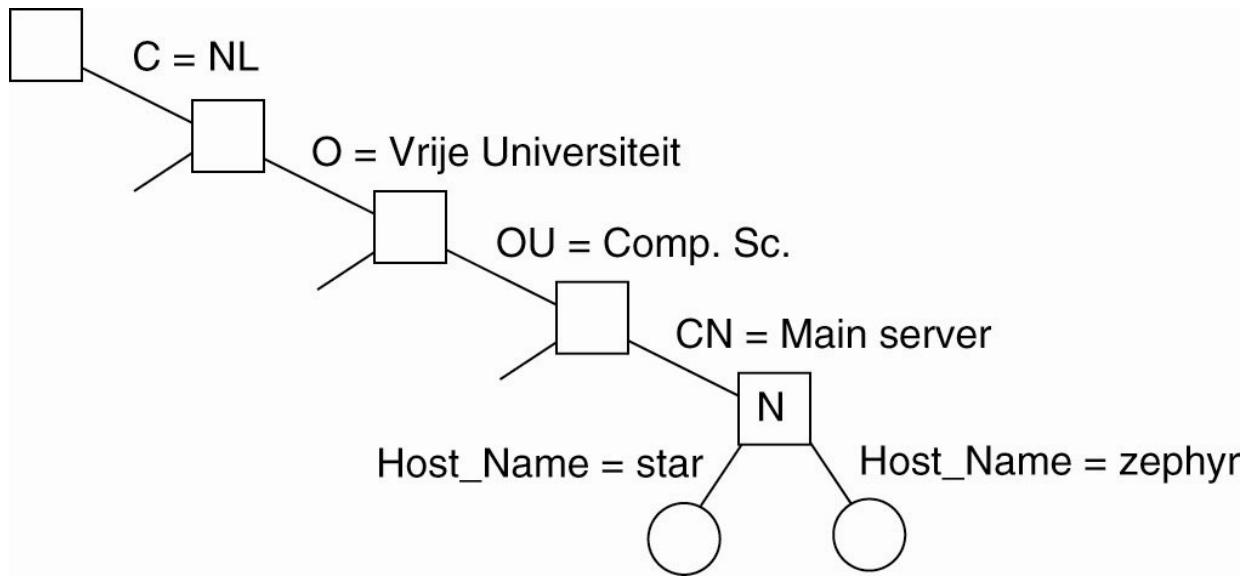
Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

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

Unit-4 Synchronization



Prof. Umesh H. Thoriya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot
✉ umesh.thoriya@darshan.ac.in
📞 9714233355





Topics to be covered

- Clock Synchronization
- Logical Clocks
- Mutual Exclusion
- Global Positioning Of Nodes
- Election Algorithms



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

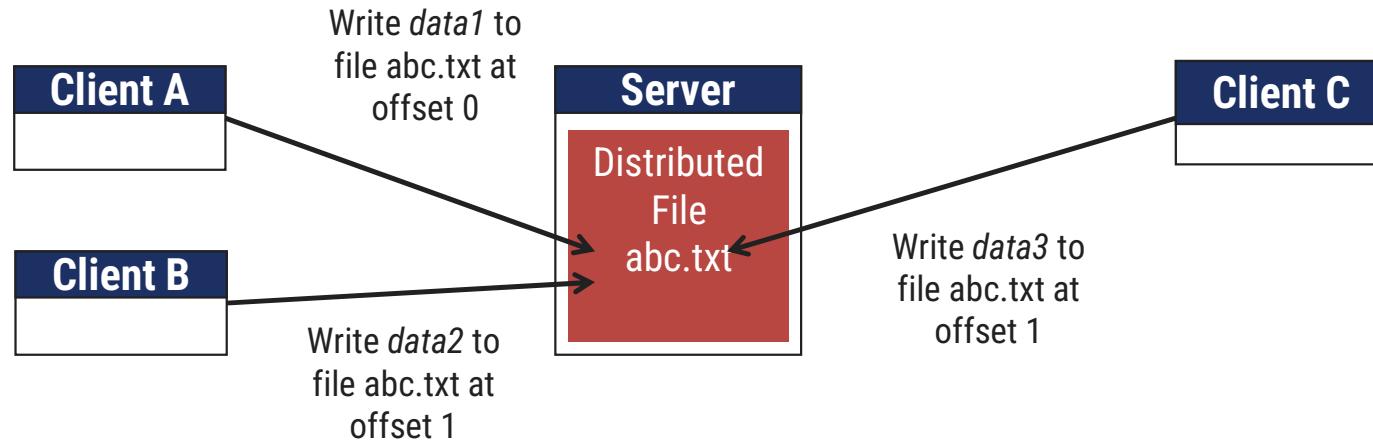
- ▶ Vehicle tracking in a City Surveillance System using a Distributed Sensor Network of Cameras
 - **Objective:** To keep track of suspicious vehicles
 - Camera Sensor Nodes are deployed over the city
 - Each Camera Sensor that detects a vehicle reports the time to a central server
 - Server tracks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

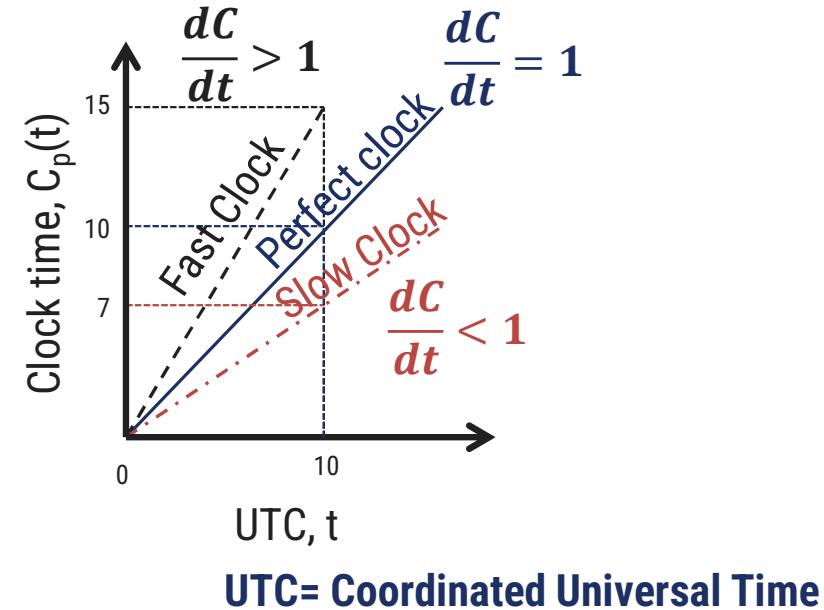
Tracking Time on a Computer

- ▶ 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
 $\frac{dC}{dt} = 1$
 - Fast clock:
 - The timer ticks more than 'H' interrupts a second
 $\frac{dC}{dt} > 1$
 - Slow clock:
 - The timer ticks less than 'H' interrupts a second
 $\frac{dC}{dt} < 1$



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

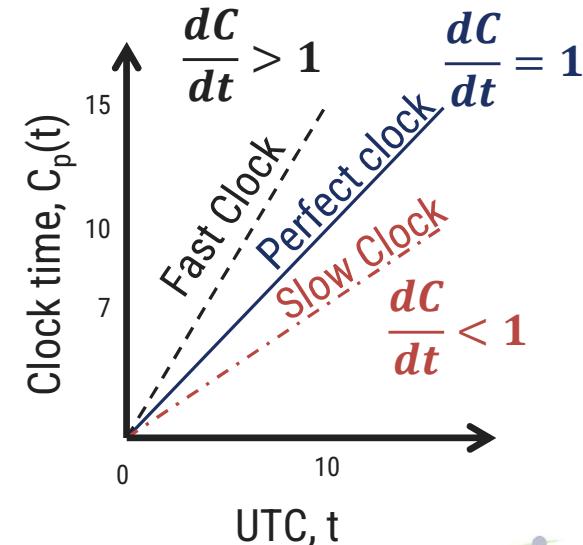
$$\text{Frequency} = dC/dt$$

- ▶ 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



8:01:48

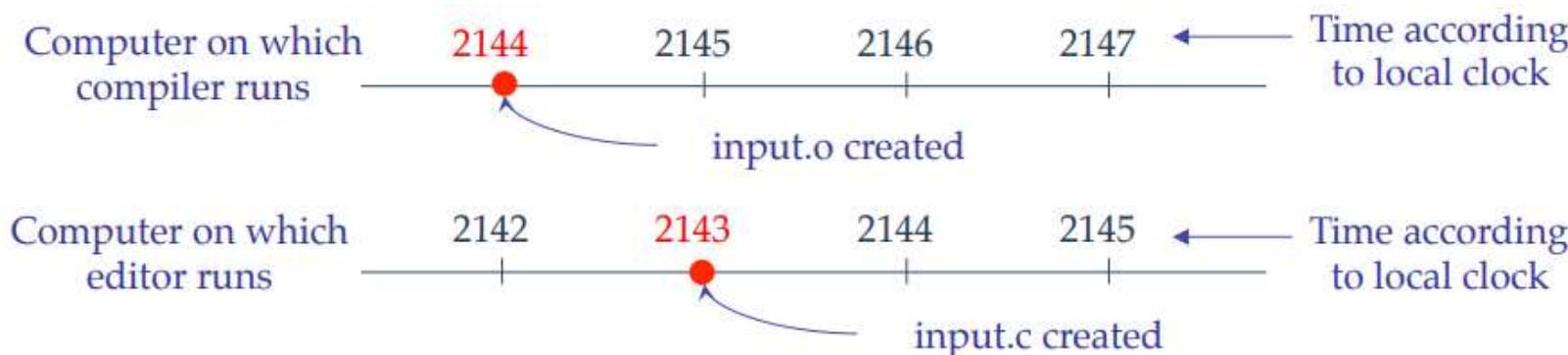
Skew = +108 seconds
+108 seconds/35 days
Drift = +3.1 sec/day

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:

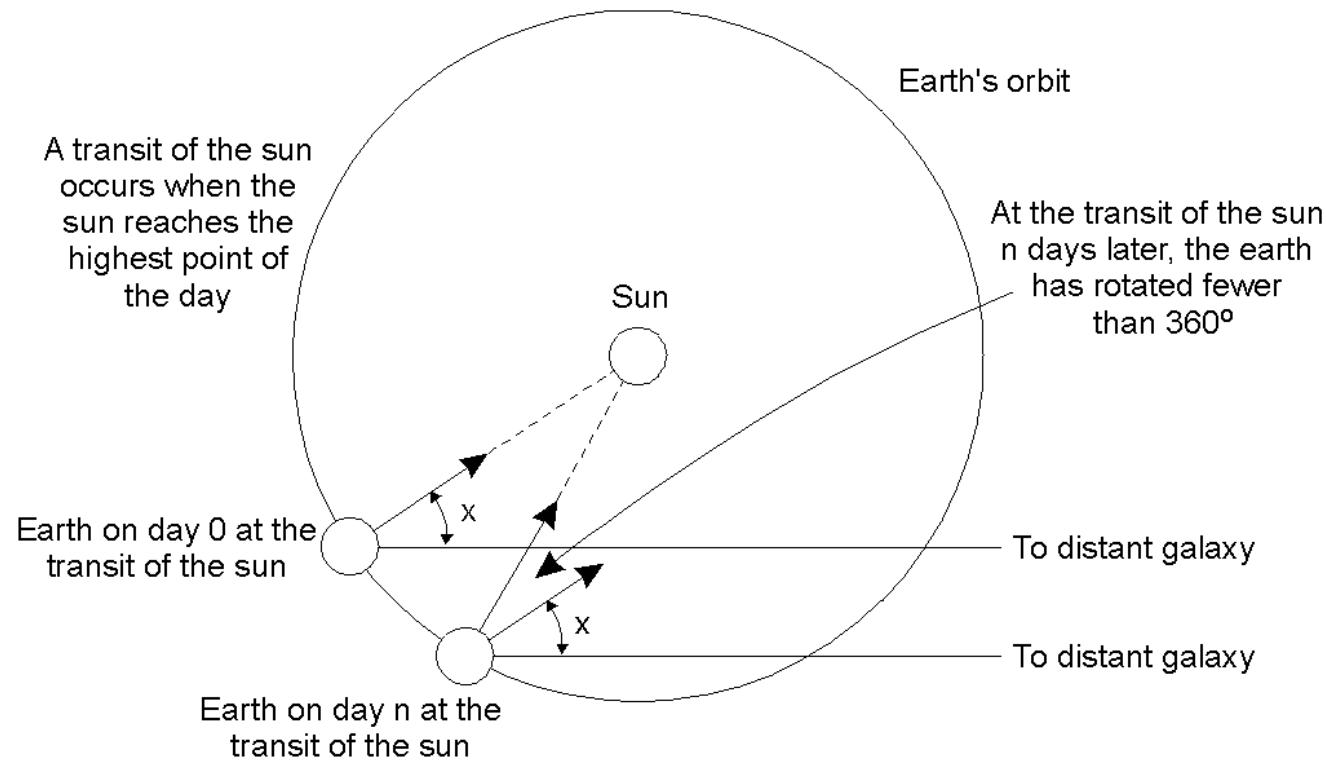


- Assume that the compiler and editor are on different machines
- `output.o` has time 2144 msec
- `output.c` is modified but is assigned time 2143 msec because the clock on its machine is slightly behind.
- Make** will not call the compiler.
- The resulting executable will have a mixture of object files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



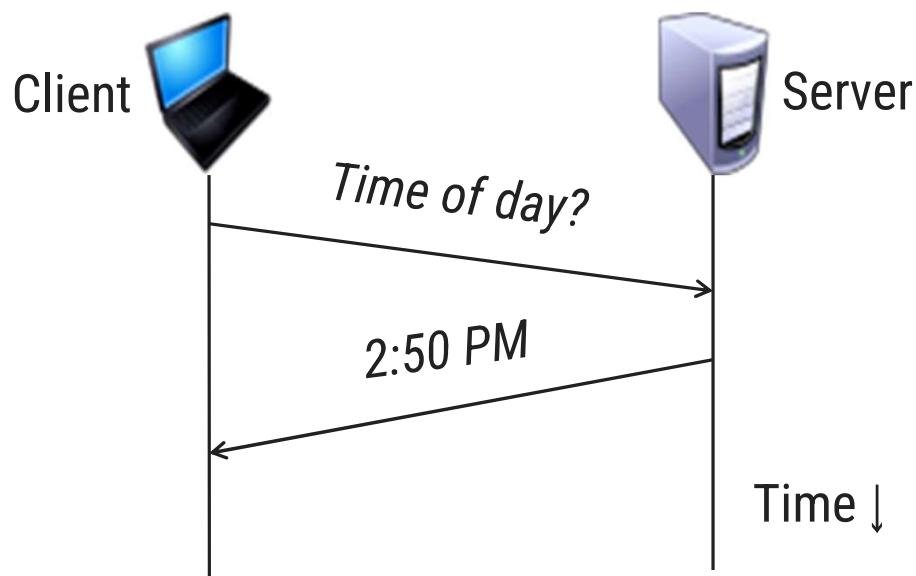
Computation of the mean solar day.

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

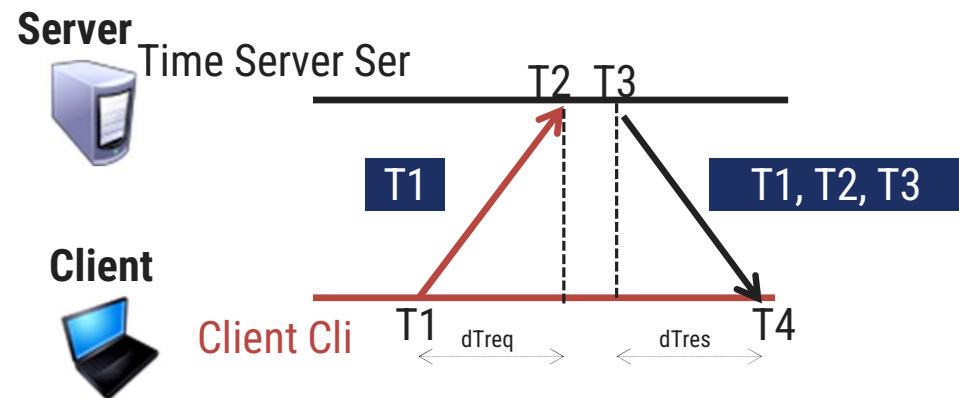
Cristian's Algorithm

- ▶ 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

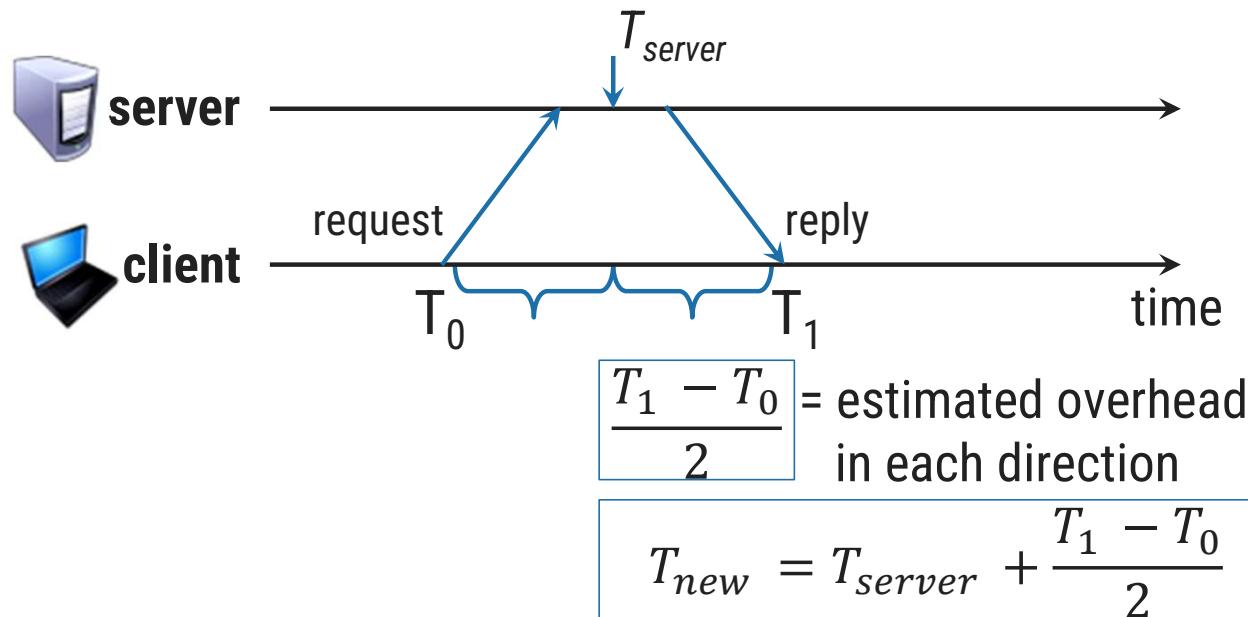
- ▶ Client Cli sends a request to Time Server Ser, time stamped its local clock time T1
- ▶ Ser will record the time of receipt T2 according to its local clock
- ▶ dTreq is network delay for request transmission
- ▶ Ser replies to Cli at its local time T3, piggybacking T1 and T2
- ▶ Cli receives the 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**

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

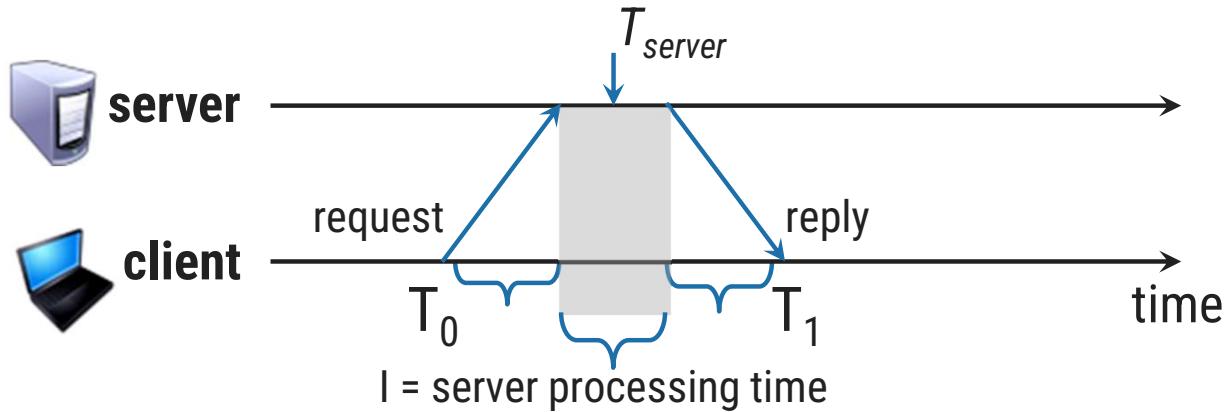


Cristian's Algorithm

- ▶ Note times:
 - Request sent: T₀
 - Reply received: T₁
- ▶ Assume network delays are symmetric.
- ▶ Client sets time to:

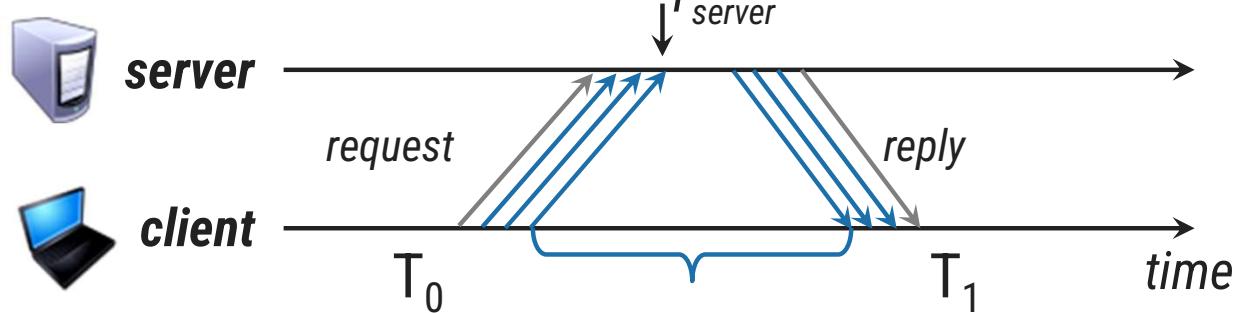


Cristian's Algorithm



$$T_{new} = T_{server} + \frac{T_1 - T_0 - I}{2}$$

Cristian's Algorithm



1. Several measurements of $(T_1 - T_0)$ are made, those measurements for which $(T_1 - T_0)$ exceeds some threshold value are considered to be unreliable and discarded.
2. Average of the remaining measurements is then calculated.

$\text{average}((T_1 - T_0)/2)$

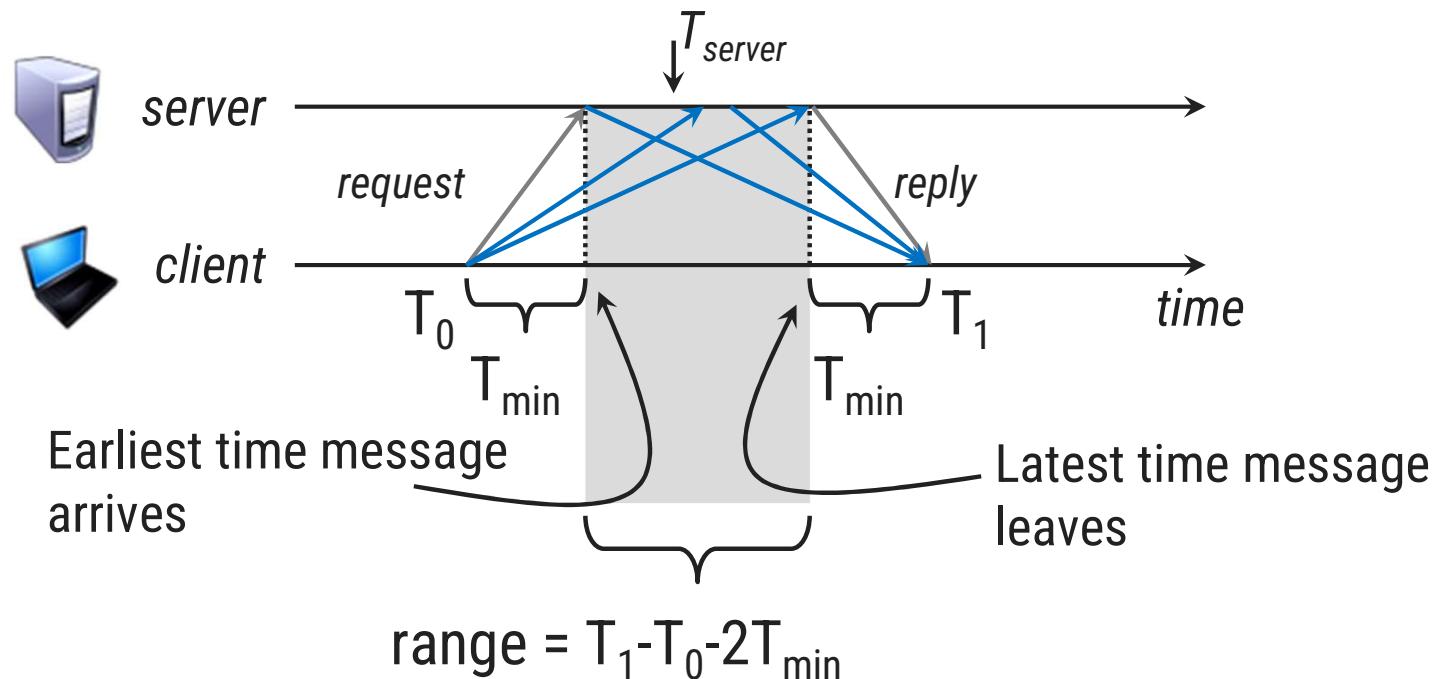
$$T_{new} = T_{server} + \text{average}((T_1 - T_0)/2)$$

Alternatively, the measurement for which the value of $(T_1 - T_0)$ is minimum is considered to be the most accurate one.

$$\min((T_1 - T_0)/2)$$

$$T_{new} = T_{server} + \min((T_1 - T_0)/2)$$

Error Bounds in Cristian's Algorithm



$$\text{Accuracy of Result} = \pm \frac{T_1 - T_0}{2} - T_{min}$$

Cristian's Algorithm

▶ Assumption about packet transmission delays:

- Cristian's algorithm assumes that the round-trip times for messages exchanged over the network are reasonably short
- The algorithm assumes that the delay for the request and response are equal
- Cristian's algorithm is intended for synchronizing computers within intranets

▶ A probabilistic approach for calculating delays:

- There is no tight bound on the maximum drift between clocks of computers

▶ Time server failure or faulty server clock:

- Faulty clock on the time server leads to inaccurate clocks in the entire DS
- Failure of the time server will render synchronization impossible

Berkeley Algorithm

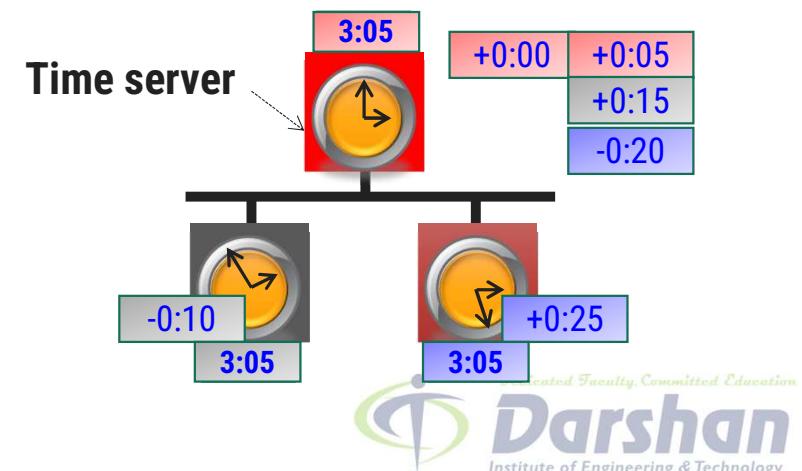
- ▶ A single time server can fail, blocking timekeeping
- ▶ The Berkeley algorithm is a **distributed algorithm for timekeeping**
- ▶ Assumes all machines have equally-accurate local clocks
- ▶ Obtains average from participating computers and synchronizes clocks to that average
- ▶ Time server **periodically 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:

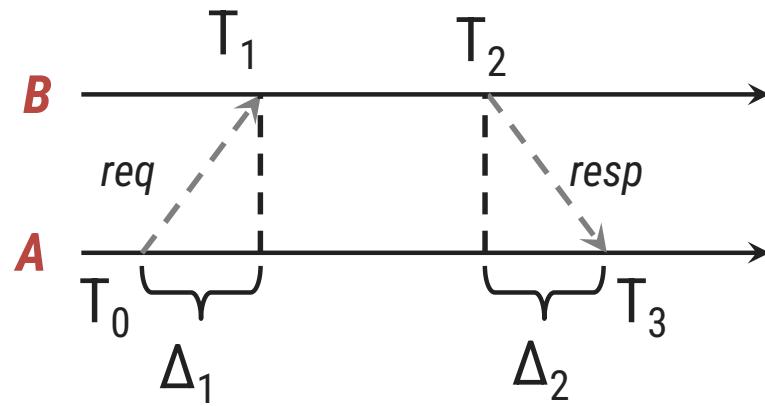
- ▶ A time server periodically (approx. once in 4 minutes) sends its time to all the computers and polls them for the time difference
- ▶ The computers compute the time difference and then reply
- ▶ The server computes an average time difference for each computer
- ▶ The server commands all the computers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

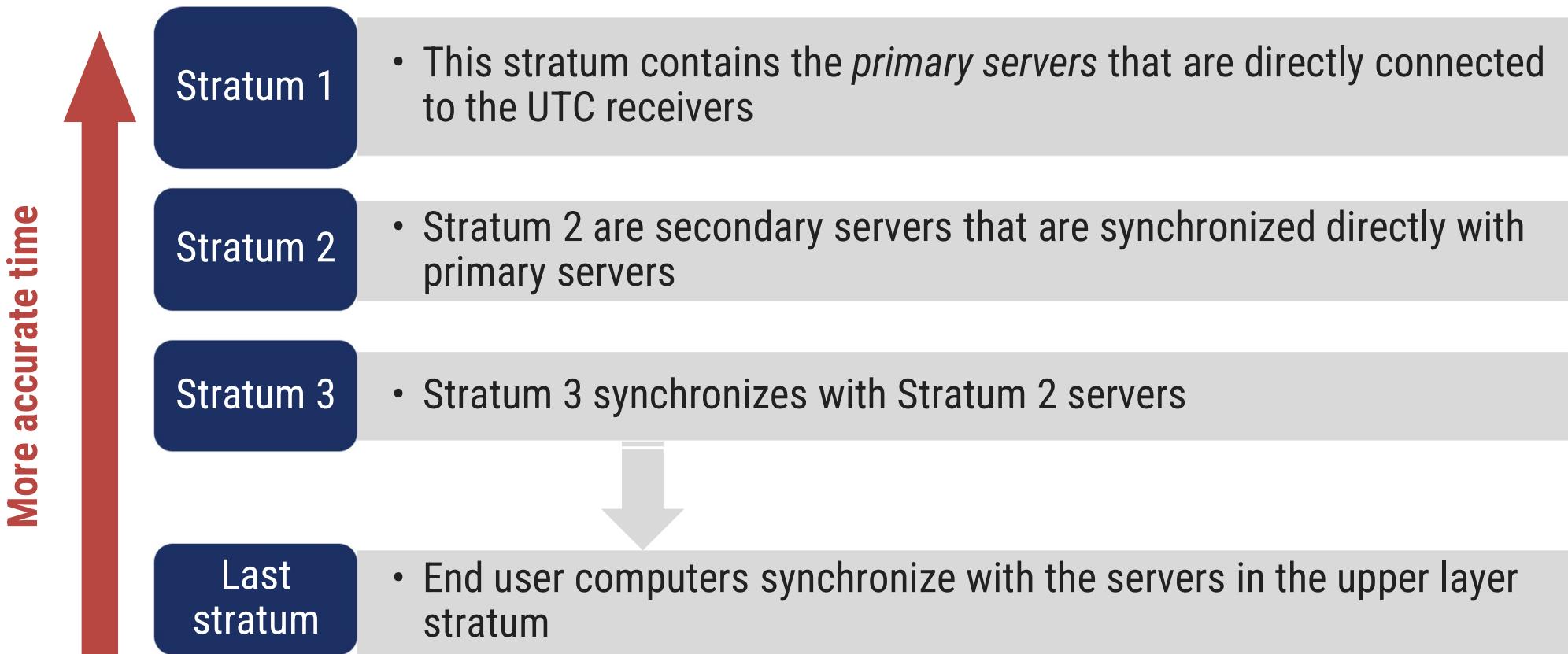
- ▶ NTP is an Application layer protocol
 - The NTP service is provided by a **network of servers located across the Internet**.
 - **Uses standard UDP Internet transport protocol**
 - Adjust system clock as **close to UTC** as possible over the Internet.
 - Enable sufficiently **frequently resynchronizations**
 - 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



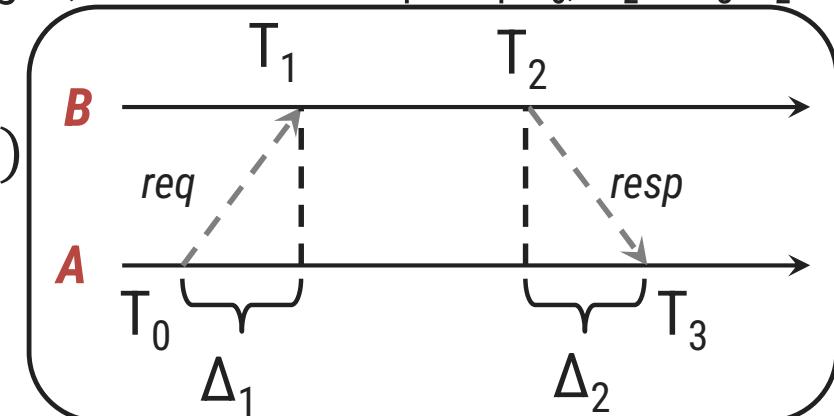
Network Time Protocol (NTP)

- 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 $a \rightarrow b$ (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

P1	P2	P3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

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

P1	P2	P3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

A diagram illustrating Lamport's Clock Algorithm. Three processes, P1, P2, and P3, are shown with their respective logical clocks. Process P1 has a current timestamp of 76. It sends a message to process P2 with a timestamp of 42, which is labeled 'm4:69'. Process P2 receives this message and updates its logical clock to 61. Process P3 has a current timestamp of 70. It receives a message from P2 with a timestamp of 69, which is labeled 'm3:60'. Process P3 updates its logical clock to 77. Arrows point from the message timestamps to the recipient's clock update.

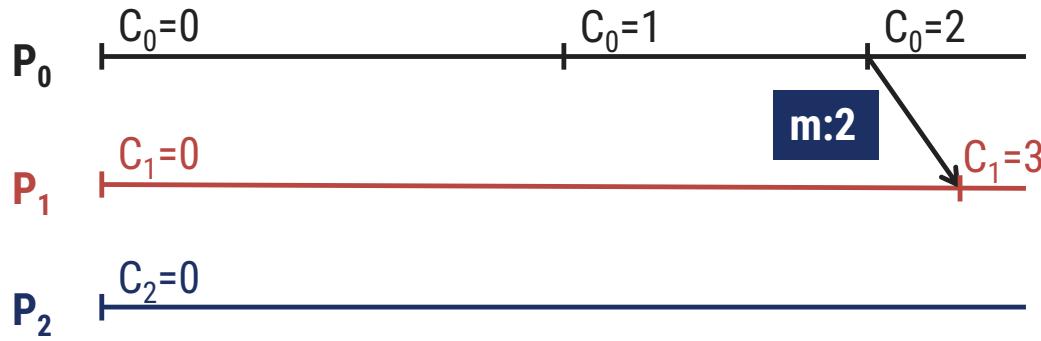
Implementation of Lamport's Clock

- ▶ 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 $\max(C_j, ts(m)) + 1$

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 → 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

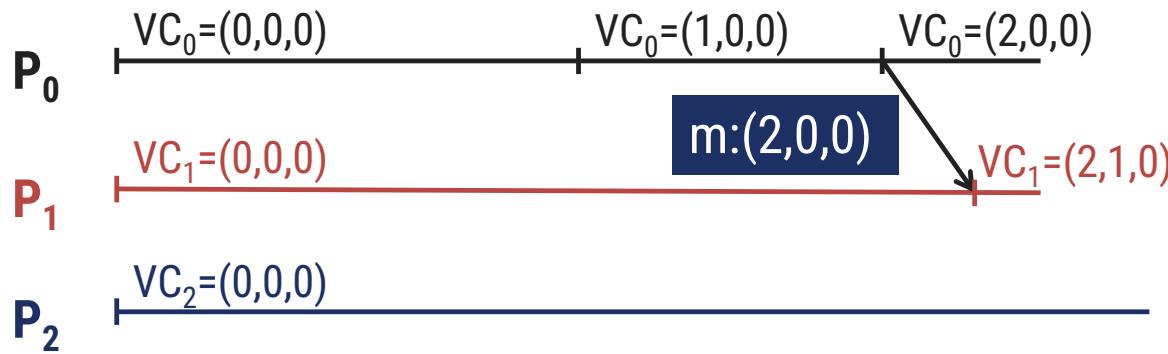
Increment VC_i whenever a new event occurs

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

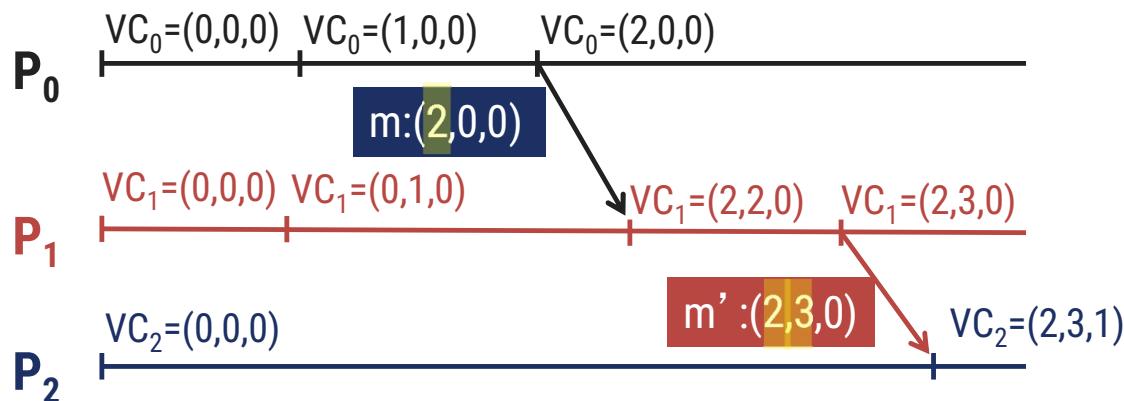
Vector Clock Update Algorithm

- ▶ 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 $VC_j[j]$



Infering Events with Vector Clocks

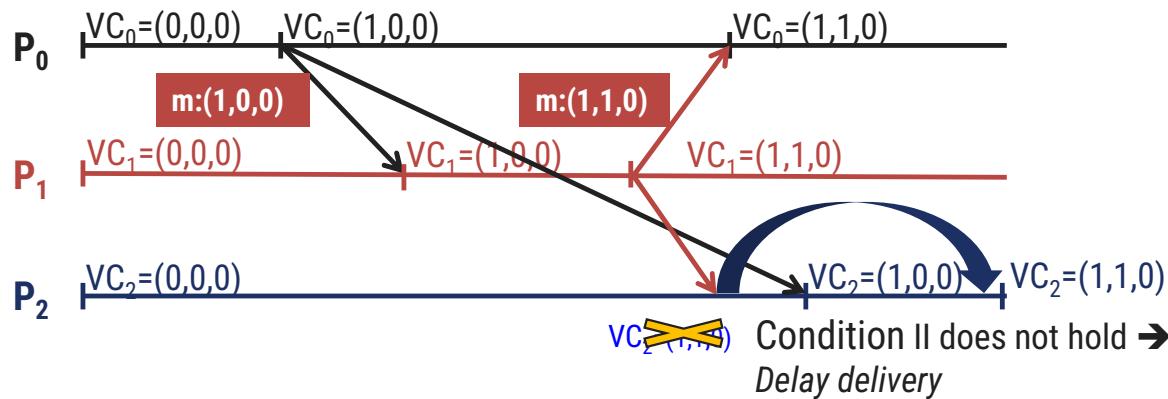
- Let a process P_i send a message m to P_j with timestamp $ts(m)$, then:
- P_j knows the number of events at the sender P_i that causally precede m
 - $(ts(m)[i] - 1)$ denotes the number of events at P_i
- P_j also knows the minimum number of events at other processes P_k that causally precede m
 -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

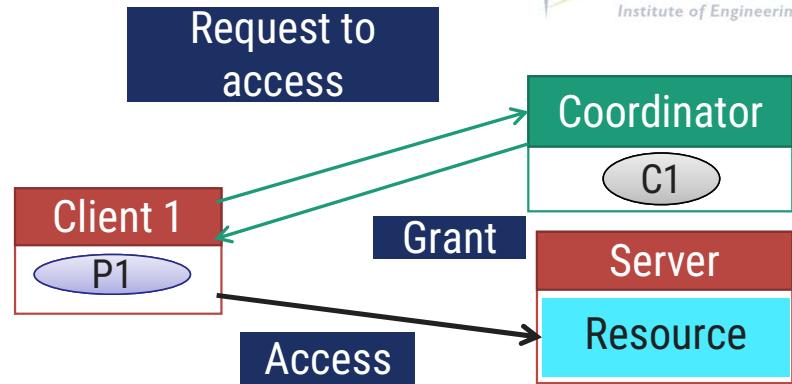
- ▶ **Mutual exclusion:** At anytime only one process should access the resource.
 - A process can get another resource first releasing its own.
- ▶ **No starvation:** if every process that is granted the resource eventually release it, every request must be eventually granted.

Types of Distributed Mutual Exclusion

Mutual exclusion algorithms are classified into two categories

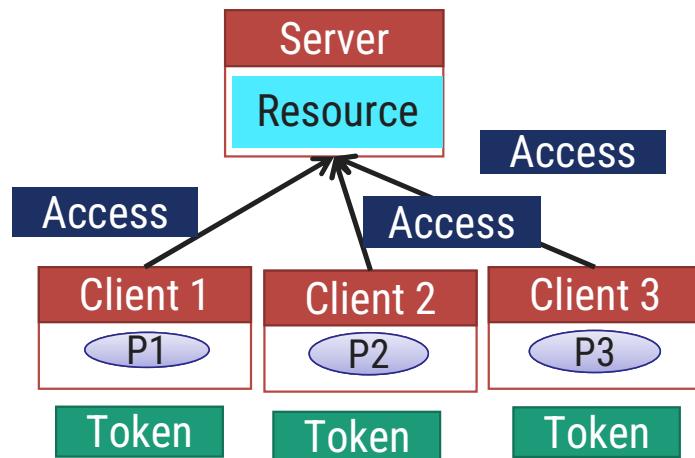
1. Permission-based Approaches

- ▶ 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



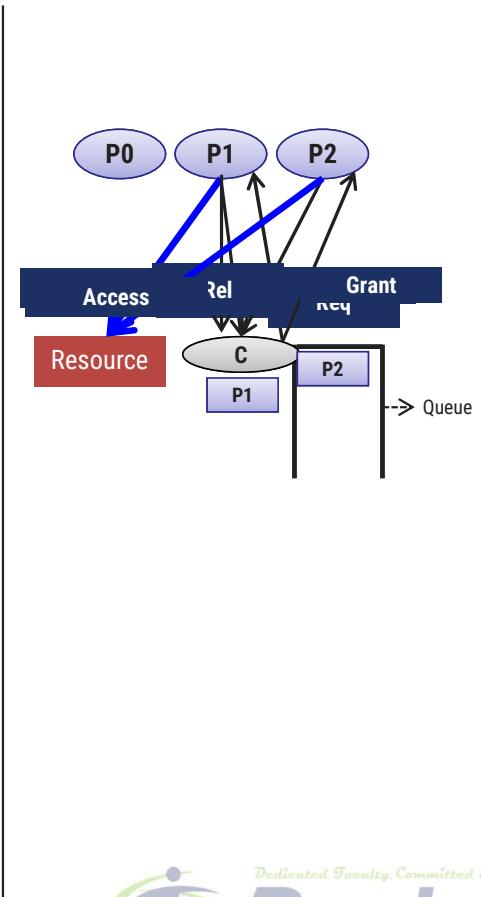
Permission-based Approaches

- ▶ 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

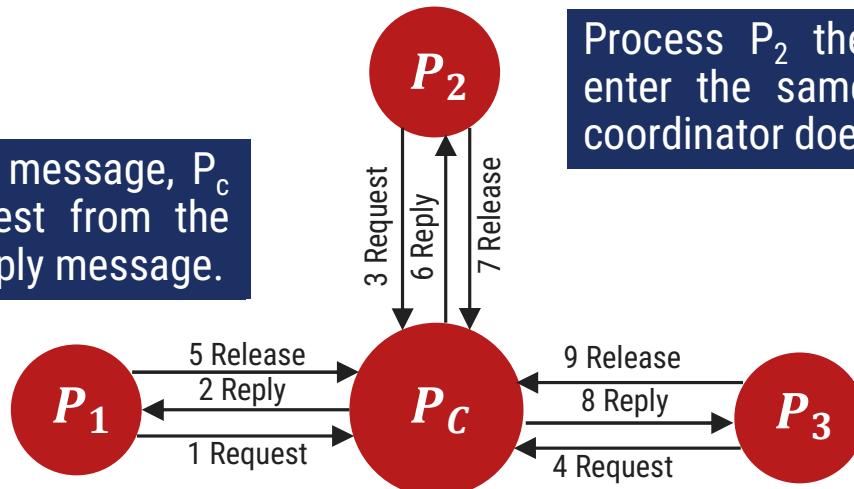


Centralized Mutual Exclusion Algorithm

- ▶ 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.

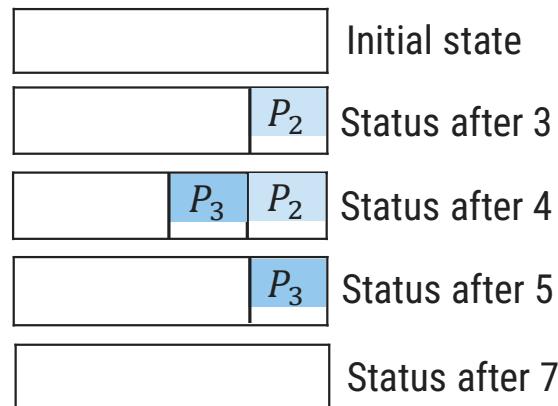
Centralized Mutual Exclusion Algorithm

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

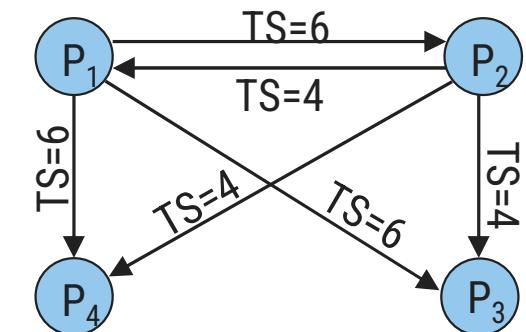
Distributed Mutual Exclusion Algorithm

- ▶ 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

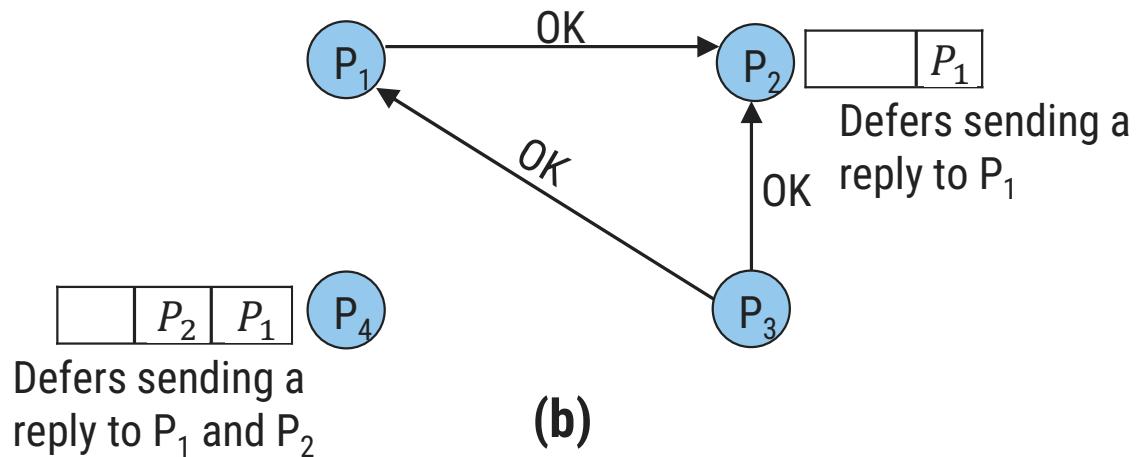
- ▶ The decision whether receiving process replies immediately to a request message or defers its reply is based on three cases:
 1. If receiver process is in its **critical section**, then it **defers its reply**.
 2. If receiver process does **not want to enter its critical section**, then it immediately **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

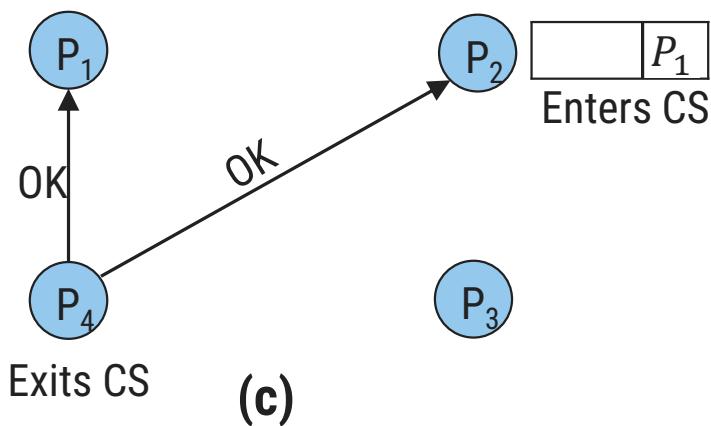


Already in CS

(a)

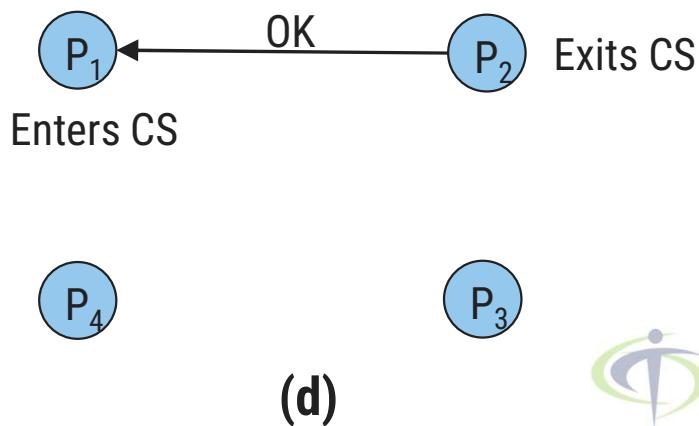


(b)



Exits CS

(c)



(d)

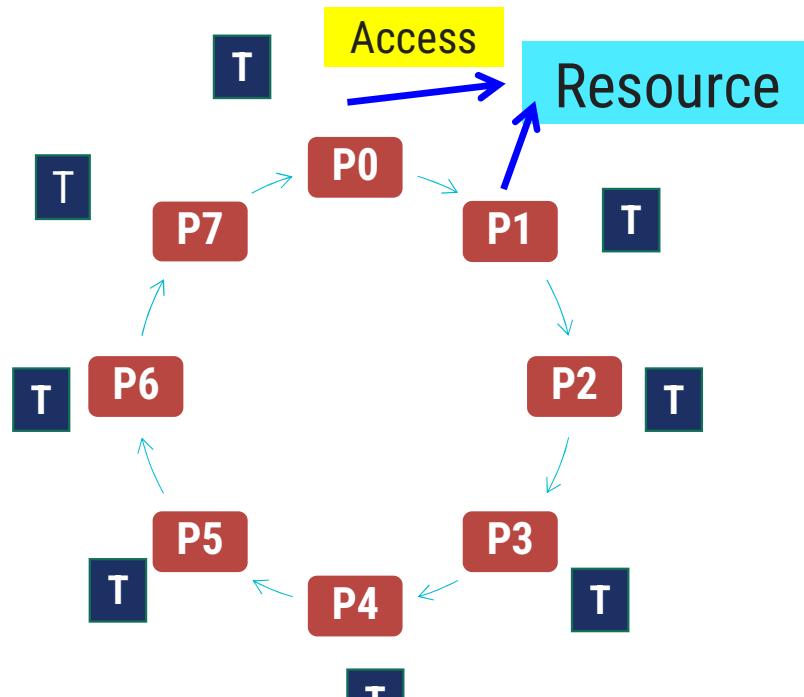
A Token Ring Algorithm

► 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

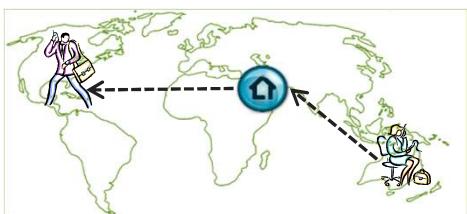
Algorithm	Number of messages required for a process to access and release the shared resource	Delay before a process can access the resource (in message times)	Problems
Centralized	3	2	<ul style="list-style-type: none"> Coordinator crashes
Decentralized	$2(n-1)$	$2(n-1)$	<ul style="list-style-type: none">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

Need for a Coordinator

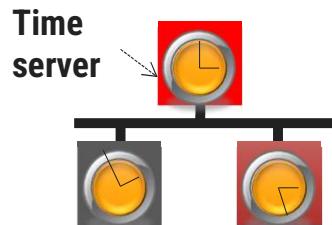
- ▶ Many algorithms used in distributed systems require a coordinator
 - For example, see the centralized mutual exclusion algorithm.
- ▶ In general, all processes in the distributed system are equally suitable for the role
- ▶ Election algorithms are designed to choose a coordinator.

Election in Distributed Systems

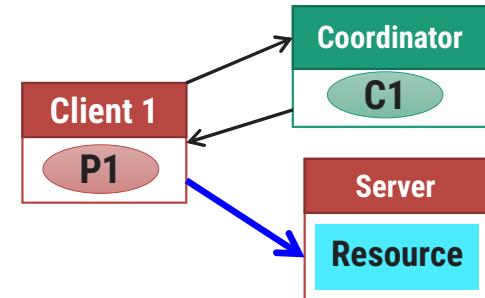
- ▶ Many distributed algorithms require one process to act as a coordinator
- ▶ Typically, it does not matter which process is elected as the coordinator



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**

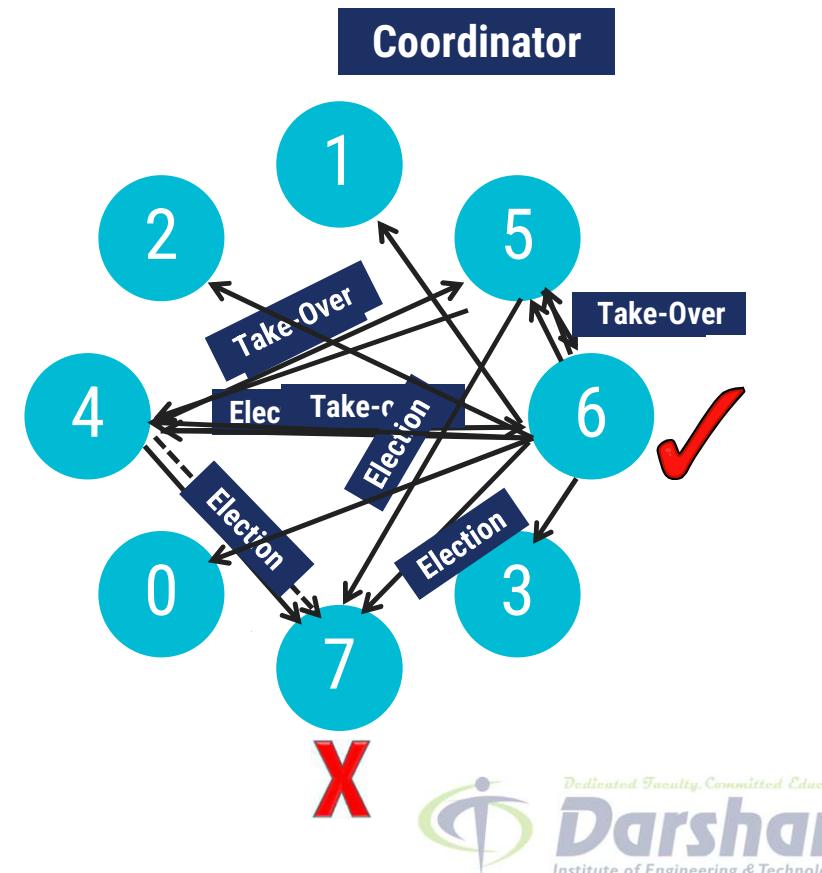
Bully Algorithm

- ▶ 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



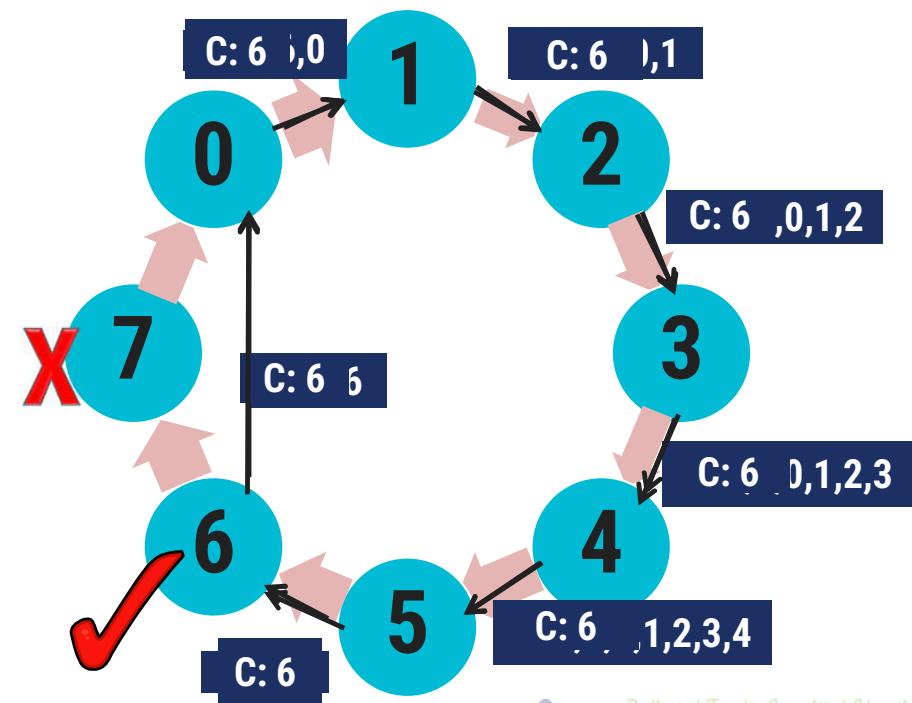
Ring Algorithm

- ▶ 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:

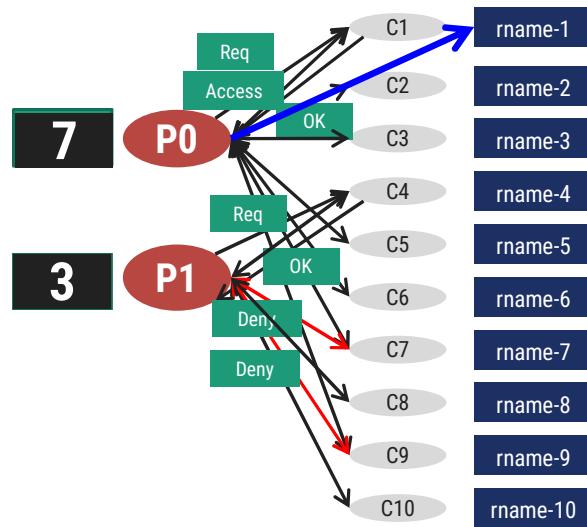
- ▶ Distributed processes are in a Distributed Hash Table (DHT) based system
- ▶ Each resource is replicated n times
 - The i^{th} replica of a resource **rname** is named as **rname-i**
- ▶ Every replica has its own coordinator for controlling access
 - The coordinator for **rname-i** is determined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

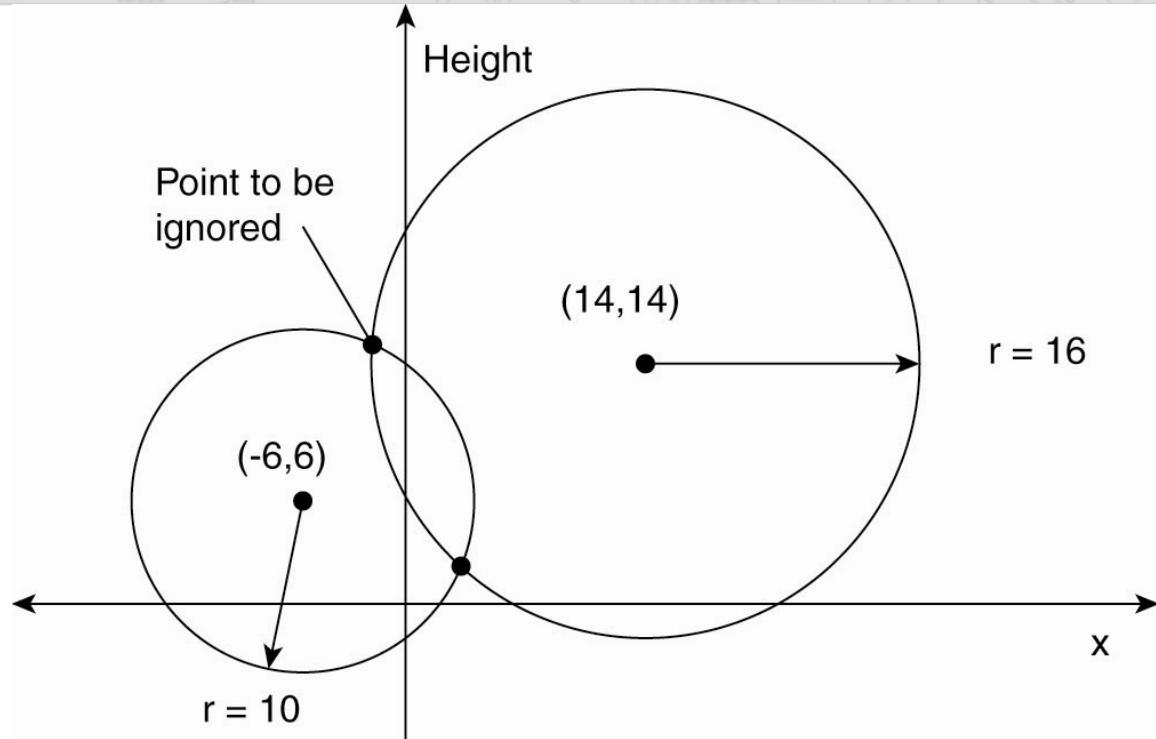
n = Number of votes gained

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

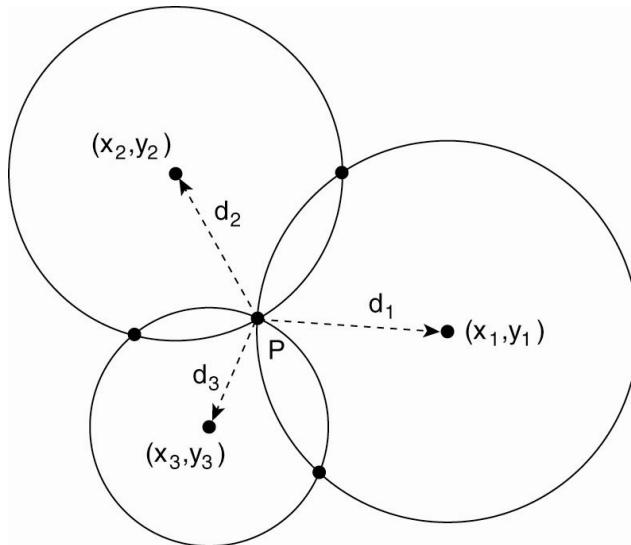
Global Positioning of Nodes

- ▶ 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

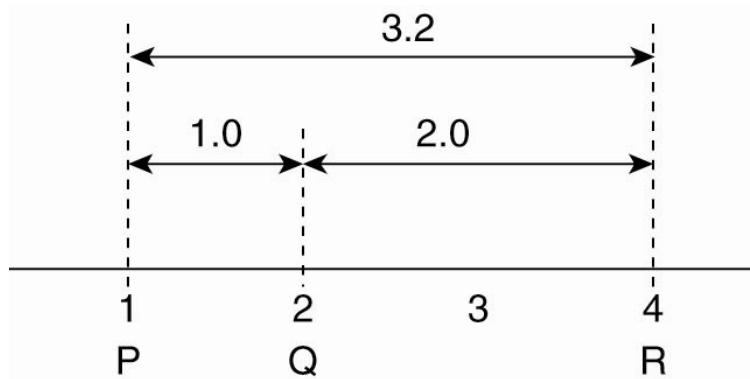
Solution

- ▶ 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 $d(b_i, b_j)$ and let each node P minimize

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



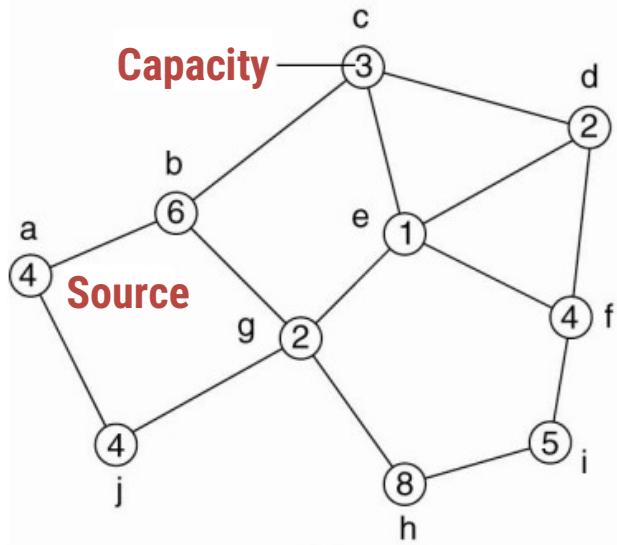
Inconsistent distance measurements in a one-dimensional space.

Elections in Wireless Environments

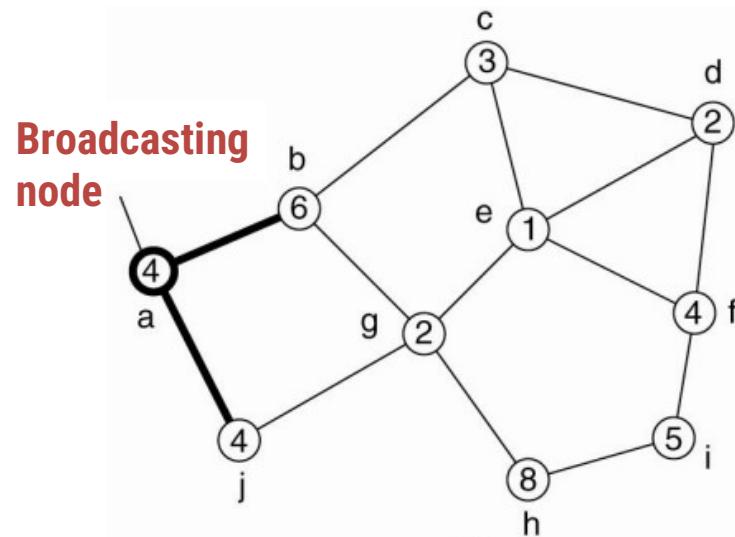
- ▶ Traditional algorithms aren't appropriate.
 - Can't assume reliable message passing or stable network configuration
- ▶ This discussion focuses on ad hoc wireless networks but ignores node mobility.
 - Nodes connect directly, no common access point, connection is short term
 - Often used in multiplayer 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



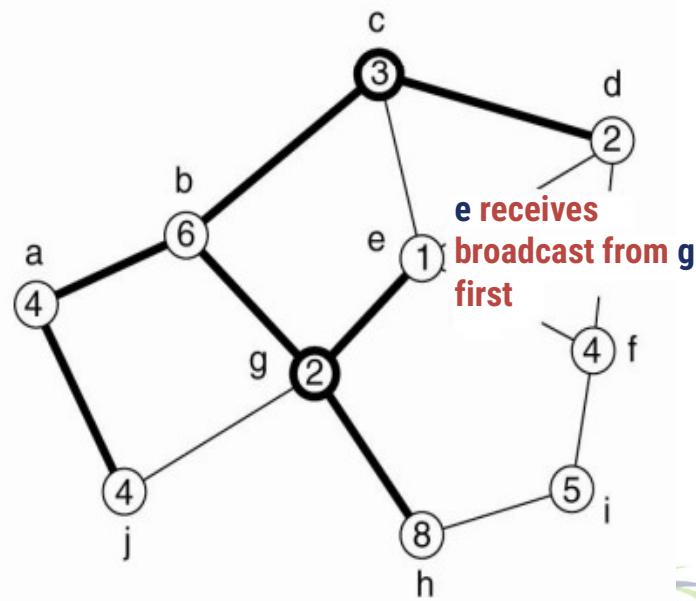
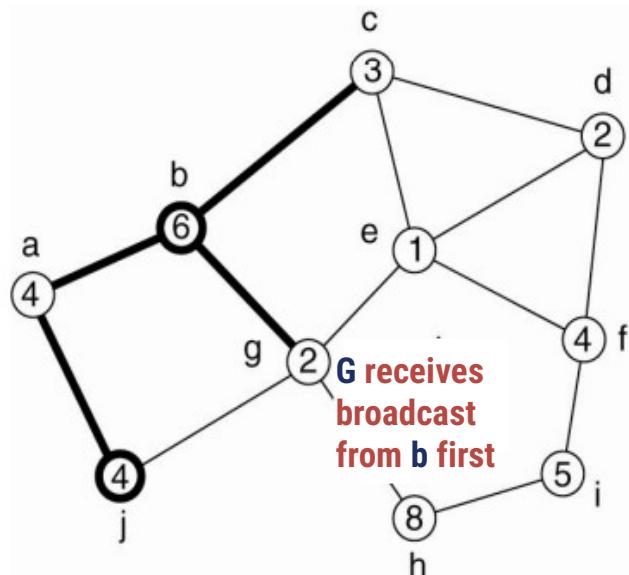
Election algorithm in a wireless network,
with node a as the source: Initial network



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

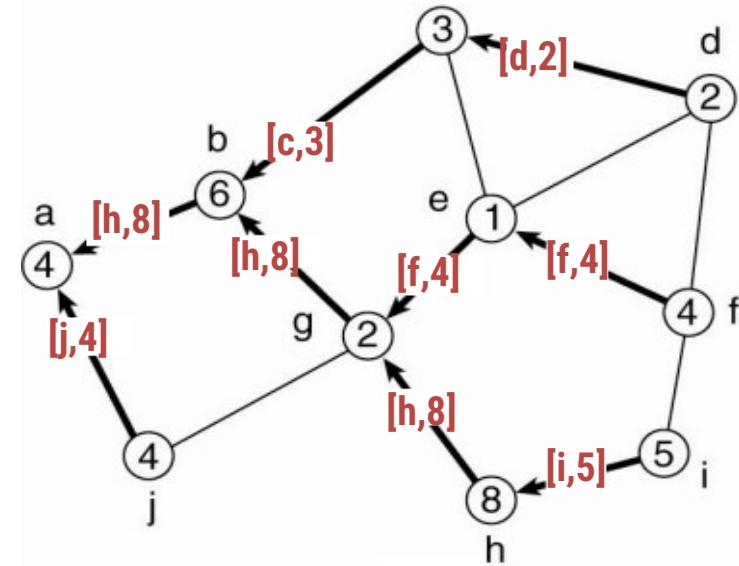
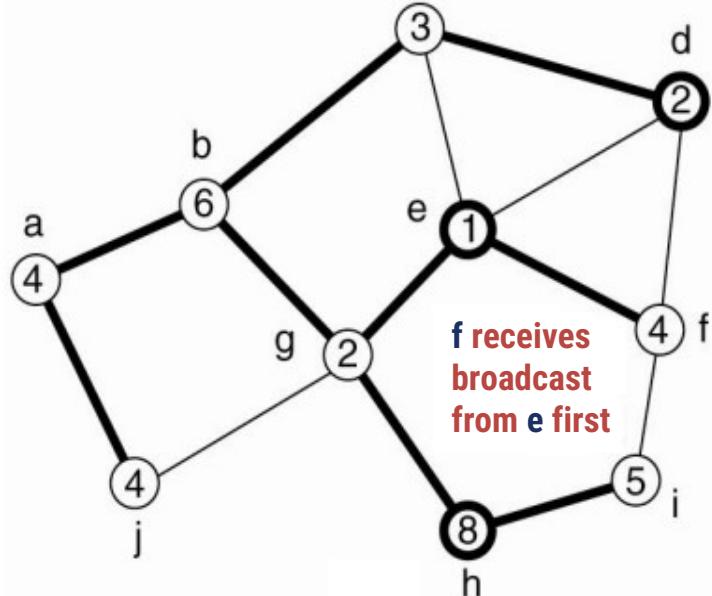
Elections in Wireless Environments

- ▶ 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



Faculty Committed Education

Elections in Large-Scale Systems

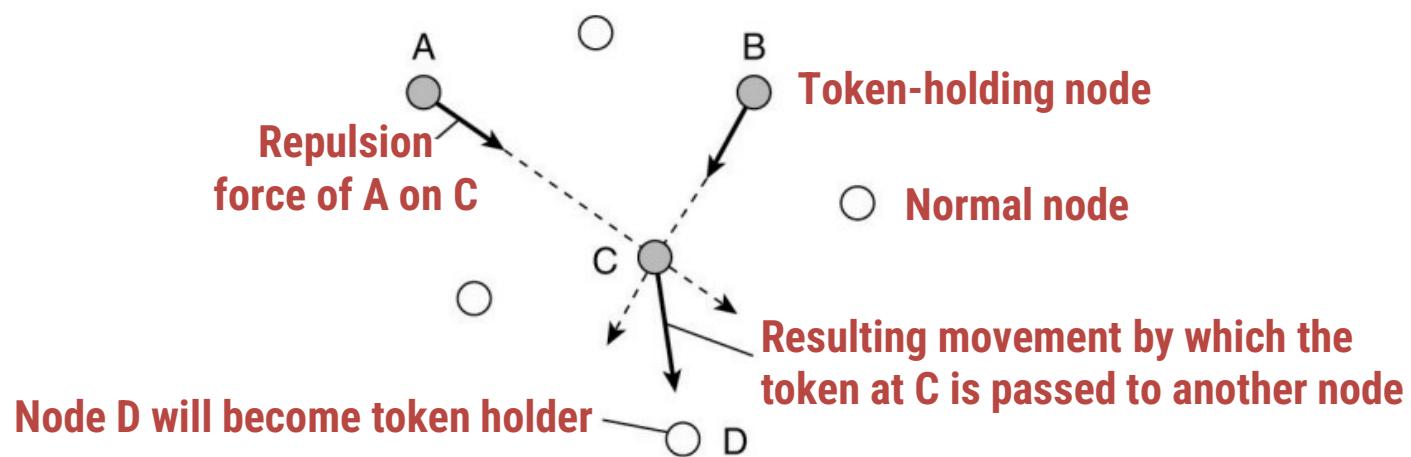
- ▶ Above discussed situations deal with selecting one coordinator as they are small distributed systems
- ▶ For a larger network, a situation arises where we need to select super peers in a peer- to-peer network.
- ▶ Requirements for superpeer selection:
 - Normal nodes should have low-latency access 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



Unit-5

Consistency, Replication and Fault Tolerance



Prof. Umesh H. Thoriya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot

✉ umesh.thoriya@darshan.ac.in
📞 9714233355



Topics to be covered

- Introduction To Replication
- Data-Centric Consistency Models
- Client-Centric Consistency Models
- Replica Management
- Consistency Protocols
- Basics of Fault Tolerance
- Process Resilience
- Reliable Client-Server Communication
- Reliable Group Communication
- Distributed Commit
- Recovery

Why Replication?

- ▶ 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

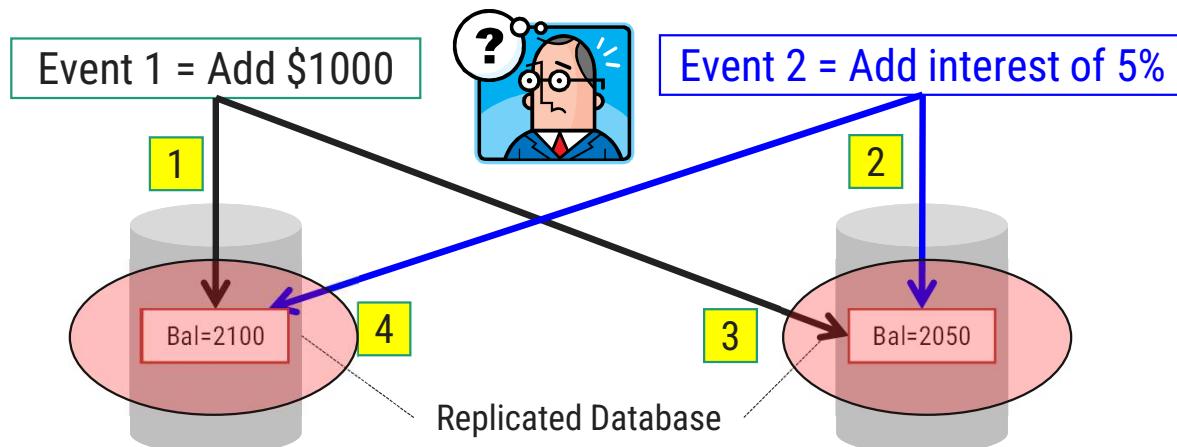
- ▶ If one copy is modified, others become inconsistent
- ▶ When and how modifications need to be carried out, determines the price of replication??

Performance and scalability

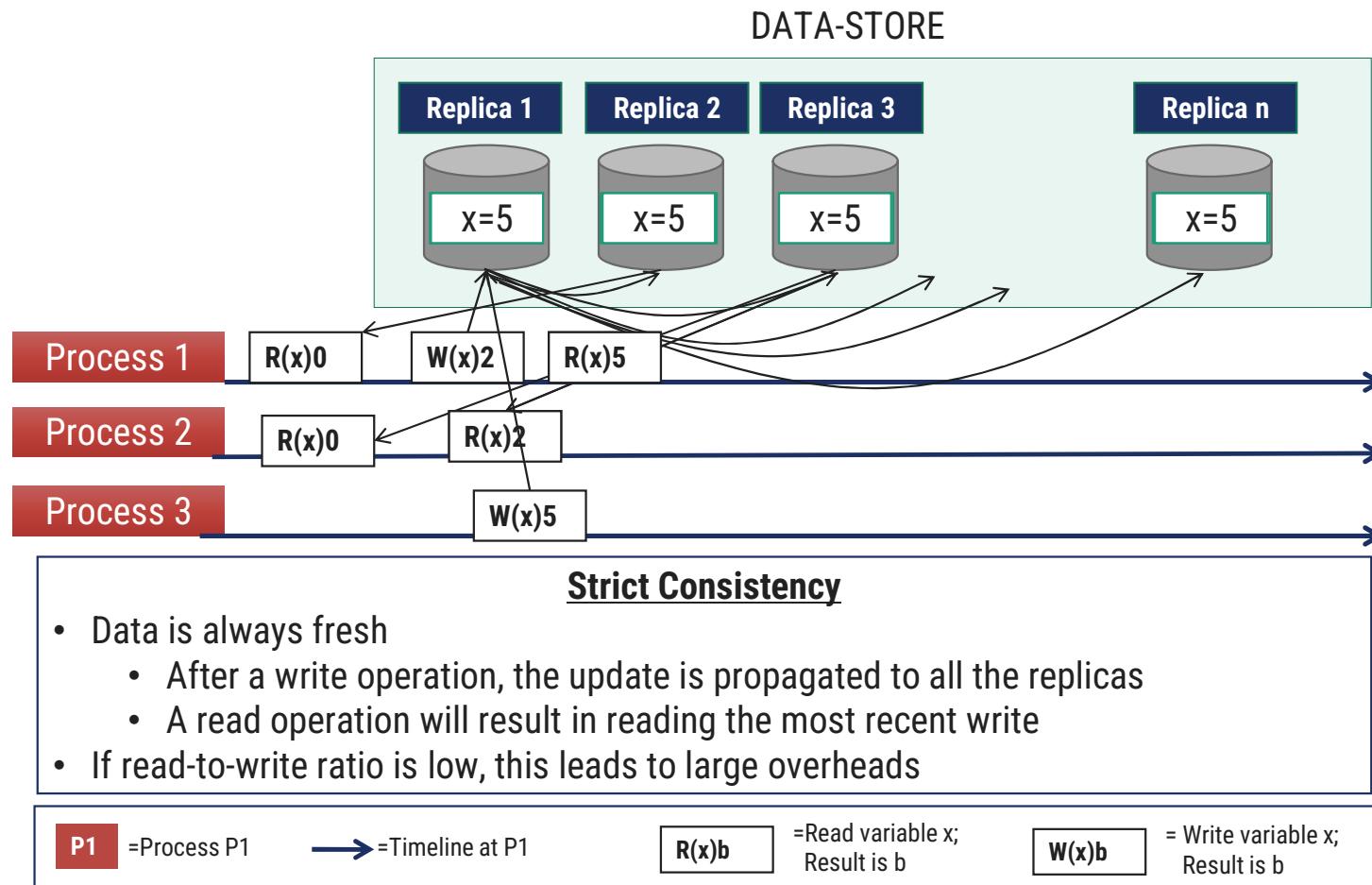
- ▶ **Main issue:** To keep replicas consistent, we generally need to ensure that all **conflicting** operations are done in the same order everywhere.
- ▶ **Conflicting operations:** From the world of transactions:
 - **Read–write conflict:** a read operation and a write operation act concurrently
 - **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?

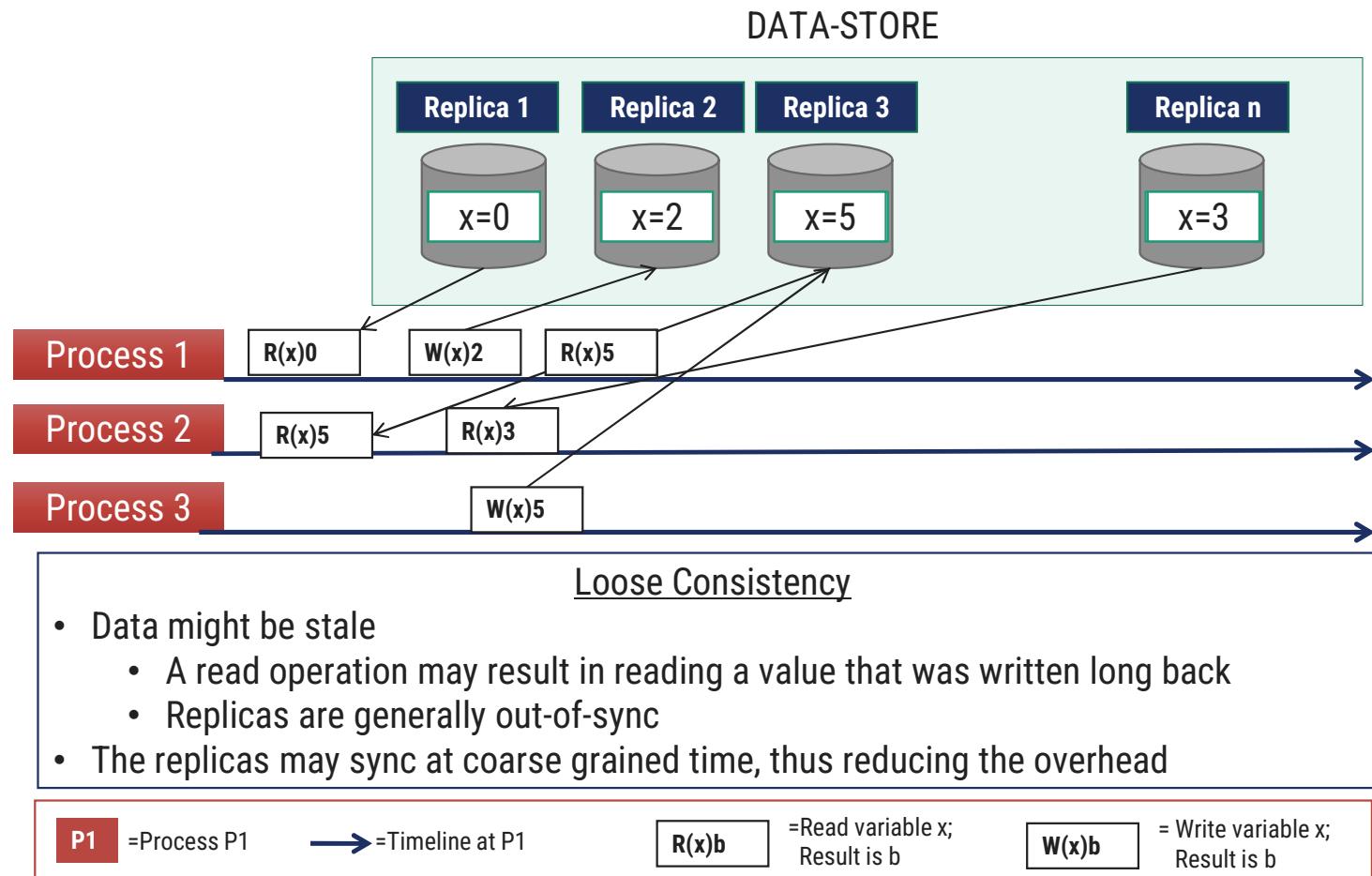
- ▶ But (server-side) replication comes with a cost, which is the necessity for maintaining consistency (or more precisely consistent ordering of updates)
- ▶ Example: A Bank Database



Maintaining Consistency of Replicated Data

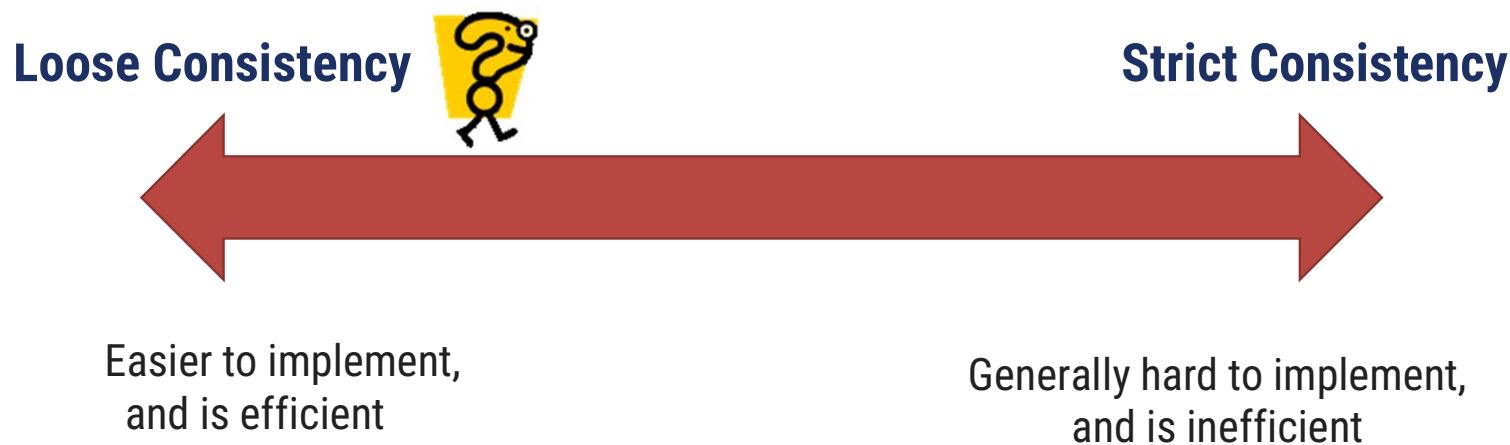


Maintaining Consistency of Replicated Data



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

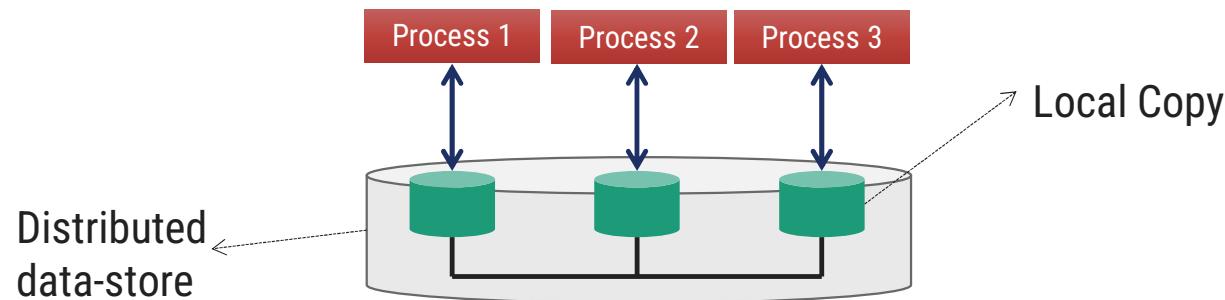
Types of Consistency Models

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

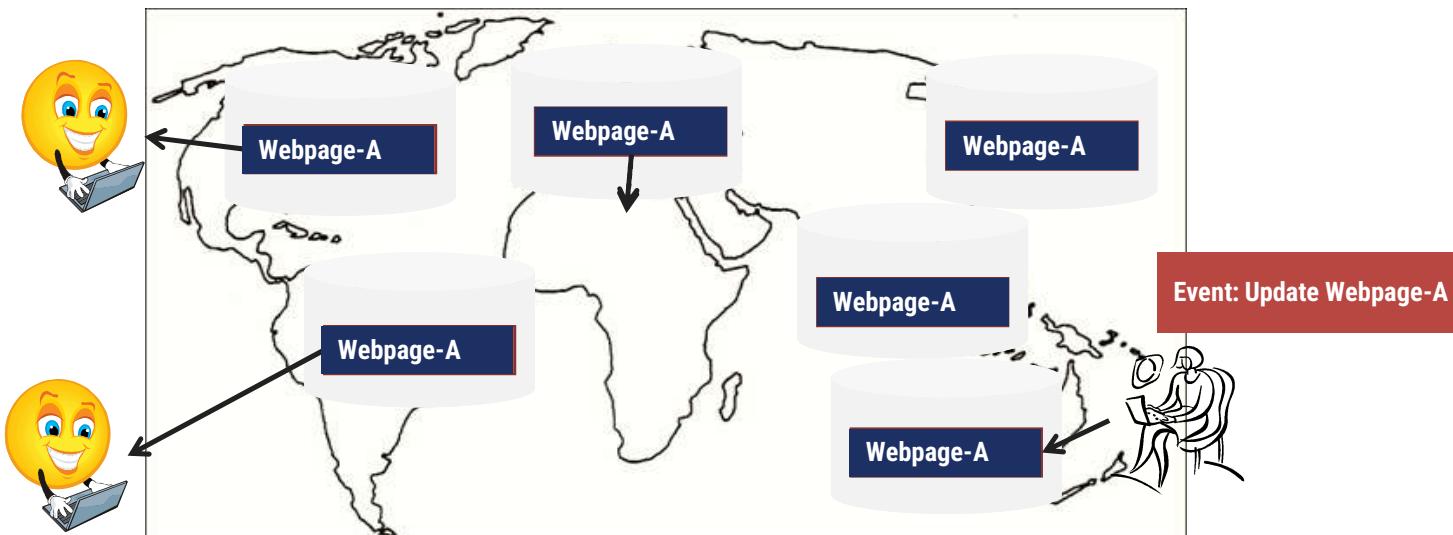
- ▶ A data-store can be read from or written to by any process in a distributed system.
- ▶ A local copy of the data-store (replica) can support “fast reads”.
- ▶ However, a **write** to a local replica needs to **be propagated to all remote replicas**.



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?

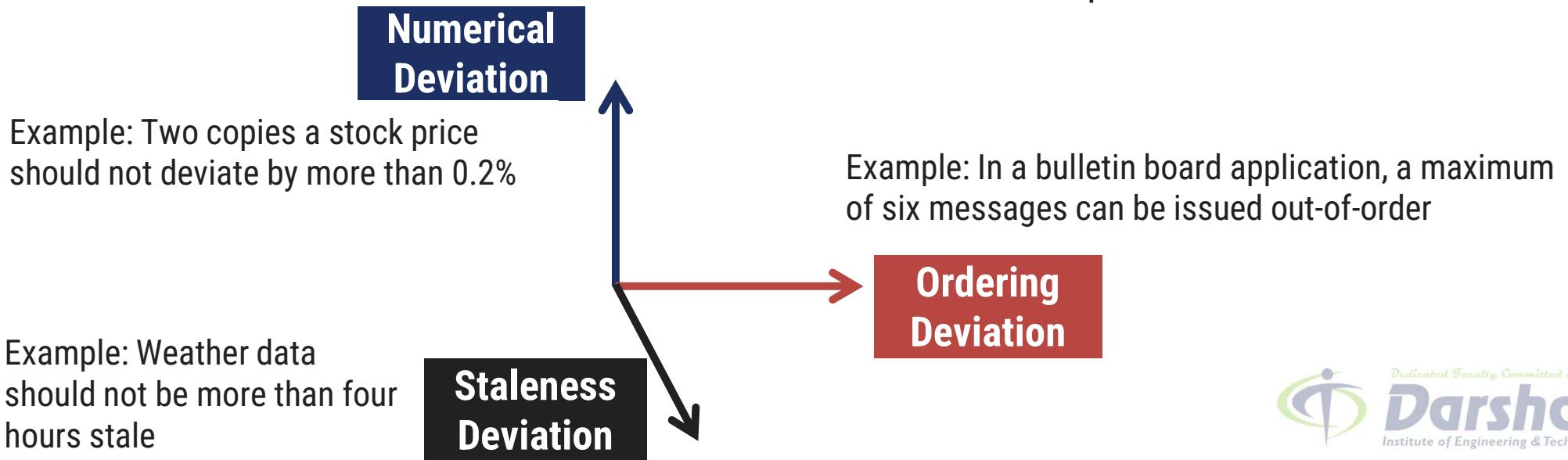


Data-Centric Consistency Model is too strict when

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

Continuous Consistency

- ▶ Continuous Consistency Model is used to measure inconsistencies and express what inconsistencies can be expected in the system
- ▶ Level of consistency is defined over three independent axes:
 1. **Numerical Deviation:** Deviation in the numerical values between replicas
 2. **Order Deviation:** Deviation with respect to the ordering of update operations
 3. **Staleness Deviation:** Deviation in the staleness between replicas



Continuous Consistency

- ▶ Consistency unit (Conit) specifies the data unit over which consistency is measured
 - For example, conit can be defined as a record representing a single stock
- ▶ Level of consistency is measured by each replica along the three dimensions
- ▶ **Numerical Deviation** : For a given replica R, how many updates at other replicas are not yet seen at R? What is the effect of the non-propagated updates on local Conit values?
- ▶ **Order Deviation** : For a given replica R, how many local updates are not propagated to other replicas?
- ▶ **Staleness Deviation** : For a given replica R, how long has it been since updates were propagated?

Example of Conit and Consistency Measures

Order Deviation at a replica R is the number of operations in R that are not present at the other replicas

Numerical Deviation at replica R is defined as $n(w)$, where

n = # of operations at other replicas that are not yet seen by R,

w = weight of the deviation

= max(update amount of all variables in a Conit)

Replica A	
x=6; y=3	
Operation	Result
<5,B>	x+=2
<8,A>	y+=2
<12,A>	y+=1
<14,A>	x=y*2
	x=2
	y=2
	y=3
	x=4

Vector clock A = (15, 5)

Order deviation = 3

Numerical deviation = (1, 5)

Replica B	
X=2; y=5	
Operation	Result
<5,B>	x+=2
<10,B>	y+=5
	x=2
	y=5

Vector clock B = (0, 11)

Order deviation = 2

Numerical deviation = (3, 6)

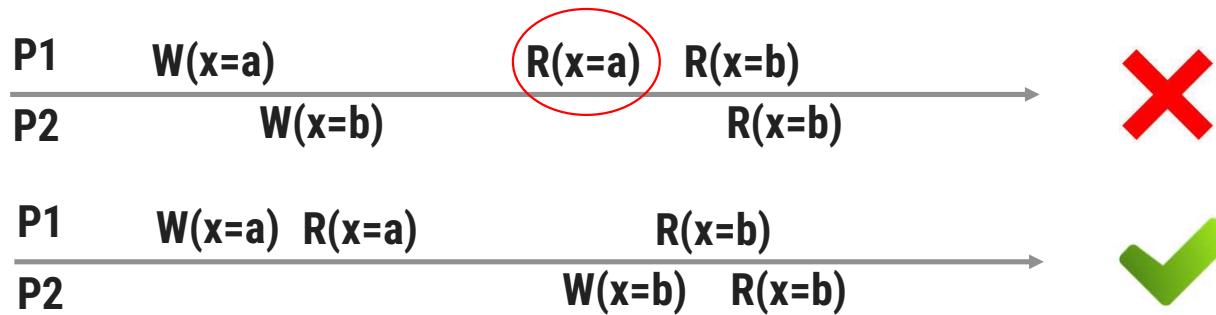
$<5,B>$ = Operation performed at B when the vector clock was 5 $<m,n>$ = Uncommitted operation $<m,n>$ = Committed operation xy = A Conit

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

- ▶ 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

- ▶ 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

P1 W(x=a)

P2 W(x=b)

P3 R(x=b) R(x=a)

P4 R(x=b) R(x=a)



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.

P1 W(x=a)

P2 W(x=b)

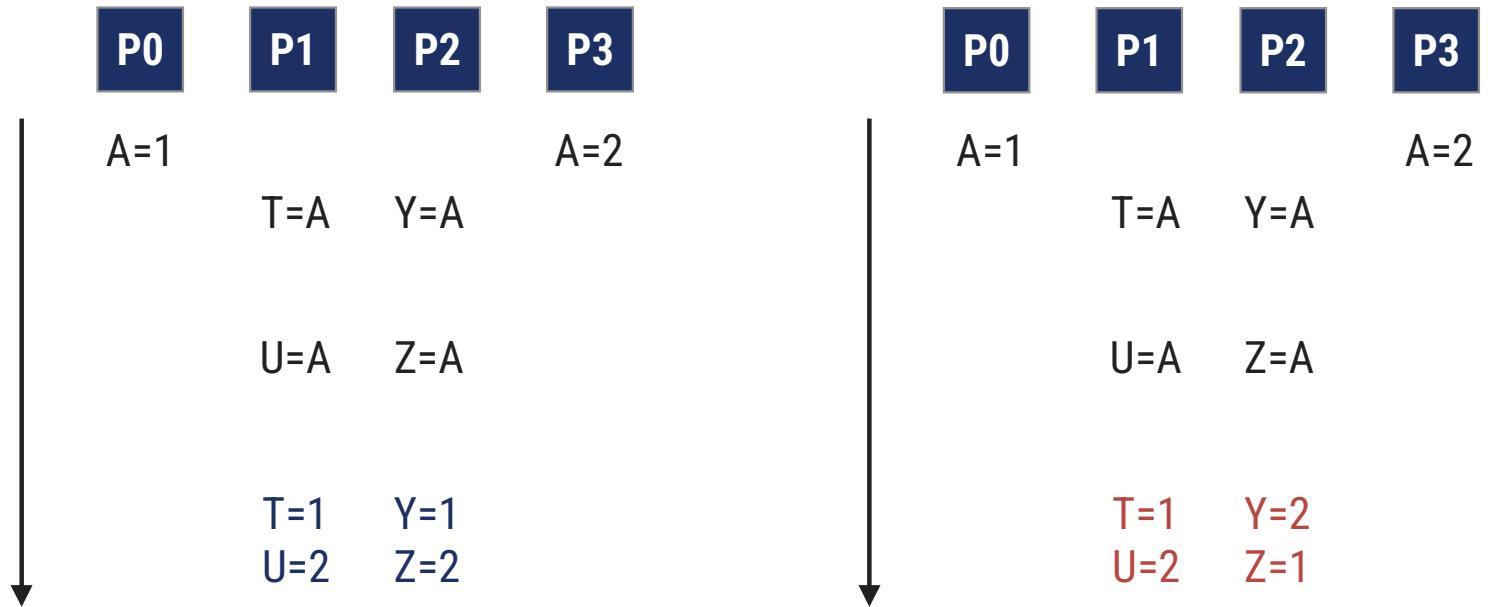
P3 R(x=b) R(x=a)

P4 R(x=a) R(x=b)



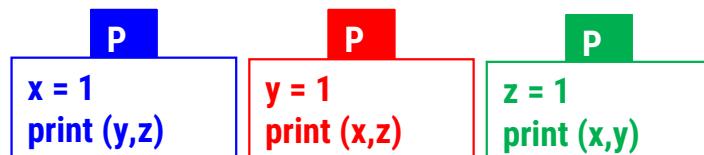
1. Violates sequential consistency - not all processes see the same interleaving of write operations.
2. To process P3, it appears as if the data item has first been changed to b and later to a.
3. BUT, P4 will conclude that the final value is b.

Sequential Consistency



Sequential Consistency

- ▶ Consider three processes P1, P2 and P3 executing multiple instructions on three shared variables x, y and z
- ▶ Assume that x, y and z are set to zero at start



- ▶ There are many valid sequences in which operations can be executed, respecting sequential consistency (or program order)
- ▶ How can a program identify the wrong sequence among the following sequences:

	<code>x = 1</code> <code>print (y,z)</code> <code>y = 1</code> <code>print (x,z)</code> <code>z = 1</code> <code>print (x,y)</code>	<code>x = 1</code> <code>y = 1</code> <code>print (x,z)</code> <code>print (y,z)</code> <code>z = 1</code> <code>print (x,y)</code>	<code>y = 1</code> <code>z = 1</code> <code>print (x,y)</code> <code>print (x,z)</code> <code>x = 1</code> <code>print (y,z)</code>	<code>y = 1</code> <code>x = 1</code> <code>z = 1</code> <code>print (x,z)</code> <code>print (y,z)</code> <code>print (x,y)</code>
Output	001011	101011	010111	111111
Signature	001011	101011	110101	111111

A red X is placed over the Signature row for the sequence 110101.

Causal Consistency

- ▶ 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

P1 $W(x=a)$

P2 $R(x=a)$ $W(x=b)$

P3 $R(x=b)$ $R(x=a)$

P4 $R(x=a)$ $R(x=b)$



- $W(x=b)$ 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.

P1 $W(x=a)$

P2 $W(x=b)$

P3 $R(x=b)$ $R(x=a)$

P4 $R(x=a)$ $R(x=b)$



- ▶ 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.

Causal Consistency

wall-clock time

P1: $w(x)a$

P2: $w(x)b$

P3: $r(x)b$ $r(x)a$

P4: $r(x)a$ $r(x)b$

wall-clock time

P1: $w(x)a$ $w(x)c$

P2: $w(x)b$

P3: $r(x)c$ $r(x)a$

P4: $r(x)a$ $r(x)b$

Only per-process ordering restrictions:

$w(x)b < r(x)b$; $r(x)b < r(x)a$;

$w(x)a \parallel w(x)b$, hence they can be seen in orders by processes

Having read c ($r(x)c$), P3 must continue to read c or some newer value (perhaps b), but can't go back to a , because $w(x)c$ was conditional upon $w(x)a$ having finished

This wasn't sequentially



FIFO Consistency

- ▶ 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 (w_{11}) and (w_{12}) are write operations performed by p_1 in that order and $(w_{21}),(w_{22})$ by p_2 . A process p_3 can see them as $[(w_{11},w_{12}),(w_{21},w_{22})]$ while p_4 can view them as $[(w_{21},w_{22}),(w_{11},w_{12})]$.

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)



Grouping operations

- ▶ The read/write operation level of granularity used in sequential and causal consistency originates from shared-memory multiprocessor systems
- ▶ Applications have coarser level of granularity enforced by mutual exclusion and transactions
 - Group of read and write operations is bracketed by ENTER_CS and LEAVE_CS operations (CS = critical section) into atomic unit
 - Data operated on within a critical section is protected against concurrent access
- ▶ **Convention:** when a process enters its critical section it should acquire the relevant synchronization variables, and likewise when it leaves the critical section, it releases these variables.
- ▶ **Critical section:** a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
- ▶ **Synchronization variables:** are synchronization primitives that are used to coordinate the execution of processes based on asynchronous events.

Grouping operations

- ▶ Semantics of **ENTER_CS** and **LEAVE_CS** are formulated in form of **synchronization variables**
 - Process entering critical section **acquires** synchronization variables
 - Process leaving critical section **releases** synchronization variables

The following three criteria must be met

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is Allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

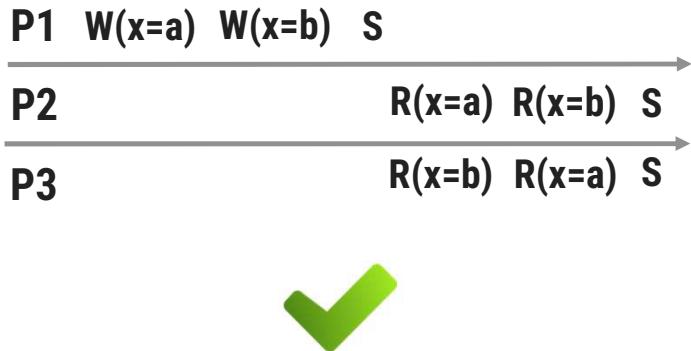
Grouping operations

- ▶ Consistency models enforced with synchronization variables
- ▶ **Weak consistency**: shared data can be counted to be consistent only after a synchronization is done
- ▶ **Release consistency**: shared data are made consistent when a critical region is exited
- ▶ **Entry consistency**: shared data pertaining to a critical region are made consistent when a critical region is entered

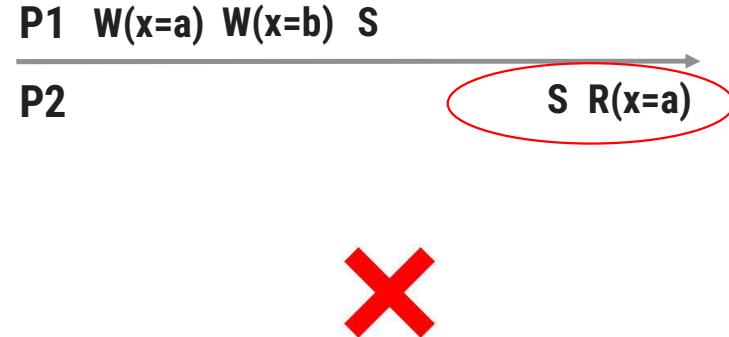
Weak Consistency Model

- ▶ Changes in memory can be made after a set of changes has happened (Example: critical section).
- ▶ Isolated access to variable is usually rare, usually there will be several accesses and then none at all.
- ▶ Difficulty is the system would not know when to show the changes.
- ▶ Application programmers can take care of this through a synchronization variable.
- ▶ For supporting weak consistency, the following requirements is must:
 1. Accesses to synchronization variables associated with a data-store are sequentially consistent.
 2. No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.
 3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

Weak Consistency Model



A valid sequence of events for weak consistency. This is because P2 and P3 have yet to synchronize, so there's no guarantees about the value in 'x'.



An invalid sequence for weak consistency. P2 has synchronized, so it cannot read 'a' from 'x' – it should be getting 'b'.

Release Consistency Model

- ▶ Enhancement of weak consistency model.
- ▶ Use of two synchronization variables
 1. **Acquire** (used to tell the system it is entering CR)
 2. **Release** (used to tell the system it has just exited CR)
- ▶ Acquire results in propagating changes made by other nodes to process's node.
- ▶ Release results in propagating changes made by the process to other nodes.
- ▶ Release consistency can be viewed as synchronization mechanism based on barriers instead of critical sections.
- ▶ Barrier defines the end of a phase of execution of a group of concurrently executing processes.
- ▶ Barrier can be implemented by using a centralized barrier server.

Release Consistency Model

P1 Acq(L) W(x=a) W(x=b) Rel(L)

P2 Acq(L) R(x=b) Rel(L)

P3

R(x=a)



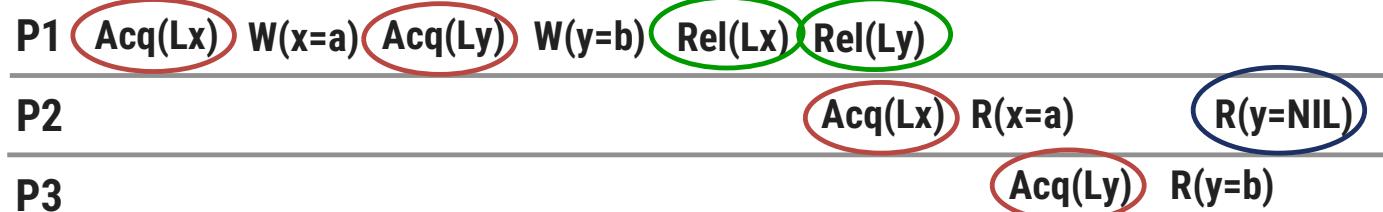
► A distributed data-store is “Release Consistent” if it obeys the following rules:

1. Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
2. Before a release is allowed to be performed, all previous reads and writes by the process must have completed.
3. Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).

Entry Consistency Model

- ▶ With release consistency, all local updates are made available to all copies during the release of the lock
- ▶ With entry consistency, each individual shared data item is associated with some **synchronization variable** (e.g. **lock or barrier**)
- ▶ When acquiring the synchronization variable, the **most recent values** of its associated shared data item must be fetched
- ▶ The release consistency affects all shared data, entry consistency affects only those associated with a synchronization variable.

Entry Consistency Model



- ▶ A different twist on things is “Entry Consistency”. Acquire and release are still used, and the data-store meets the following conditions:
 1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
 2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
 3. After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

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.

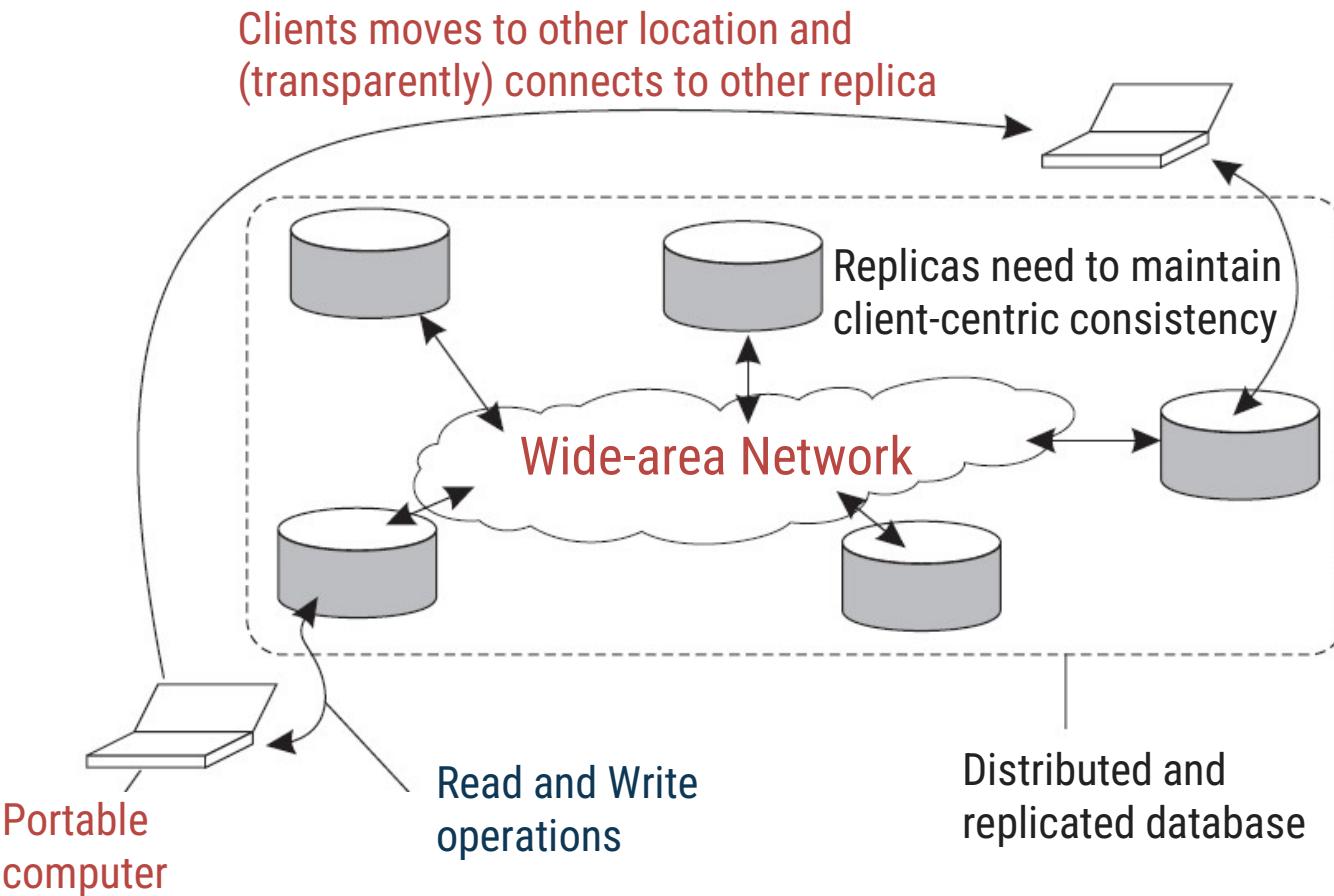
Client-Centric Consistency Models

- ▶ 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 not 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

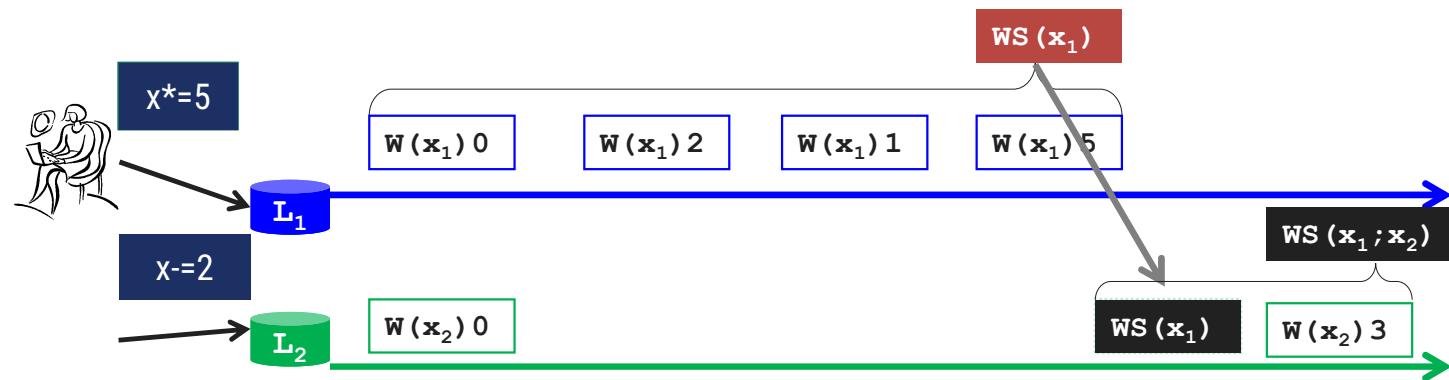
- ▶ In Systems that tolerate high degree of inconsistency, if no updates take place for a long time all replicas will gradually and eventually become consistent. This form of consistency is called eventual consistency.
- ▶ Eventual consistency only requires those updates that guarantee propagation to 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 x replicated on two replicas. Let x_i be the local copy of a data 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

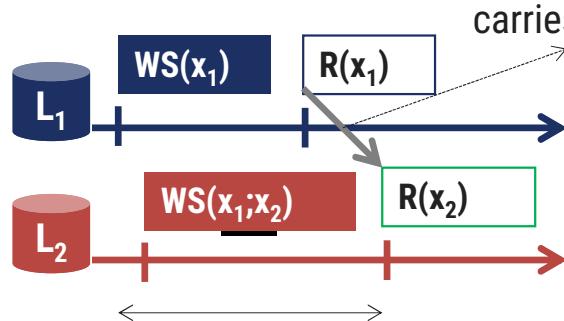
Four types of client-centric consistency models

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

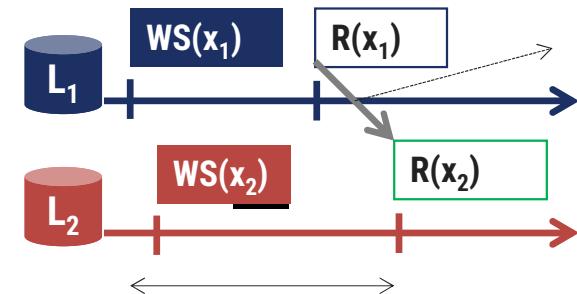


Order in which client process carries out the operations

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



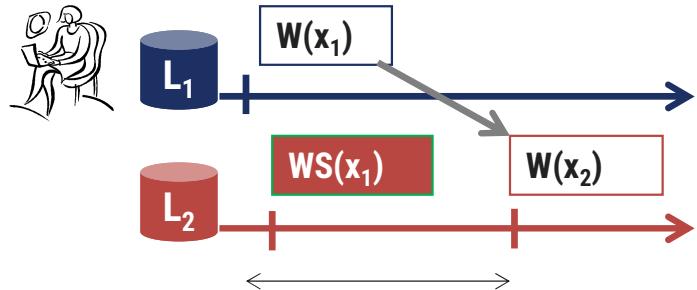
A data store that does not provide monotonic reads.

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).

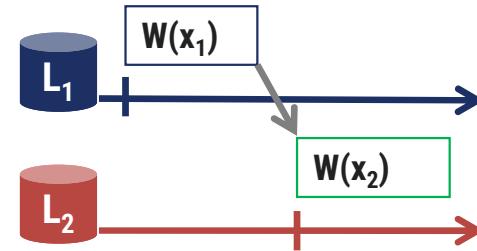
Monotonic Writes

- ▶ 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



$W(x_2)$ operation should be performed only after the result of $W(x_1)$ has been updated at L_2



The data-store does not provide monotonic write consistency

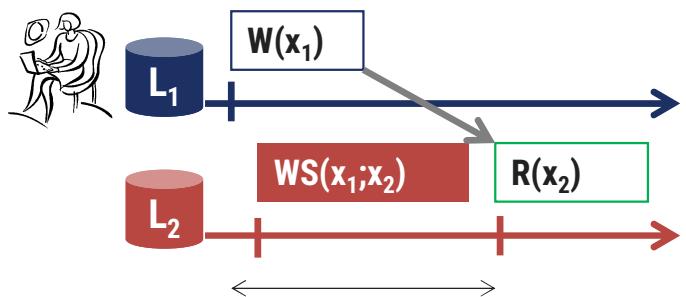
1. Process P performs a write operation on x at local copy L1, presented as the operation $W(x_1)$.
2. Later, P performs another write operation on x, but this time at L2, shown as $W(x_2)$.
3. To ensure monotonic-write consistency, the previous write operation at L1 must have been propagated to L2.
4. This explains operation $W(x_1)$ at L2, and why it takes place before $W(x_2)$.

1. Situation in which monotonic-write consistency is not guaranteed.
2. Missing is the propagation of $W(x_1)$ 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 $W(x_1)$ completed at L1.

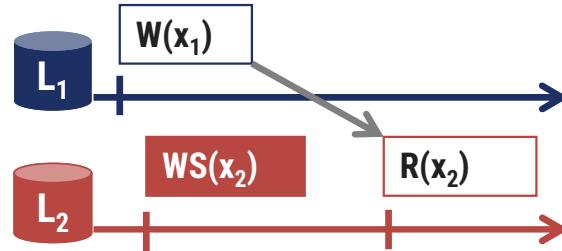
Read Your Writes

- ▶ 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



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

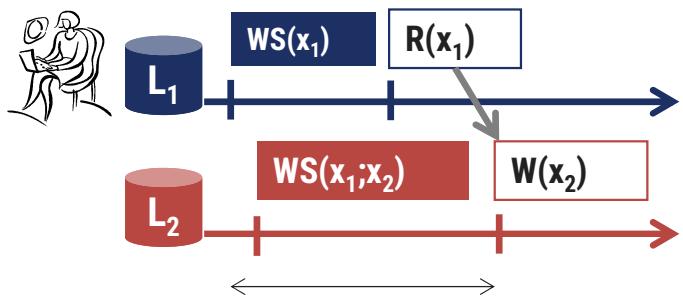
1. Process P performed a write operation W(x₁) and later a read operation at a different local copy.
2. Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation.
3. This is expressed by WS (x₁; x₂), which states that W (x₁) is part of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

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.

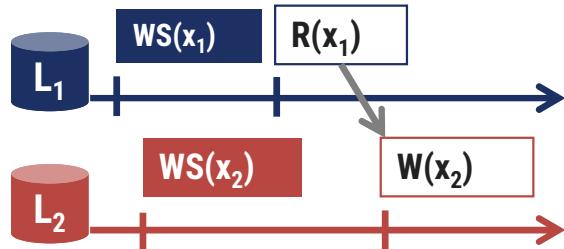
Writes Follow Reads

- ▶ 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



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

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.

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 that 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

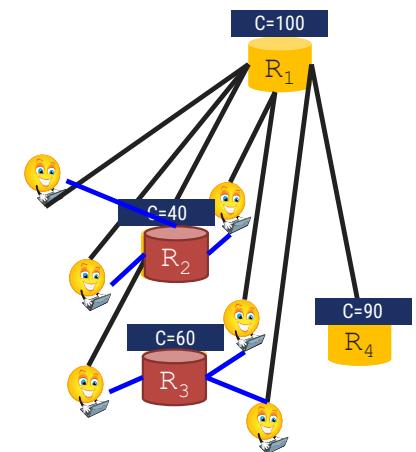
Factors that affect placement of replica servers:

- ▶ 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

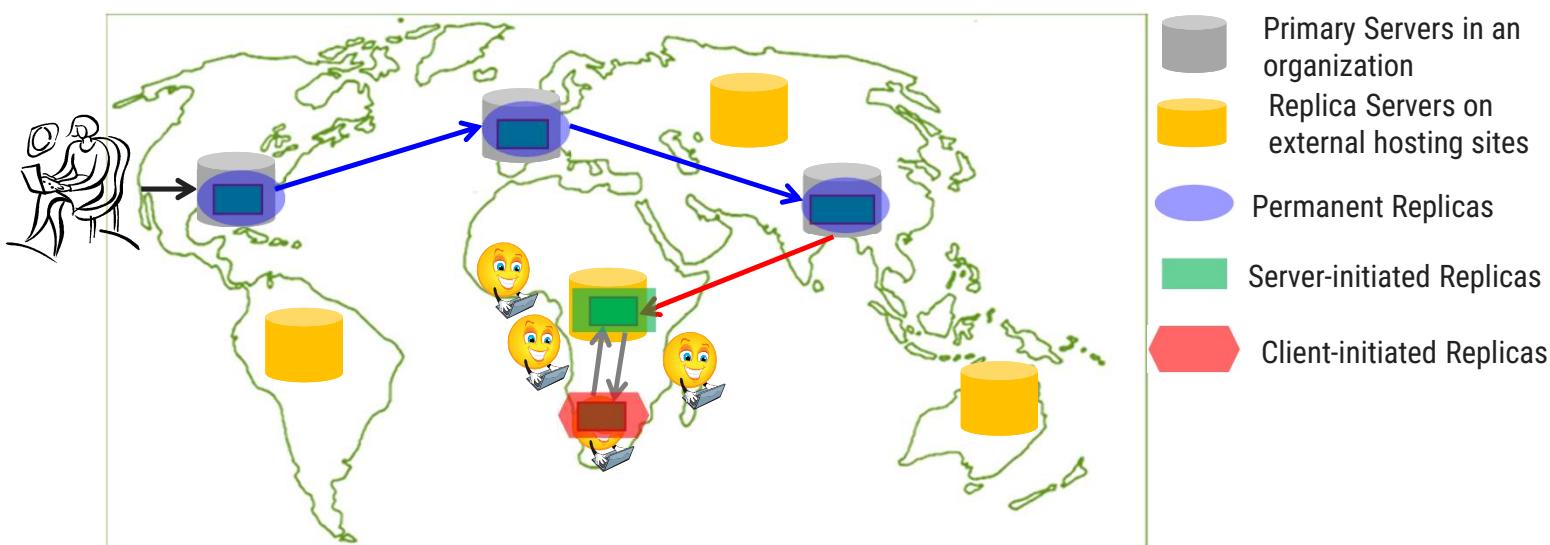
- ▶ Problem: K replica servers should be placed on some of the N possible replica sites such that
- ▶ Clients have low-latency/high-bandwidth connections
- ▶ Suggested a Greedy Approach

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

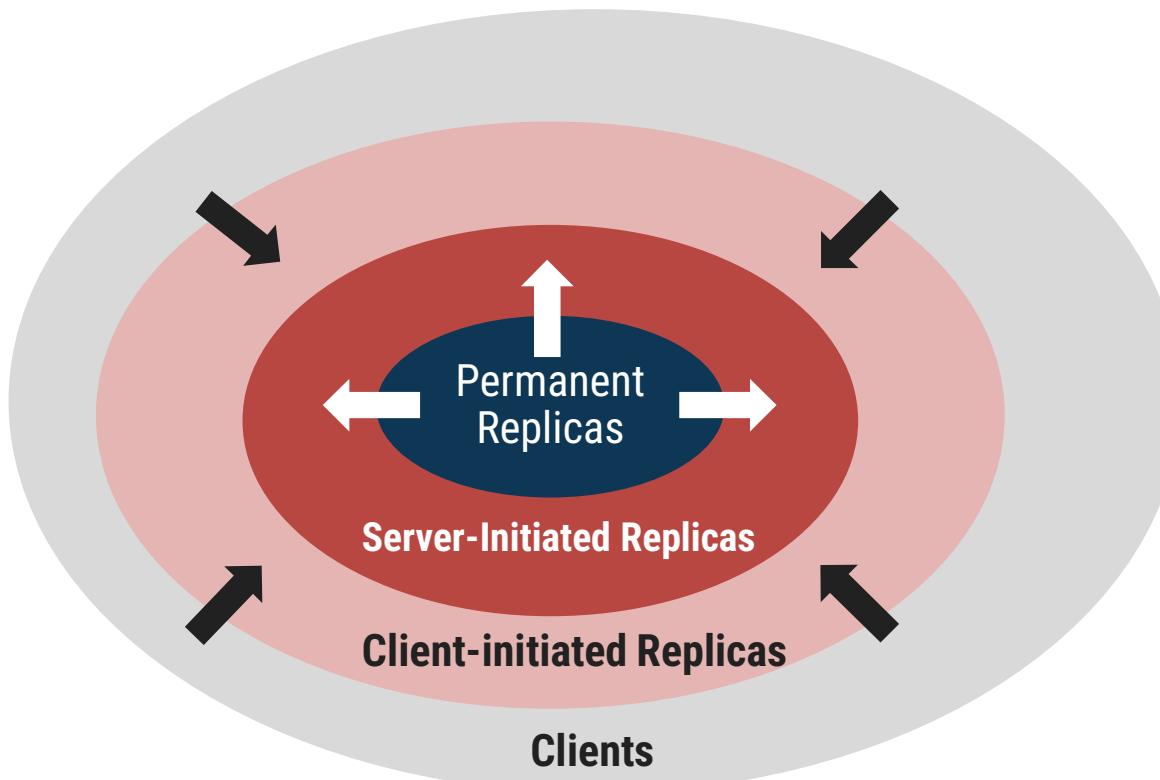


Content Replication and Placement

- ▶ 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



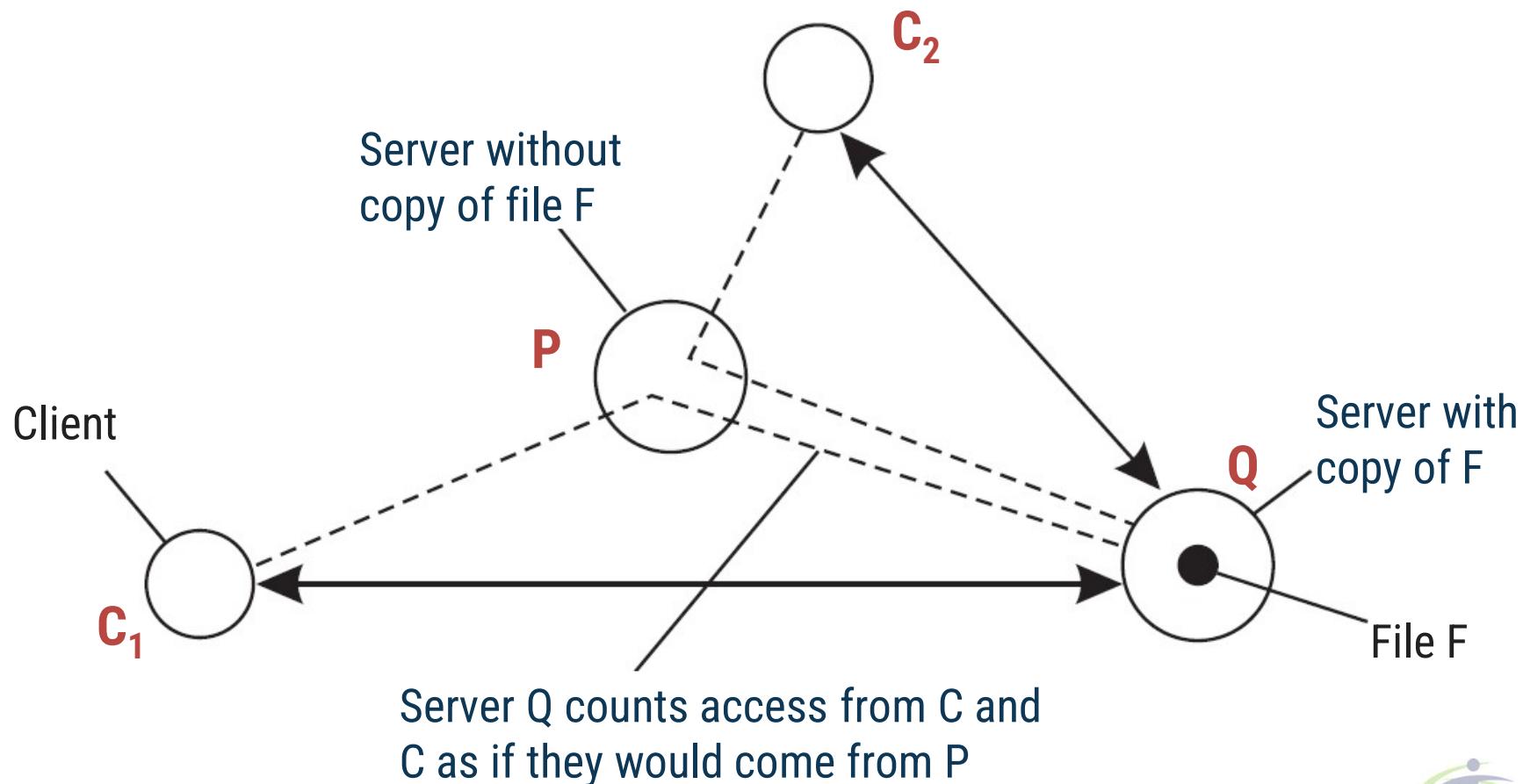
→ Server-initiated Replication

→ Client-initiated Replication

Permanent Replicas

- ▶ 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

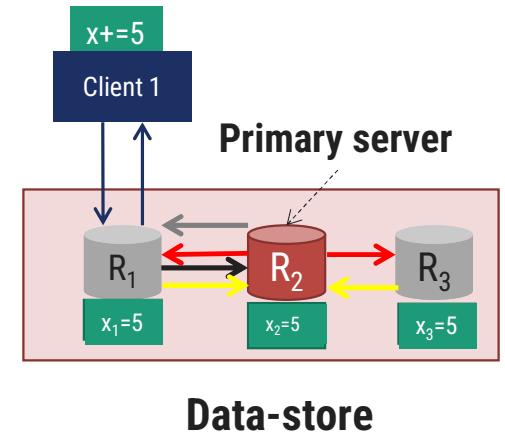
Remote-Write Protocol

► 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])

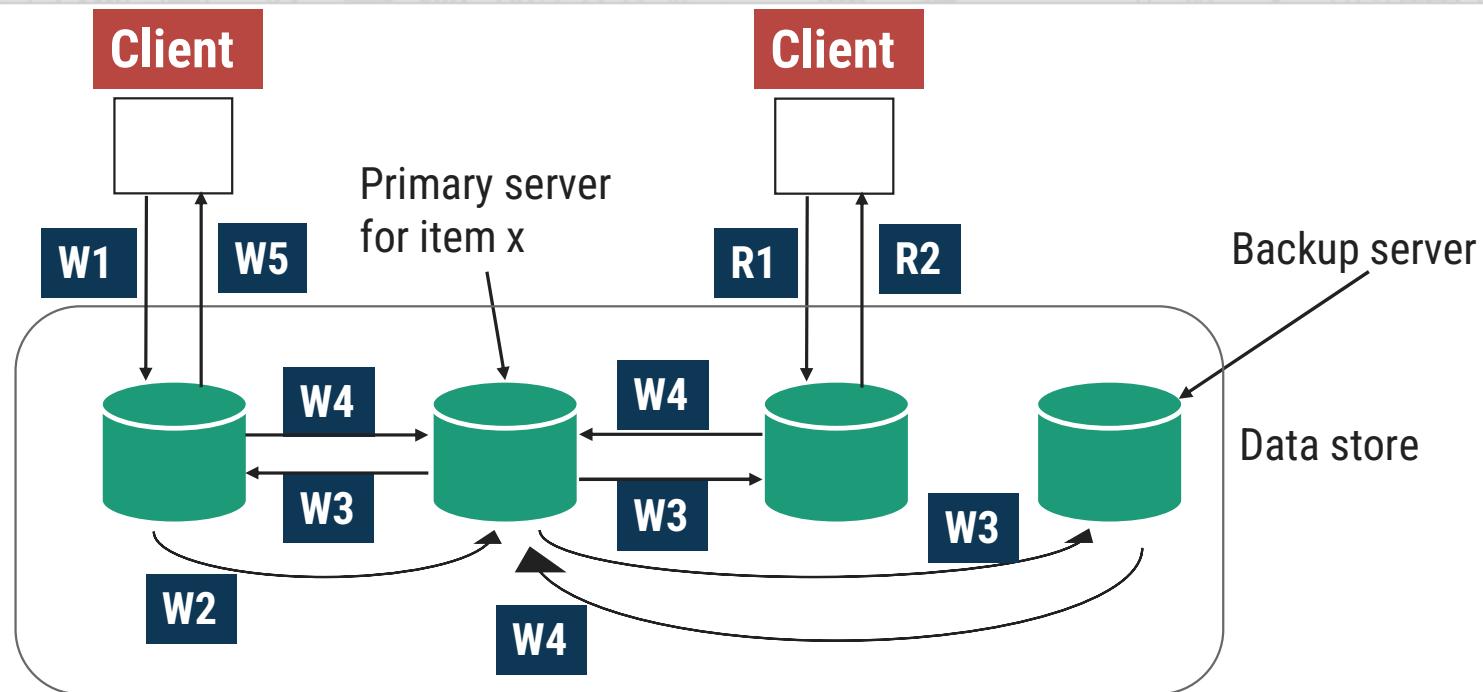
- Client connects to some replica R_c
- If the client issues write operation to R_c :
 - R_c forwards the request to the primary replica R_p
 - R_p updates its local value
 - R_p forwards the update to other replicas R_i
 - Other replicas R_i update, and send an ACK back to R_p
- After R_p receives all 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

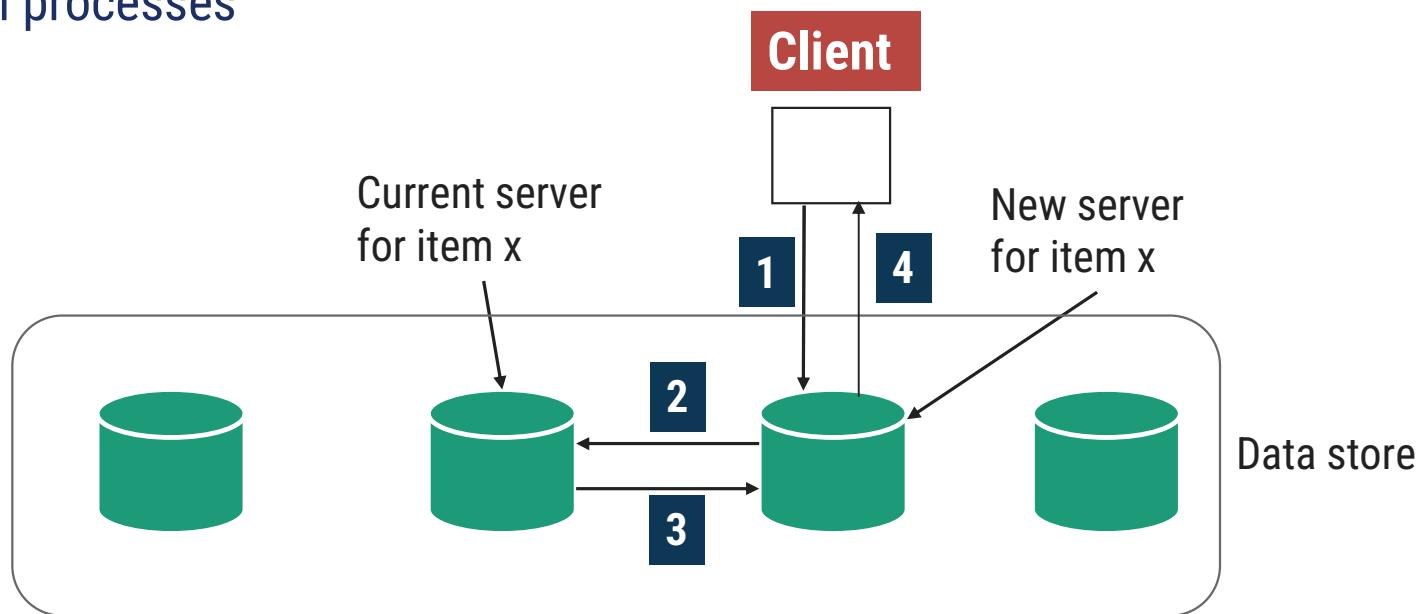
- R1.** Read request
- R2.** Response to read

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

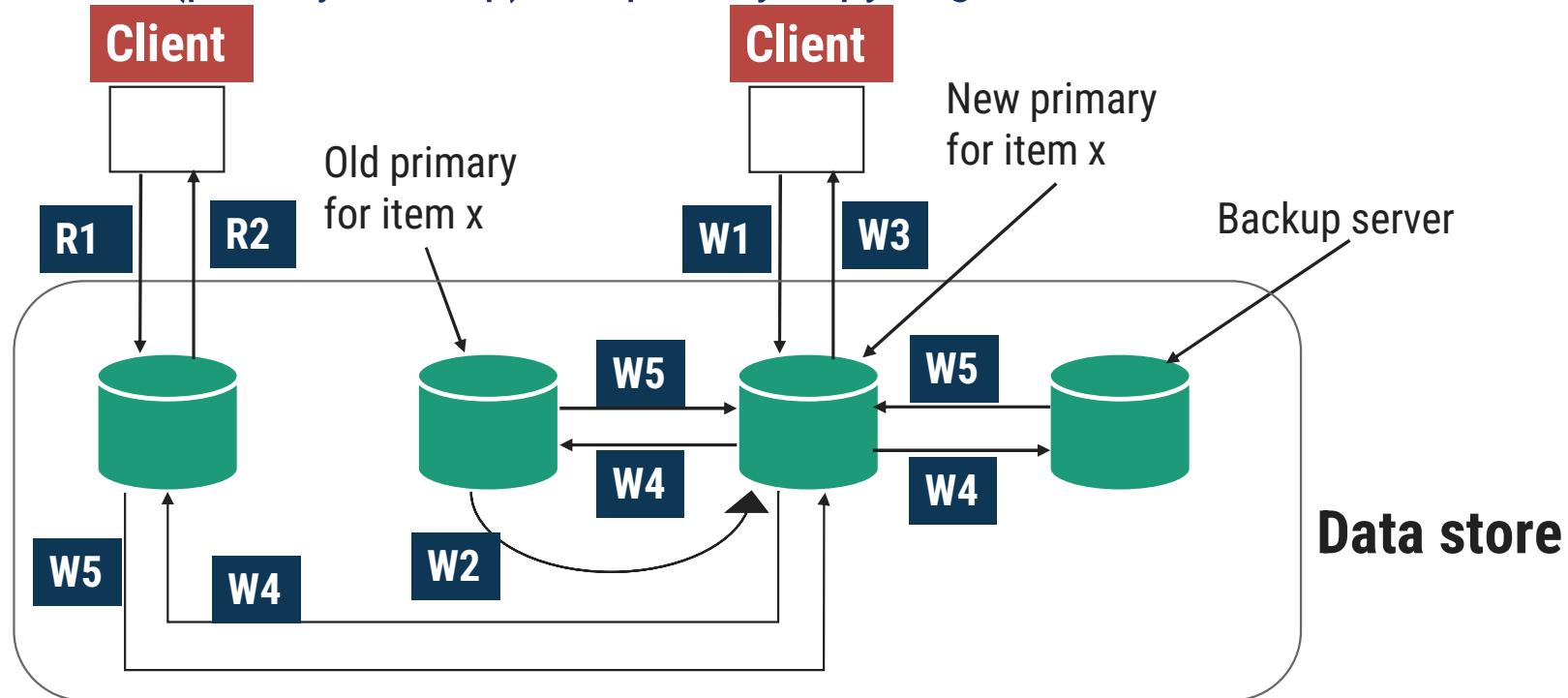
Case-1 : there is only a single copy of each data item x (no replication) a single copy is migrated between processes



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

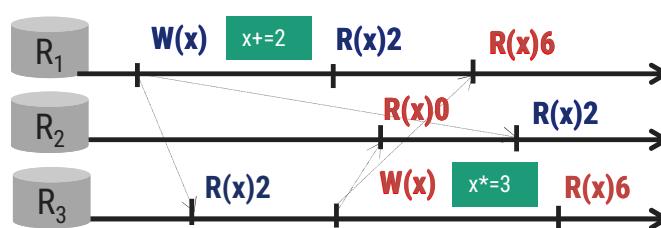
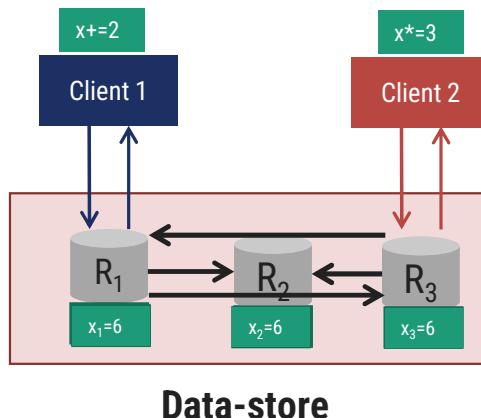
R2. Response to read

Replicated-Write Protocol

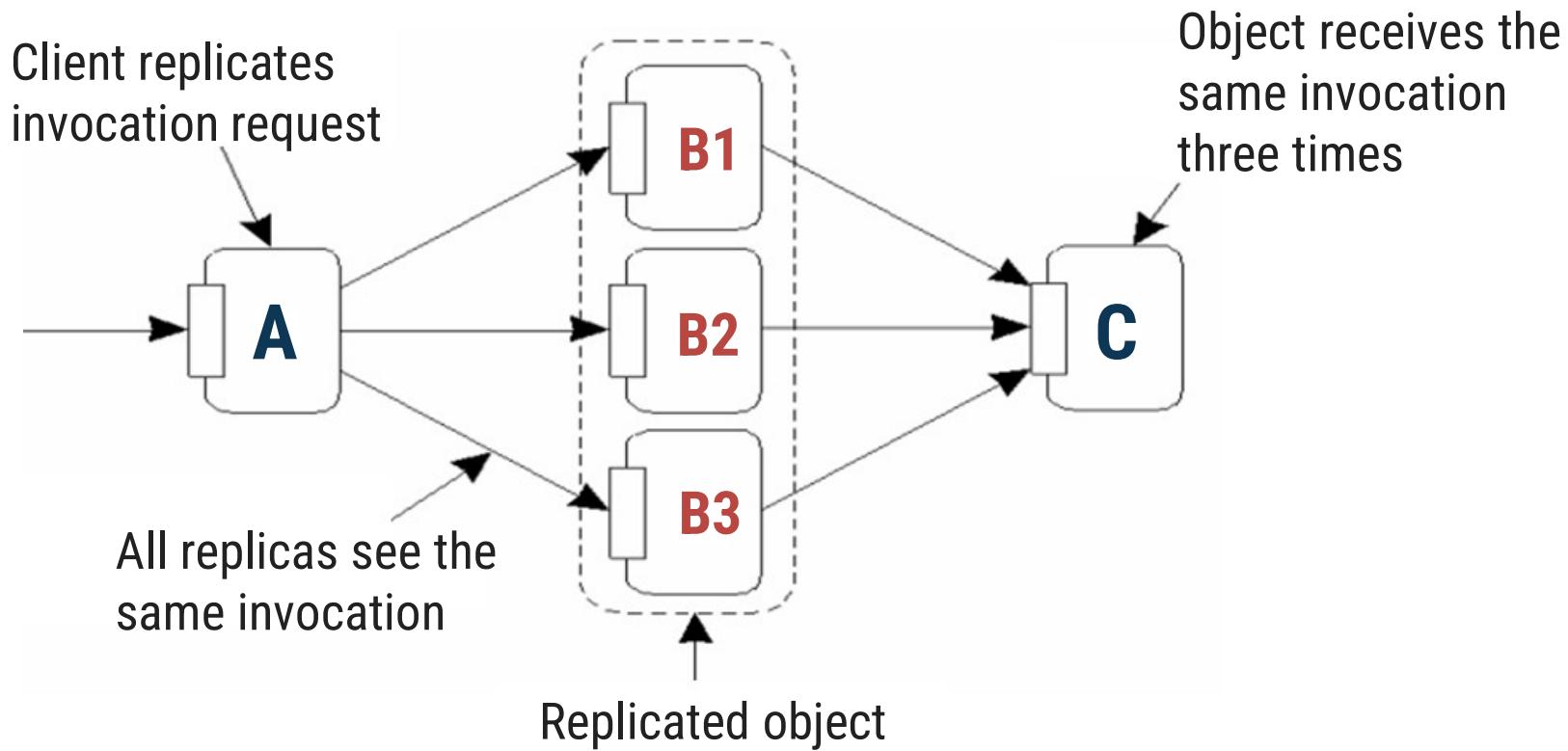
- ▶ In the primary base replication, the write operation is performed on one copy, but in the duplicate write protocol, the write operation is executed for plural copies.
- ▶ There are two ways for the replicated-write protocol,
 1. Active replication: An operation is forwarded to all replicas
 2. Quorum-Based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

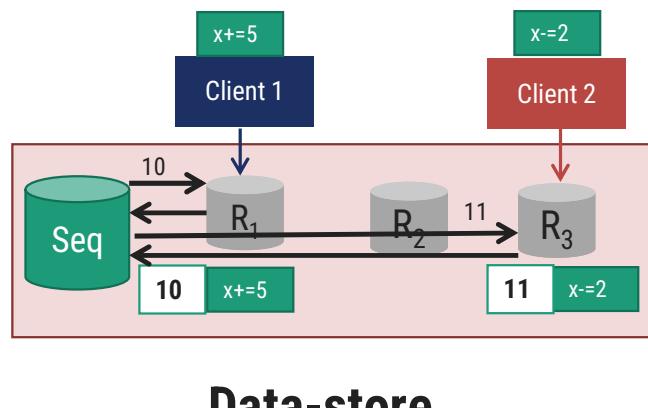


The problem of replicated invocations – ‘B’ is a replicated object (which itself calls ‘C’). When ‘A’ calls ‘B’, how do we ensure ‘C’ isn’t invoked three times?

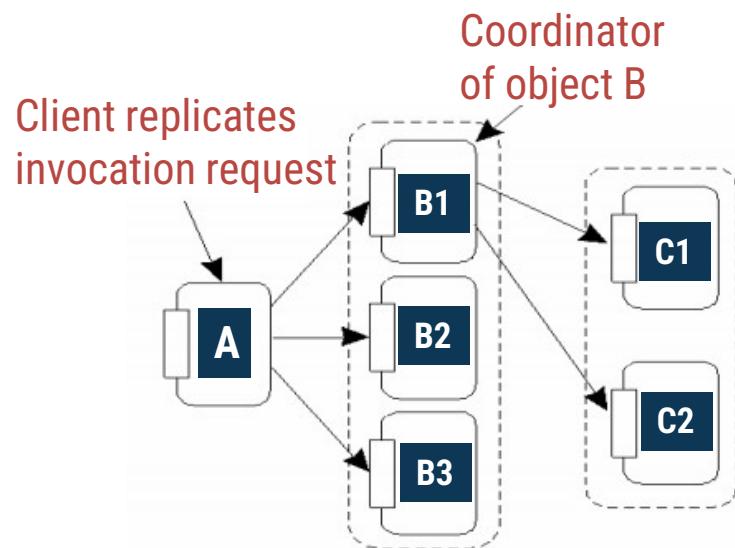
Centralized Active Replication Protocol

Approach

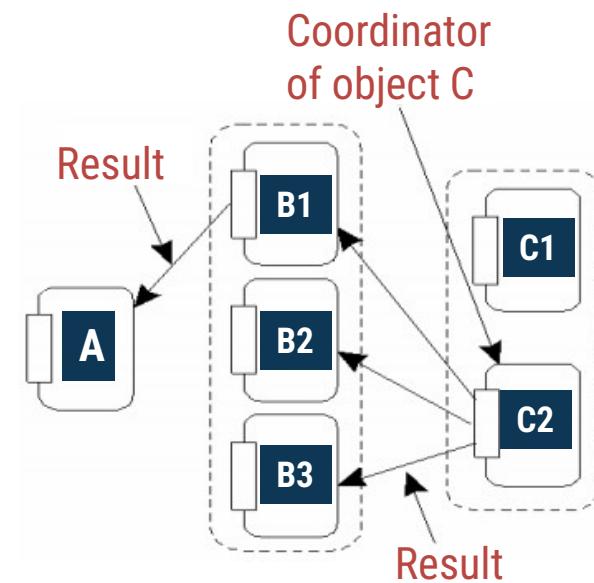
- ▶ There is a centralized coordinator called sequencer (**Seq**)
- ▶ When a client connects to a replica R_c and issues a write operation
 - R_c forwards the update to the **Seq**
 - **Seq** assigns a sequence number to the update operation
 - R_c propagates the sequence number and the operation to 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

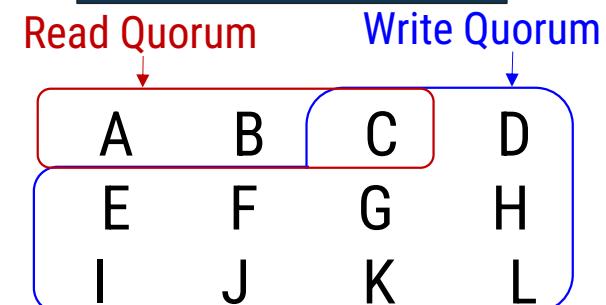
- ▶ Based on voting
- ▶ Read quorum (N_R) : Num. of servers must agree on version num. for a read
- ▶ Write quorum (N_W) : Num. of servers must agree on version num. for a write
- ▶ If N is the total number of replicas,
- ▶ N_R and N_W must fulfill:
 - $N_R + N_W > N$: Prevents read-write conflicts
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



$N_R = 3$ and $N_W = 10$

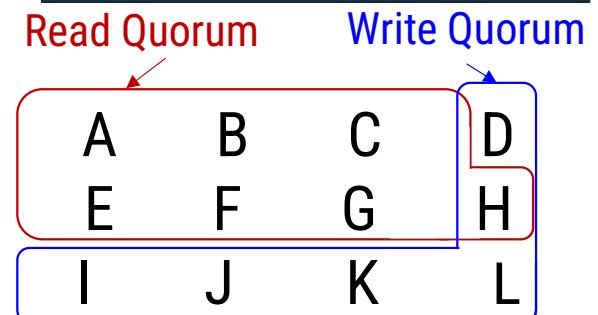
C1: $N_R + N_W = 13 > N = 12$
→ No RW conflicts

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

Quorum-Based Protocols – Example 2

- ▶ $N_R = 7$ and $N_W = 6$
- ▶ Why violating C2 causes WW conflicts?
 - If one client chooses {A, B, C, E, F, G} as its write set
 - And another client chooses {D, H, I, J, K, L} as its write set
 - The two updates will be accepted without detecting that they actually conflict, thus leading to an inconsistent view!

A choice that may lead to **write-write** conflicts



$$N_R = 7 \text{ and } N_W = 6$$

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

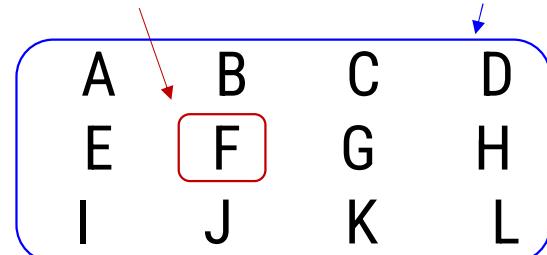
C2: $N_W \nmid 12/2 = 6$
→ WW conflicts may arise

Quorum-Based Protocols – Example 3

- ▶ $N_R = 1$ and $N_W = 12$
- ▶ A client can read a replicated file by finding any copy
 - Good read performance!
- ▶ A client needs to attain a write quorum on all copies
 - Slow write performance!
- ▶ This example demonstrates a scheme that is generally referred to as ROWA (or Read-Once, Write-All)

A correct choice, known as **ROWA** (read one, write all).

Read Quorum Write Quorum



$$N_R = 1 \text{ and } N_W = 12$$

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

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

Cache Coherence Protocols

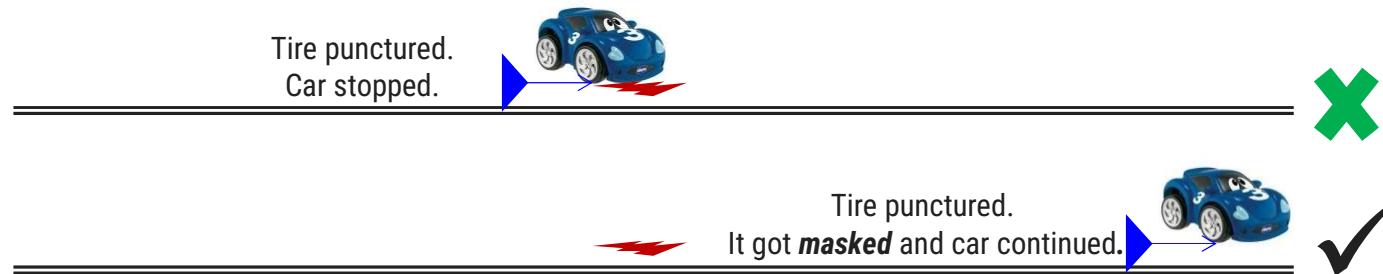
- ▶ These are a special case, as the cache is typically controlled by the client not the server.
- ▶ Coherence Detection Strategy:
 - When are inconsistencies actually detected?
 - Statically at compile time: extra instructions inserted.
 - Dynamically at runtime: code to check with the server.
- ▶ Coherence 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).

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

Availability

Reliability

A Dependable System

Safety

Maintainability

- A system temporarily fails to operate correctly, nothing catastrophic happens

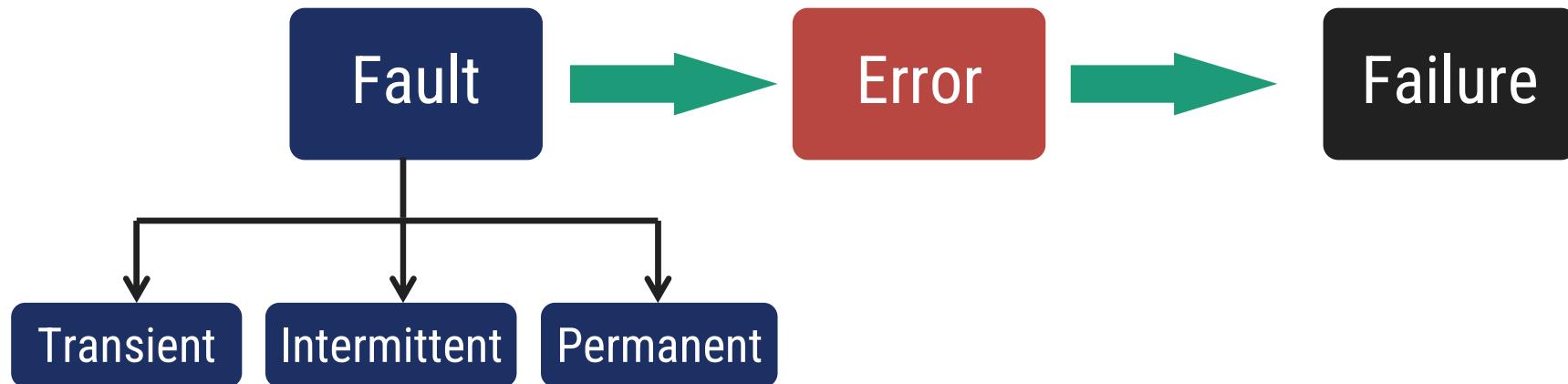
- A highly-reliable system is one that will most likely continue to work without interruption during a relatively long period of time

- How easy a failed system can be repaired

Availability vs. Reliability

- ▶ There is a distinction between **availability** and **reliability**
 - Availability refers to the probability that a system is operating correctly at any given moment
 - $\text{Availability} = \text{MTTF}/(\text{MTTF}+\text{MTTR})$
 - Reliability measures how long a system can operate without a breakdown
- ▶ A **highly-available (HA)** system is one that will most likely be working at a given instant in time
- ▶ A **highly-reliable** system is one that will most likely continue to work without interruption during a relatively long period of time

Faults, Errors and Failures



Failure Models

Type of Failure	Description
Crash Failure	A server halts, but was working correctly until it stopped
Omission Failure <ul style="list-style-type: none">■ Receive Omission■ Send Omission	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">■ Value Failure■ State Transition Failure	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).

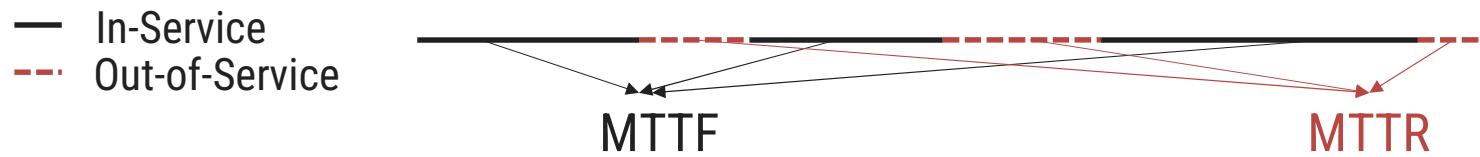
▶ Permanent Fault:

- 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**)



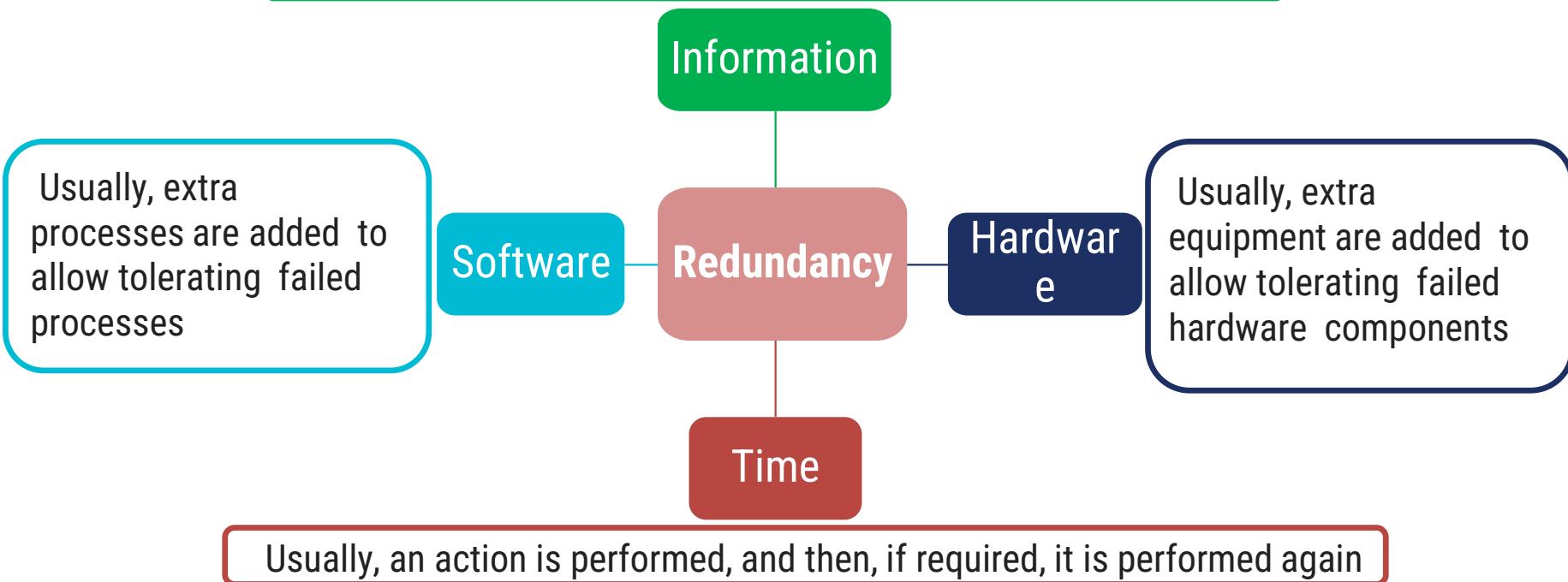
Fault Tolerance Requirements

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

Faults Masking by Redundancy

- ▶ The key technique for masking faults is to use redundancy

Usually, extra bits are added to allow recovery from garbled bits

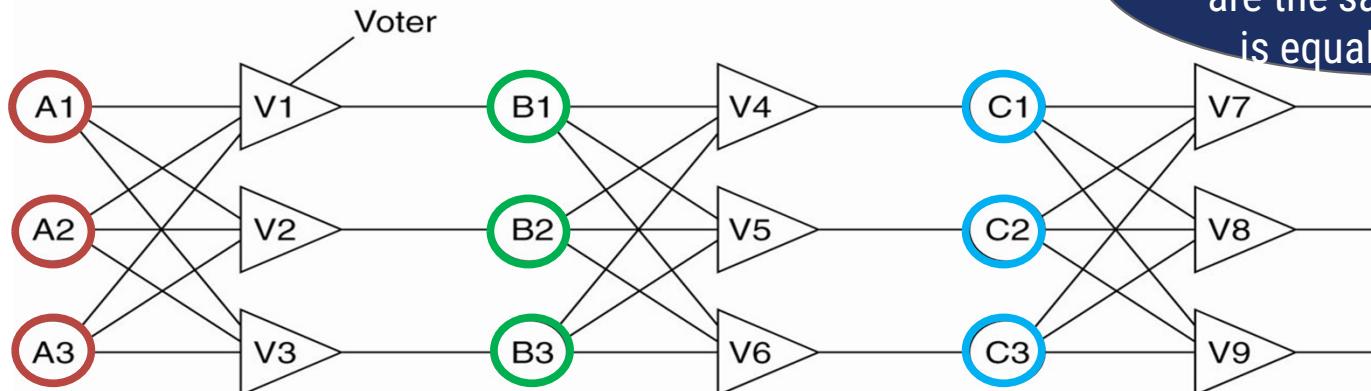


Triple Modular Redundancy

If one is faulty, the final result will be incorrect



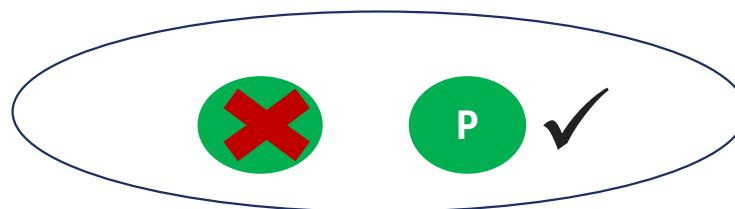
If 2 or 3 of the inputs are the same, the output is equal to that input



Each device is replicated 3 times and after each stage is a triplicated voter

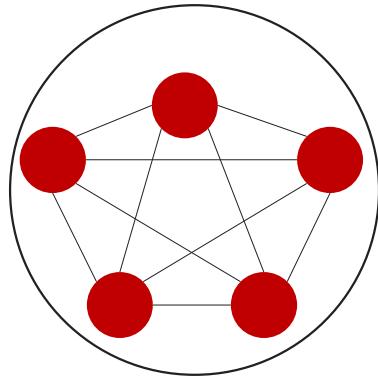
Process Resilience

- ▶ 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

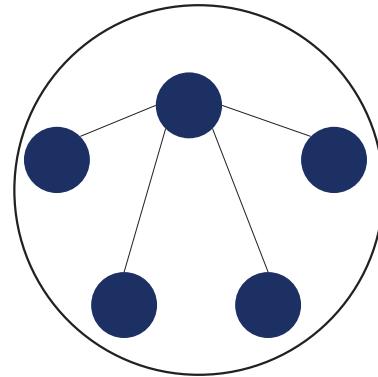


Flat Versus Hierarchical Groups

Flat Group



Hierarchical Group



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

- ▶ 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.

Failure Masking and Replication

- ▶ By organizing a fault tolerant group of processes, we can protect a single vulnerable process.
- ▶ There are two approaches to arranging the replication of the group:

1. Primary (backup) Protocols:

- A group of processes is organized in a **hierarchical** fashion in which a primary coordinates all write operations.
- When the primary crashes, the backups execute some election algorithm to choose a new primary.

2. Replicated-Write Protocols:

- Replicated-write protocols are used in the form of active replication, as well as by means of **quorum-based protocols**.
- Solutions correspond to organizing a collection of identical processes into a **flat group**.

Agreement in Faulty Systems

- ▶ To have all non-faulty processes reach consensus on some issue (quickly).
- ▶ The two-army problem.
- ▶ Even with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.
- ▶ Possible cases:
 1. Synchronous (lock-step) versus asynchronous systems.
 2. Communication delay is bounded (by globally and predetermined maximum time) or not.
 3. Message delivery is ordered (in real-time) or not.
 4. Message transmission is done through unicasting or multicasting.

Agreement in Faulty Systems

- ▶ **Goal:** have all non-faulty processes reach consensus on some issue, and establish that consensus within a finite number of steps
- ▶ In Distributed database system, there may be a situation where data managers have to decide
 - Whether to **commit** or **Abort** the Transaction.
- ▶ In Distributed, mutual trust or agreement is required.
- ▶ In Distributed database transaction, when there is no failure, Agreement is easy.
- ▶ **If failure, processes must exchange their value with other processes.**
- ▶ So, Agreement protocol helps us to decide or agree on a value in presence of failures.

Agreement in Faulty Systems

		Message Ordering				Communication Delay	
Process Behavior	Synchronous	Unordered		Ordered			
		✓	✓	✓	✓		
				✓	✓	Bounded	
					✓	Unbounded	
	Asynchronous				✓	Bounded	
					✓	Unbounded	
		Unicast	Multicast	Unicast	Multicast		
Message Transmission							

Agreement in Faulty Systems

- ▶ In practice most distributed systems assume that:
 - Processes behave asynchronously
 - Message transmission is unicast
 - Communication delays are unbounded
- ▶ Usage of ordered (reliable) message delivery is typically required
- ▶ The agreement problem has been originally studied by Lamport and referred to as the *Byzantine Agreement Problem [Lamport et al.]*

Byzantine Agreement Problem

► Lamport assumes:

- Processes are **synchronous**
- Messages are **unicast** while preserving **ordering**
- Communication **delay is bounded**
- There are **N** processes, where each process **i** will provide a value **v_i** to the others
- There are at most **k faulty processes**

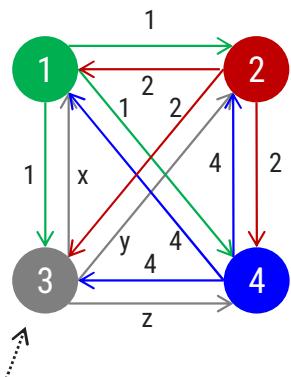
Byzantine Agreement Problem

		Message Ordering				Communication Delay		
		Unordered	Ordered					
Process Behavior	Synchronous	✓	✓	✓	✓	Bounded		
	Synchronous			✓	✓	Unbounded		
	Asynchronous				✓	Bounded		
	Asynchronous				✓	Unbounded		
		Unicast	Multicast	Unicast	Multicast			
Message Transmission								

Lamport suggests that each process i constructs a vector V of length N , such that if process i is non-faulty, $V[i] = v_i$. Otherwise, $V[i]$ is undefined

Byzantine Agreement Problem

► Case I: $N = 4$ and $k = 1$



Faulty process

Step1: Each process sends its value to the others

Step2: Each process collects values received in a vector

1 Got $(1, 2, x, 4)$ $(1, 2, y, 4)$ $(1, 2, 3, 4)$ $4 \text{ Got}(1, 2, z, 4)$	2 Got $(1, 2, y, 4)$ (a, b, c, d) $(1, 2, z, 4)$	4 Got $(1, 2, x, 4)$ $(1, 2, y, 4)$ (i, j, k, l)
---	--	--

Step3: Every process passes its vector to every other process

Byzantine Agreement Problem

Step 4:

- ▶ Each process examines the i^{th} element of each of the newly received vectors
- ▶ If any value has a **majority**, that value is put into the result vector
- ▶ If no value has a majority, the corresponding element of the result vector is marked **UNKNOWN**

1 Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

Result Vector:

(1, 2, UNKNOWN, 4)

2 Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

Result Vector:

(1, 2, UNKNOWN, 4)

4 Got

(1, 2, x, 4)
(1, 2, y, 4)
(i, j, k, l)

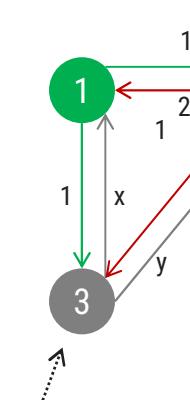
Result Vector:

(1, 2, UNKNOWN, 4)

The algorithm reaches an agreement

Byzantine Agreement Problem

► Case II: N = 3 and k = 1



Faulty
process

Step1: Each process sends its value to the others

Step2: Each process collects values received in a vector

Step3: Every process passes its vector to every other process

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

1 Got
(1, 2, y)
(a, b, c)

2 Got
(1, 2, x)
(d, e, f)

Byzantine Agreement Problem

Step 4:

- ▶ Each process examines the i^{th} element of each of the newly received vectors
- ▶ If any value has a **majority**, that value is put into the result vector
- ▶ If no value has a majority, the corresponding element of the result vector is marked **UNKNOWN**

1 Got
(1, 2, y)
(a, b, c)

The algorithm has failed
to produce
an agreement

2 Got
(1, 2, x)
(d, e, f)

Result Vector:
(UNKNOWN, UNKNOWN, UNKNOWN)

Result Vector:
(UNKNOWN, UNKNOWN, UNKNOWN)

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 doOperation 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 retransmissions 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.

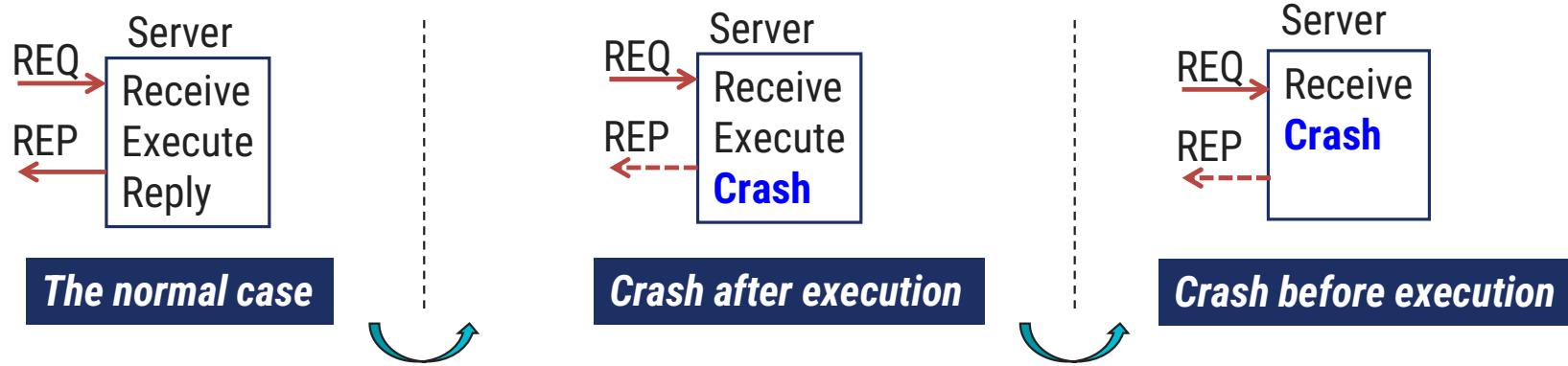
► Client crashes after sending request:

- 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.

Server crashes after receiving request

► Three events that can happen at the server:

1. Send the completion message (M),
2. Print the text (P),
3. Crash (C).

► Server events can occur in six different orderings:

Ordering	Description
M→P→C	A crash occurs after sending the completion message and printing the text
M→C(→P)	A crash occurs after sending the completion message, but before the text is printed
P→M→C	A crash occurs after printing the text and sending the completion message
P→C(→M)	The text is printed, after which a crash occurs before the completion message is sent
C(→P→M)	A crash happens before the server could do anything
C(→M→P)	A crash happens before the server could do anything

Server crashes after receiving request

- ▶ After the crash of the server, the client does not know whether its request to print some text was carried out or not
- ▶ The client has a choice between 4 strategies:

Reissue Strategy	Description
Never	Never reissue a request, at the risk that the text will not be printed
Always	Always reissue a request, potentially leading to the text being printed twice
Reissue When Not ACKed	Reissue a request only if it did not yet receive an ACK that its print request had been delivered to the server
Reissue When ACKed	Reissue a request only if it has received an ACK for the print request

Server crashes after receiving request

- Example: a client send a message to a server for printing (P) it, having a completion message back (M). The server can crash (C)

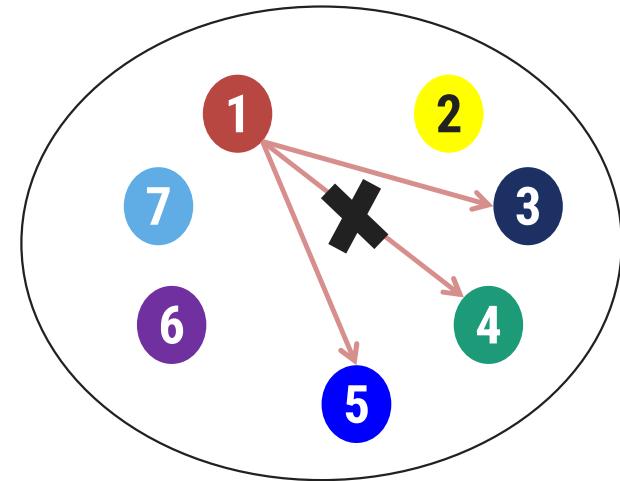
Client	Strategy M -> P			Strategy P -> M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
 DIP = Text is printed twice
 ZERO = Text is not printed at all

Different combinations of client and server strategies in the presence of server crashes

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

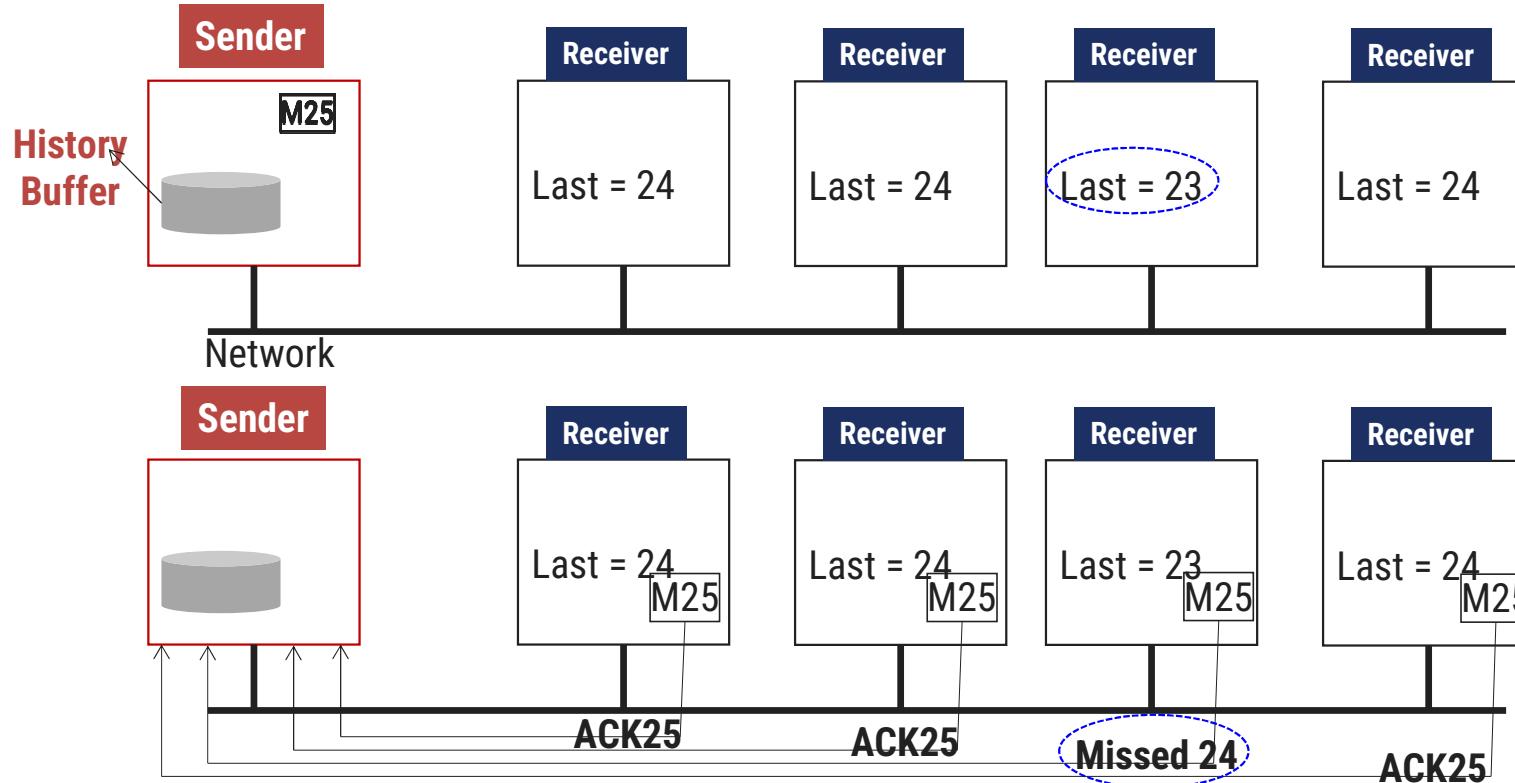
- ▶ Reliable multicasting indicates that a message that is sent to a process group should be delivered to each member of that group
- ▶ A distinction should be made between:
 - Reliable communication in the presence of faulty processes
 - Reliable communication when processes are assumed to operate correctly
- ▶ 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



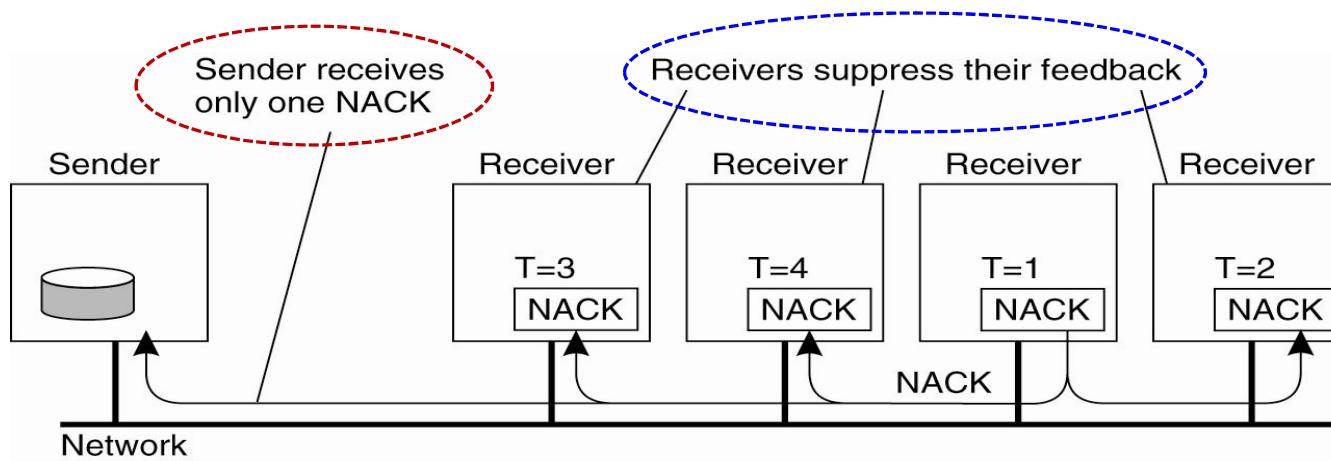
Scalability in Reliable Multicasting

- ▶ 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.

Nonhierarchical Feedback Control

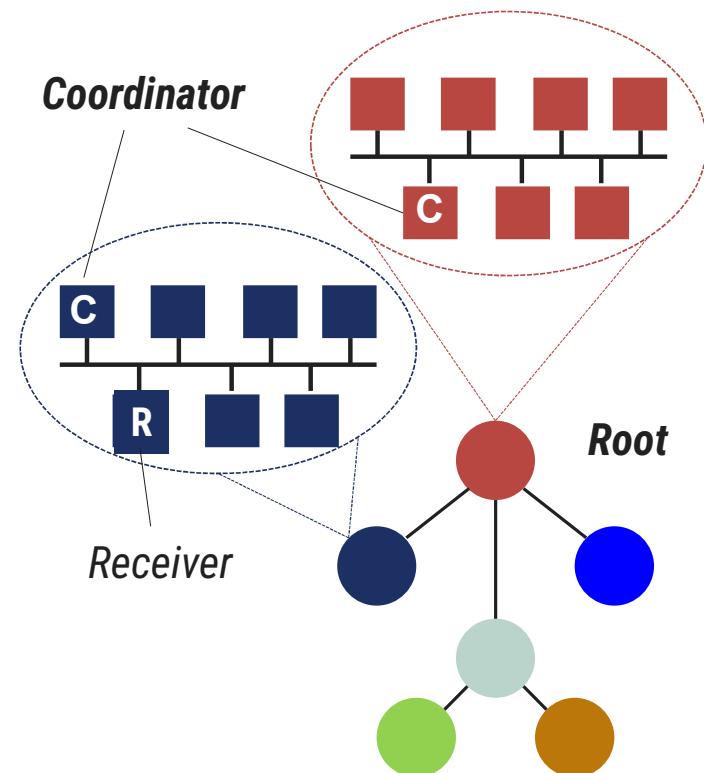
► **Feedback Suppression** – reducing the number of feedback messages to the sender (as implemented in the Scalable Reliable Multicasting Protocol).

- Successful delivery is never acknowledged, only missing messages are reported (NACK), which are multicast to all group members.
- If another process is about to NACK, this feedback is suppressed as a result of the first multicast NACK.
- In this way, only a single NACK is delivered to the sender



Hierarchical Feedback Control

- ▶ Feedback suppression is basically a nonhierarchical solution
- ▶ Achieving scalability for very large groups of receivers requires that hierarchical approaches are adopted
- ▶ The group of receivers is partitioned into a number of subgroups, which are organized into a tree
- ▶ The subgroup containing the sender S forms the **root** of the tree
- ▶ Within a subgroup, any reliable multicasting scheme can be used
- ▶ Each subgroup appoints a local **coordinator C** responsible for handling retransmission requests in its subgroup
- ▶ If C misses a message m, it asks the C of the parent subgroup to retransmit m



Atomic Multicast

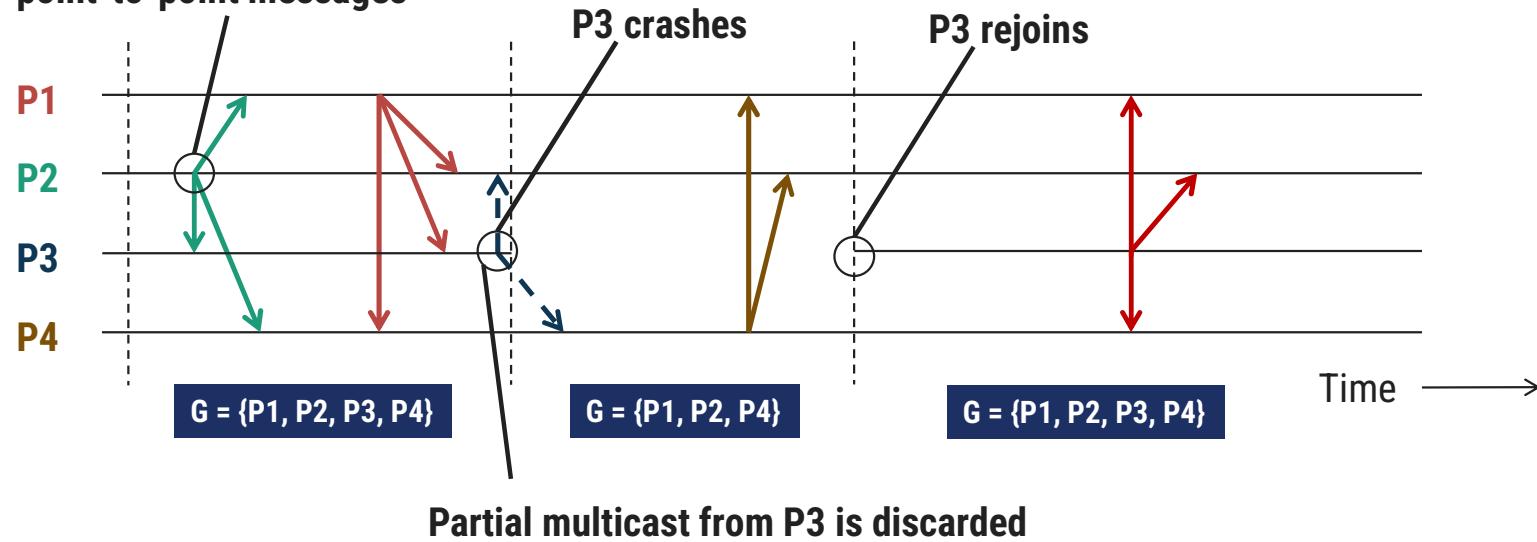
- ▶ **P1:** What is often needed in a distributed system is the guarantee that a message is delivered to either all processes or to none at all
- ▶ **P2:** It is also generally required that all messages are delivered in the same order to all processes
- ▶ Satisfying P1 and P2 results in an **atomic multicast**
- ▶ Atomic multicast:
 - Ensures that non-faulty processes maintain a **consistent view**
 - Forces reconciliation when a process recovers and rejoins the group

Virtual Synchrony

- ▶ A multicast message m is uniquely associated with a list of processes to which it should be delivered
- ▶ This delivery list corresponds to a group view (G)
- ▶ There is only one case in which delivery of m is allowed to fail:
 - When a group-membership-change is the result of the sender of m crashing
- ▶ In this case, m may either be delivered to all remaining processes, or ignored by each of them

Virtual Synchrony

Reliable multicast by multiple point-to-point messages



The Principle of Virtual Synchronous Multicast

Message Ordering

- ▶ Four different virtually synchronous multicast orderings are distinguished:
 1. Unordered multicasts
 2. FIFO-ordered multicasts
 3. Causally-ordered multicasts
 4. Totally-ordered multicasts

Message Ordering

► Unordered multicasts

- A reliable, unordered multicast is a virtually synchronous multicast in which no guarantees are given concerning the order in which received messages are delivered by different processes

Process P1	Process P2	Process P3
Sends m1	Receives m1	Receives m2
Sends m2	Receives m2	Receives m1

Three communicating processes in the same group

Message Ordering

▶ FIFO-Ordered Multicasts

- With FIFO-Ordered multicasts, the communication layer is forced to deliver incoming messages from the same process in the same order as they have been sent

Process P1	Process P2	Process P3	Process P4
Sends m1	Receives m1	Receives m3	Sends m3
Sends m2	Receives m3	Receives m1	Sends m4
	Receives m2	Receives m2	
	Receives m4	Receives m4	

Four processes in the same group with two different senders.

Message Ordering

► Causally-ordered

- It preserves potential causality between different messages
- If message m₁ causally precedes another message m₂, regardless of whether they were multicast by the same sender or not, the communication layer at each receiver will always deliver m₁ before m₂

► Total-ordered

- This multicast requires that when messages are delivered, they are delivered in the same order to all group members
- Regardless of whether message delivery is unordered, FIFO-ordered, or causally-ordered

Virtually Synchronous Reliable Multicasting

- ▶ A virtually synchronous reliable multicasting that offers total-ordered delivery of messages is what we refer to as atomic multicasting

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Six different versions of virtually synchronous reliable multicasting

Distributed Commit

- ▶ 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.
- Step 2** When a participant receives a *VOTE_REQUEST* message, it returns either a *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

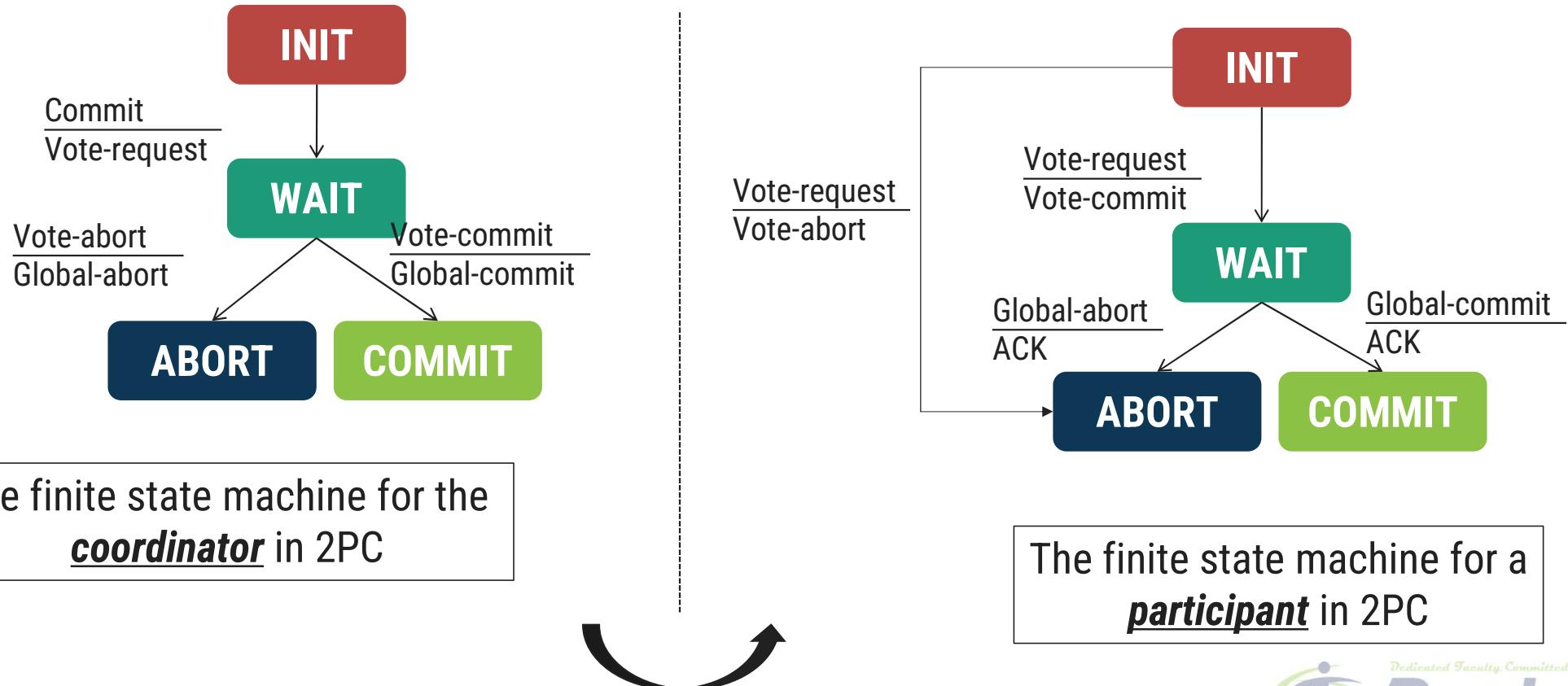
Step 1

- 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.

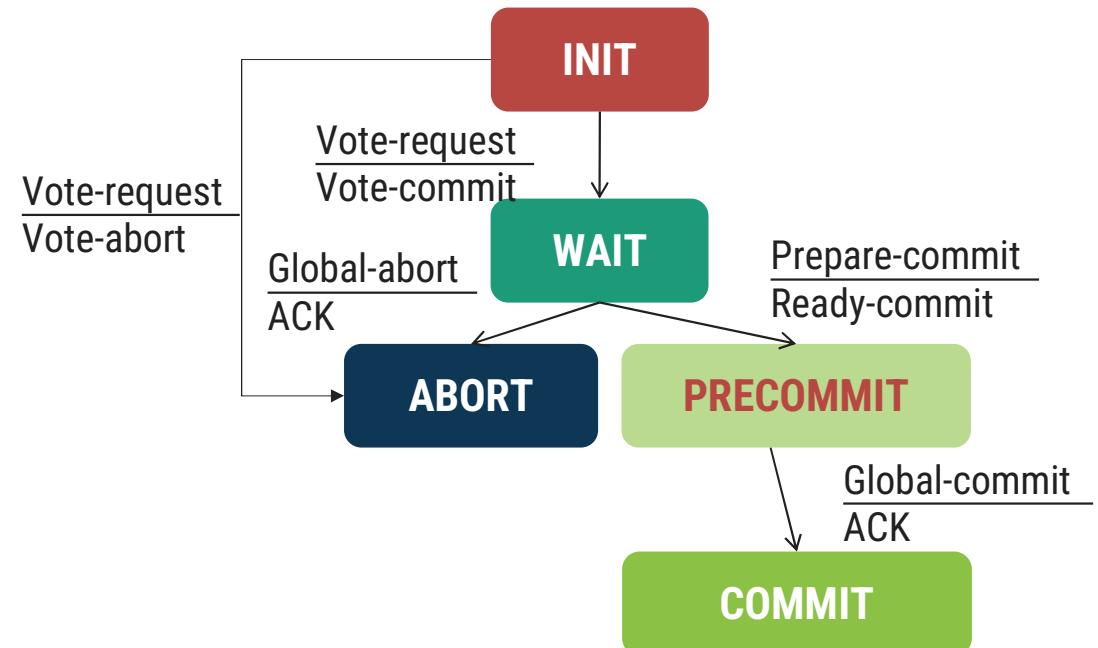
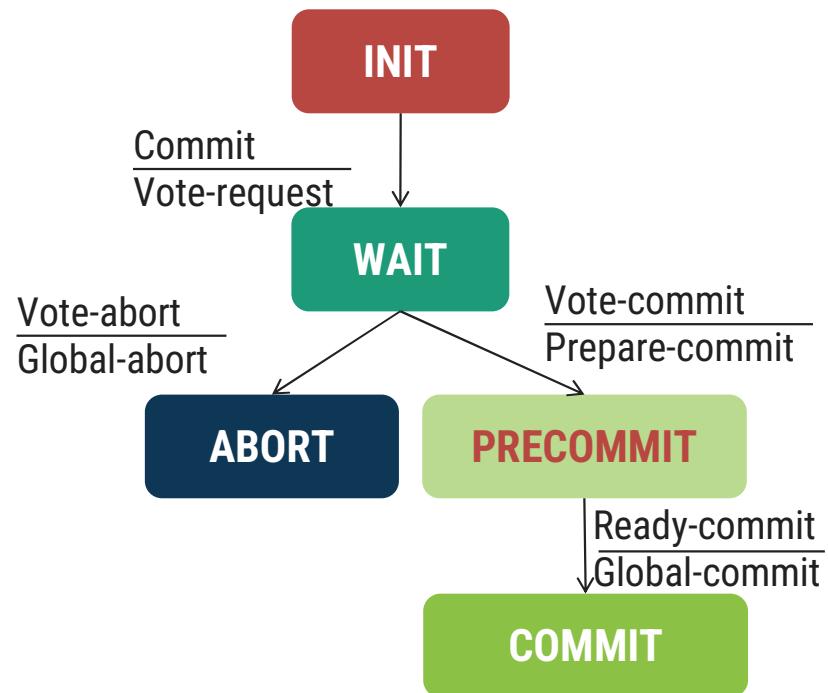
Step 2

- Each participant that voted for a commit waits for the final reaction by the coordinator.
- If a participant receives a ***GLOBAL_COMMIT*** message, it locally commits the transaction.
- Otherwise, when receiving a ***GLOBAL_ABORT*** message, the transaction is locally aborted as well.

2PC Finite State Machines



3PC Finite State Machines



Three-Phase Commit Protocol

- ▶ Essence: the states of the coordinator and each participant satisfy the following two conditions:
 1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
 2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.
- ▶ 3PC protocol when failures occur:
 - Coordinator is blocked in PRECOMMIT.
 - On timeout, it concludes that a participant has crashed, but that participant is known to have voted for commit. So coordinator can multicast a GLOBAL COMMIT
 - In addition, it relies on a recovery protocol for the crashed participant to eventually commit its part of the transaction when it comes up again.

Three-Phase Commit Protocol

- ▶ Participant blocked in **Pre-commit state**
 - Contact others
 - Collectively decide to commit
- ▶ Participant blocked in **Ready state**
 - Contact others
 - If any in Abort, then abort transaction
 - If any in Pre-commit, then move to Pre-commit state
 - If all in Ready state, then abort transaction

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:

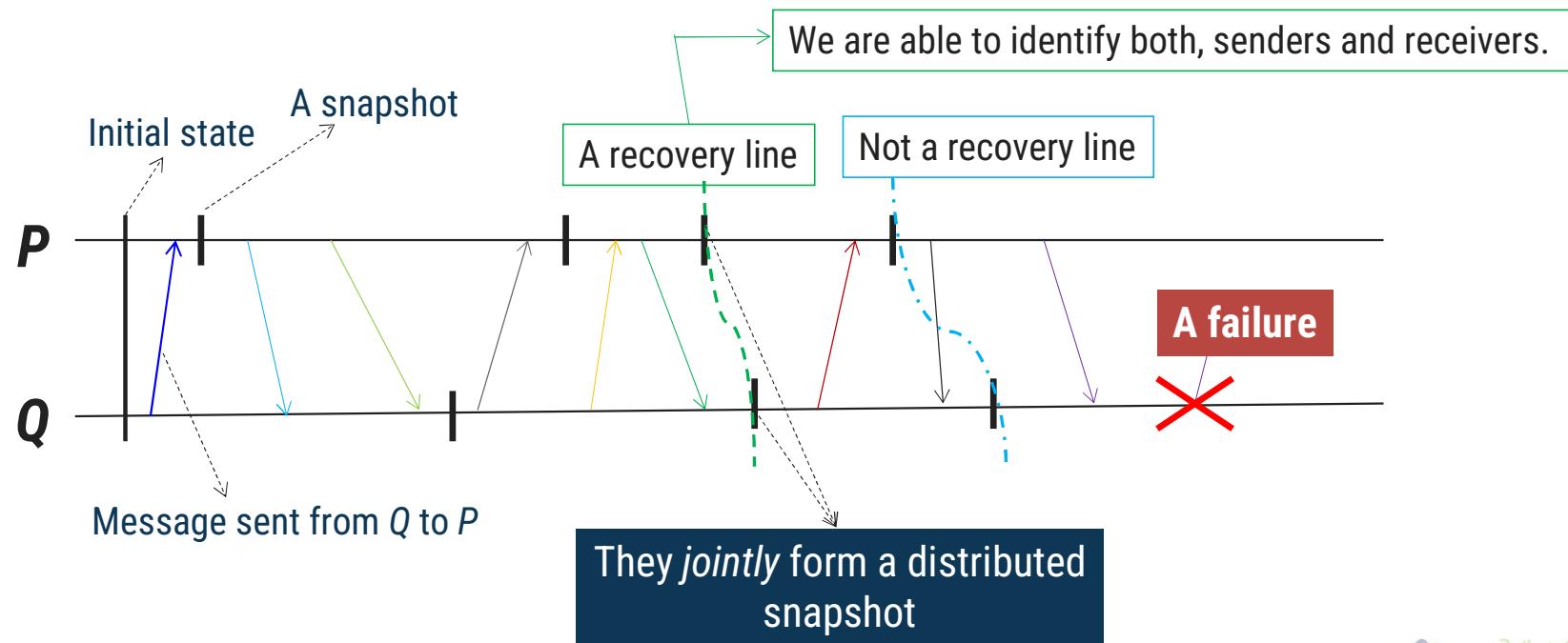
- ▶ Bring the system into a correct state, from which it can then continue to execute
- ▶ Forward recovery is typically faster than backward recovery but requires that it has to be known in advance which errors may occur
- ▶ Problem:
 - In order to work, all potential errors need to be accounted for 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



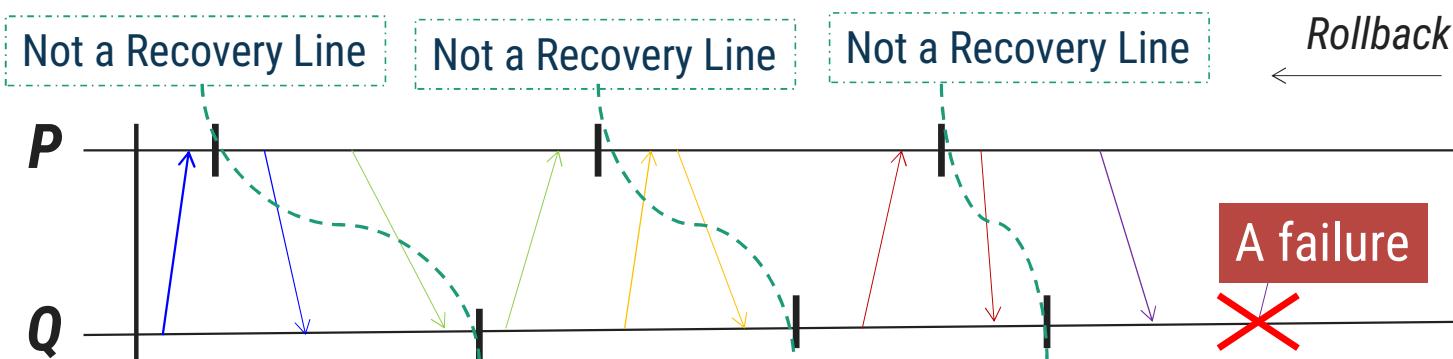
Checkpointing

► 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

Domino Effect

- ▶ Independent checkpointing may make it difficult to find a recovery line, leading potentially to a domino effect resulting from cascaded rollbacks



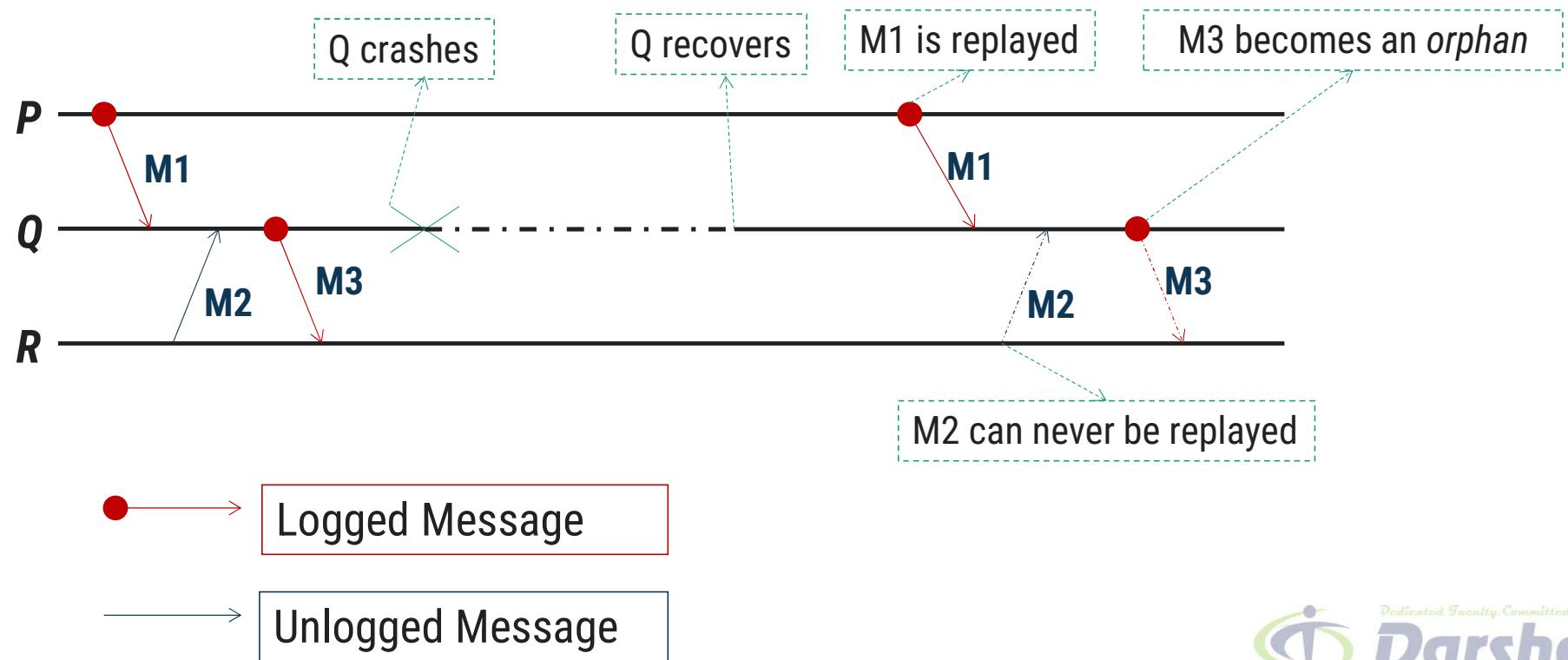
- ▶ With coordinated checkpointing, the saved state is automatically globally consistent, hence, domino effect is inherently avoided

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



Distributed System (DS)
GTU #3170719



Unit-6 Security



Prof. Umesh H. Thoriya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot

 umesh.thoriya@darshan.ac.in
 9714233355



Topics to be covered

- Introduction to Security- Security Threats, Policies, and Mechanisms
- Design Issues
- Basics of Cryptography
- Secure Channels- Authentication, Message Integrity and Confidentiality
- Secure Group Communication
- Access Control- General Issues in Access Control,
- Firewalls
- Secure Mobile Code
- Denial of Service;
- Security Management
- Key Management
- Secure Group Management
- Authorization Management

Introduction to Security

- ▶ Computer security is about keeping systems, programs, and data secure
- ▶ What services do you expect from secure Distributed Systems?
- ▶ Secure DS should provide
 - Confidentiality of Information : Information is disclosed only to authorized parties
 - Integrity of Information : Alterations to system's assets is made only in an authorized way
- ▶ Secure DS are immune against possible security threats that compromise confidentiality and integrity

Introduction to Security

Security in distributed systems can be divided into two parts.

1. One part concerns the **communication between users or processes**, possibly residing on different machines.
 - The principal mechanism for ensuring secure communication is:
 - Secure channels
 - Authentication
 - Message integrity
 - Confidentiality
2. The other part concerns authorization, which deals with ensuring that a process gets only those access rights to the resources in a distributed system it is entitled to.

Goals of Security

- ▶ **Secrecy:** Information within the system must be accessible only to authorized users.
- ▶ **Privacy:** Information given to the users must be used only for the purpose for which it was given.
- ▶ **Authenticity:** The user must be able to verify that the data obtained is from expected sender only.
- ▶ **Integrity:** Information must be protected from unauthorized access.

Types of Security Threats

► Types of security threats to consider:

1. Interception
2. Interruption
3. Modification
4. Fabrication

Types of Security Threats

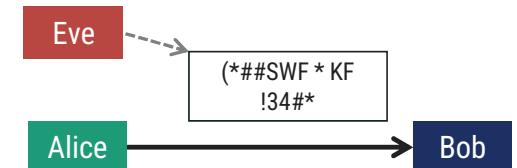
1. Interception

- Unauthorized party has gained access to a service or data
- Example: Illegal copying of files, Eavesdropping over network



2. Interruption

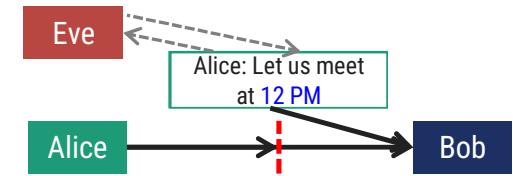
- Services or data become unavailable, unusable or destroyed
- Example: Denial-of-Service (DoS) Attacks



Types of Security Threats

3. Modification

- Unauthorized changing of data or tampering with services
- Example: Changing a program to secretly log the activities



4. Fabrication

- Additional data or activity is generated that would normally not exist
- Example: Replay attacks



Security Policy and Mechanisms

- ▶ Security policy describes what actions the entities in a system are allowed to take and which ones are prohibited.
- ▶ Security mechanisms implement security policies.
- ▶ The following techniques are used:
 - **Encryption:** Provides a means to implement confidentiality, since it transforms the data into cypher text which attacker cannot understand.
 - **Authentication:** To verify whether the user, client, server etc. are authentic or not. User are authenticated by password.
 - **Authorization:** To check as a weather the client is authorized to perform specific task.
 - **Auditing:** Tools are used to trace which clients accessed what information and when they did so.

Cryptography

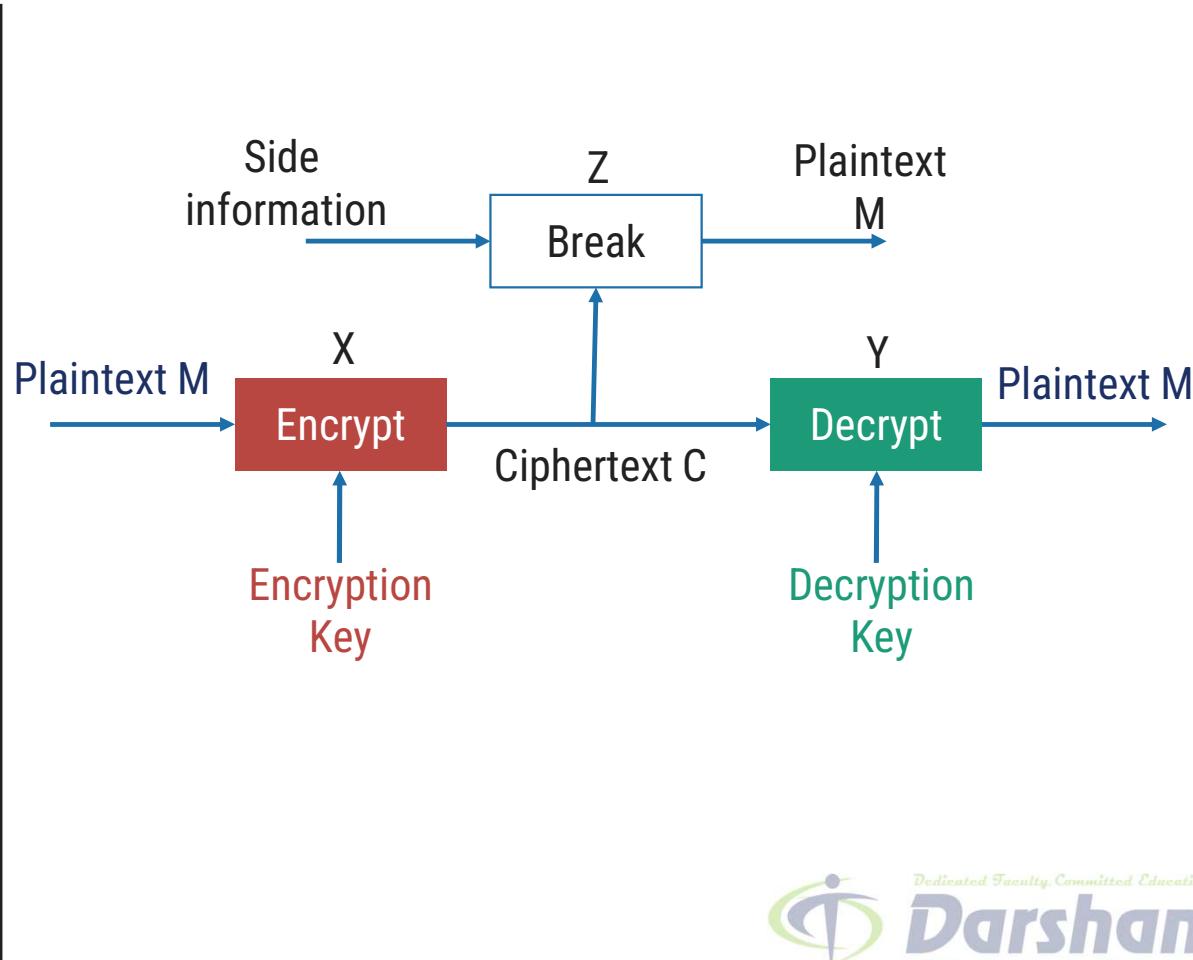
- ▶ Cryptography is defined as a means of **protecting private information** against **unauthorized access** in cases where physical security is difficult to achieve.
- ▶ Cryptography is carried out using two basic operations:
- ▶ **Encryption:** The process of transforming intelligible information (plaintext) into unintelligible form (cipher text).
- ▶ **Decryption:** The process of transforming the information from cipher text to plaintext.

Cryptography

- ▶ The encryption algorithm has the following form: $C = E(P, K_e)$
- ▶ Where
 - P = plaintext to be encrypted
 - K_e = encryption key
 - C = resulting cipher text
- ▶ The decryption algorithm is performed by the same matching function which has the following form: $P = D(C, K_d)$
- ▶ Where,
 - C = cipher text to be decrypted
 - K_d = decryption key

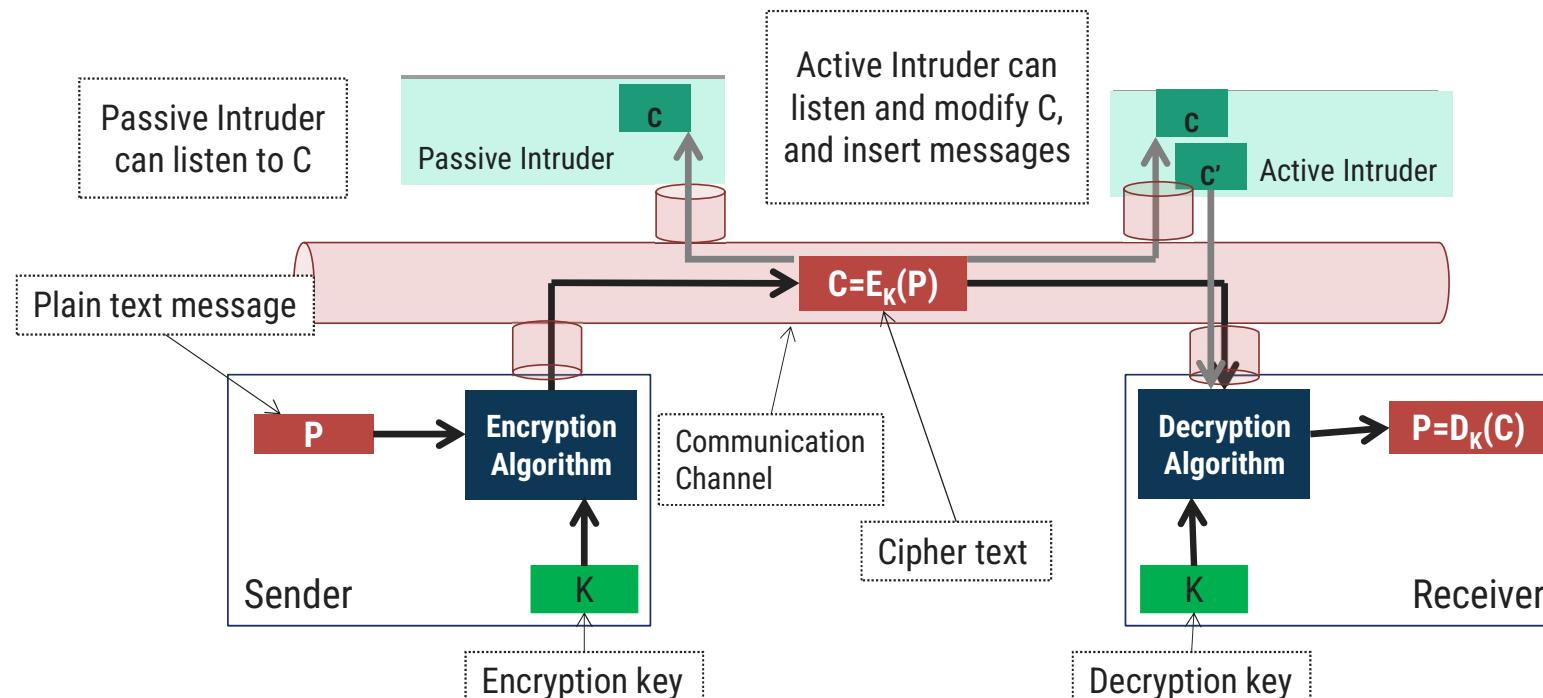
Prof. Uresh H. Patel

#3170719 (DS) • Unit 6 - Security



Cryptographic systems

- ▶ Cryptography is the study of techniques for secure communication in the presence of third parties



$C=E_K(P) \rightarrow C$ is obtained by encrypting the plain text P with key K
 $P=D_K(C) \rightarrow P$ is obtained by decrypting the cipher text P with key K

Types of Cryptographic Systems

► Two types

- Symmetric Cryptosystem (Shared-key system)
- Asymmetric Cryptosystem (Public-key system)

Symmetric Cryptosystem



- ▶ Safe with a strong lock, only Alice and Bob have a copy of the key
 - Alice encrypts: locks message in the safe with her key
 - Bob decrypts: uses his copy of the key to open the safe

Symmetric Cryptosystem

- ▶ A symmetric cryptosystem uses the **same key** for both encryption and decryption.

$$P = D_K(E_K(P))$$

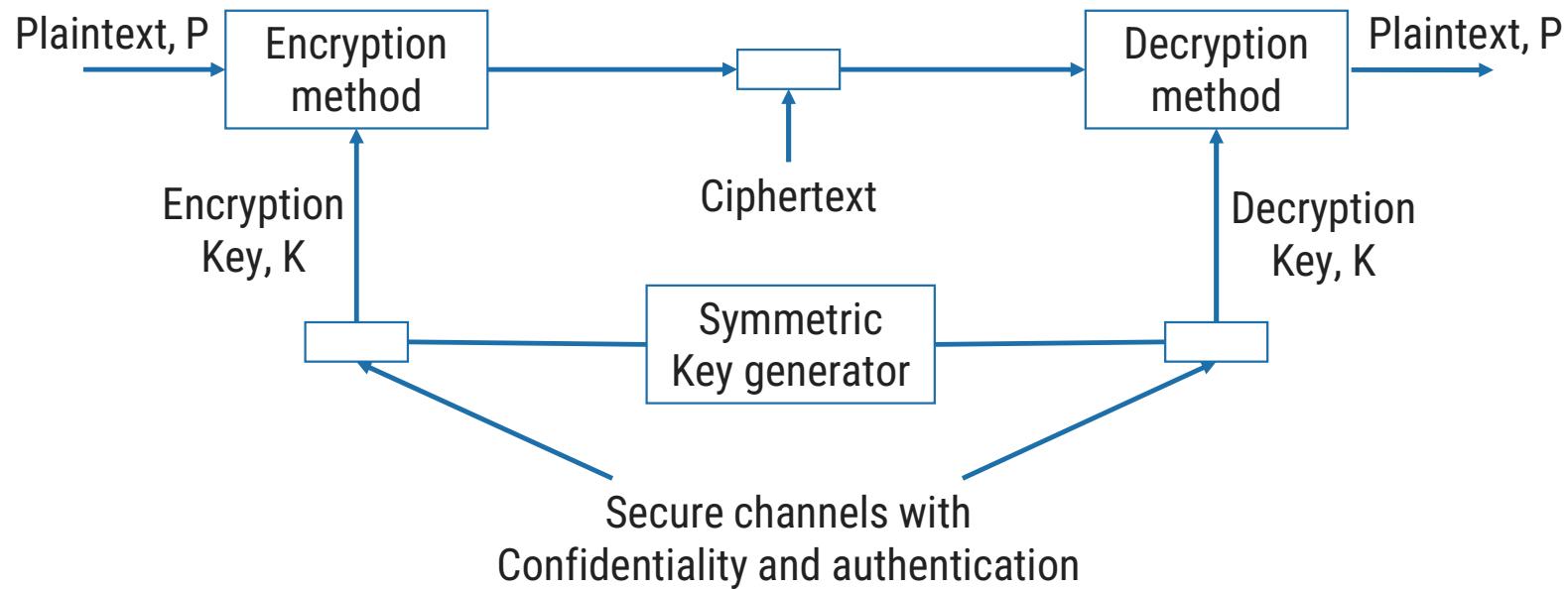
- ▶ The shared key $K_{A,B}$ between Alice A and Bob B should be kept secret
- ▶ It is necessary that the key should be easily alterable if required and is always kept secret.
- ▶ This implies that the key is known only to authorized users.
- ▶ Symmetric cryptosystems are also called as **shared key or private key cryptosystems** since both sender and receiver share the same key.



$$C = E_K(P)$$

$$P = D_K(C)$$

Symmetric Cryptosystem



Secret key Distribution

Asymmetric Cryptography



New Idea:

Use the good old mailbox principle.

Everyone can drop a letter.

But Only the owner has the correct key to open the box.



Public Key

Private Key

Asymmetric Cryptography

- ▶ In asymmetric cryptosystem, the keys used for encryption and decryption are different but they form a unique pair.
- ▶ There are separate keys for encryption (K_e) and decryption (K_d).
- ▶ $P = D K_d(E K_e(P))$
- ▶ One key in the asymmetric cryptosystem is kept private while the other one is made public.
- ▶ Hence these types of cryptosystems are referred to as public key systems.

Type of Attacks

Passive Attacks	Active Attacks
Browsing	Virus
Inferencing	Worm
Masquerading	Logic bomb
	Integrity attack
	Authenticity attack
	Delay attack
	Replay attack
	Denial attack

Passive Attack

- ▶ Intruder access unauthorized information from a computer system but cannot cause harm to the system.
- ▶ **Browsing:** Intruders here attempt to read stored files, traverse message packet on the network, access other process memory, etc.
- ▶ **Inferencing:** The intruder records and analyzes past activities and access methods and uses this information to draw inferences.
- ▶ **Masquerading:** An intruder Masquerades as an authorized user or a program to gain access to unauthorized data or resource.

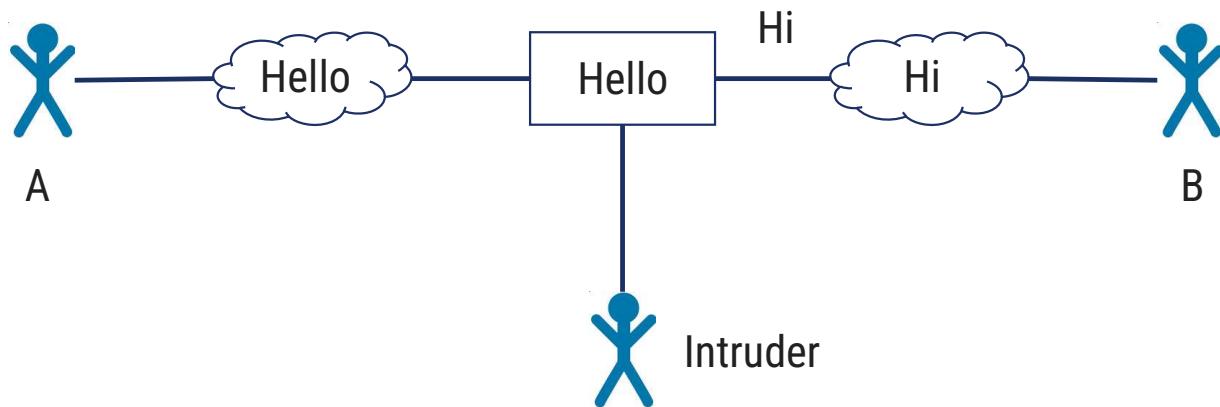
Active Attack

- ▶ **Virus:** is a small computer program that needs to be executed by either running it or having it loaded from boot sector of a disk.
- ▶ **Worm:** is a small piece of software the uses computer network and security holes to replicate itself.
- ▶ **Logic bomb:** is a piece of code intentionally inserted into a software system that will set off a malicious function when specified conditions are met.

Active Attack

► Integrity Attack:

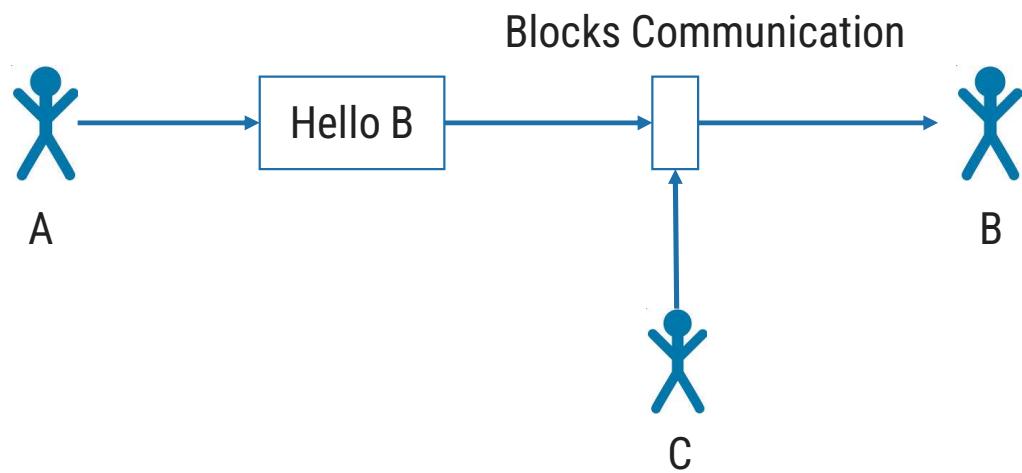
- An intruder can change the message while it is traveling in the communication channel and the receiver may interpret it as original message.



Active Attack

► Denial Attack

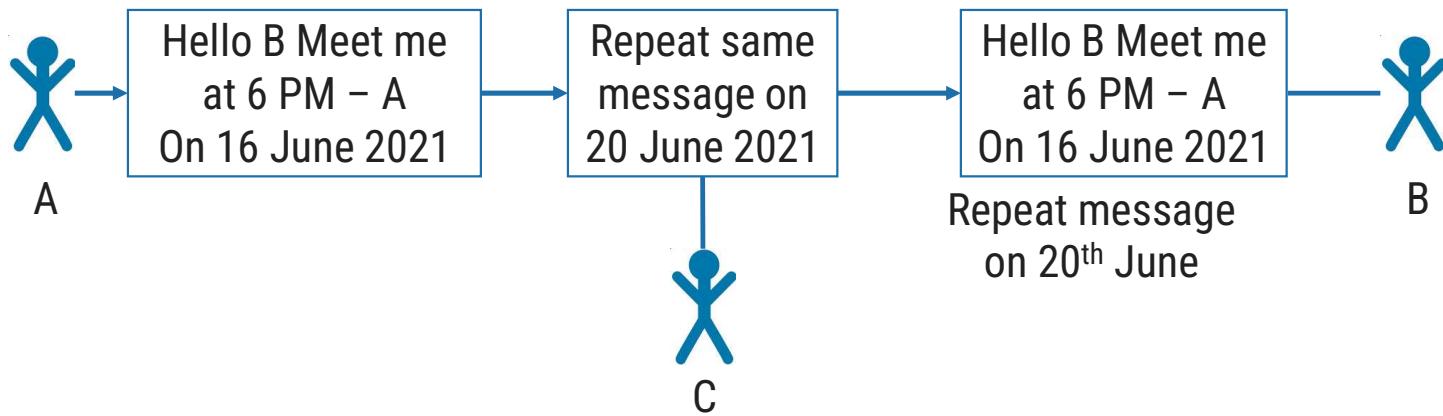
- An intruder might partly or completely block communication path between two processes.



Active Attack

► Replay Attack

- An intruder retransmit an old message that is accepted as new message by the receiver.



Design issues when considering security

Three design issues when considering security

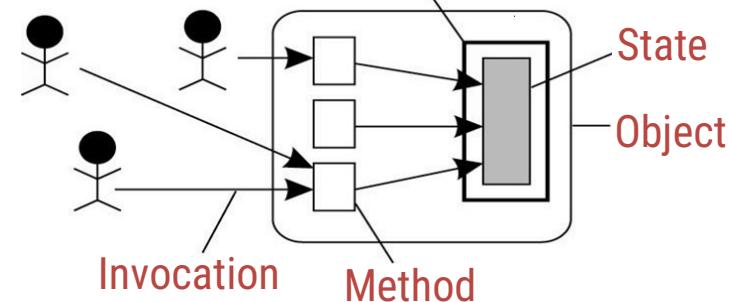
1. Focus of Control.
2. Layering of Security Mechanisms.
3. Simplicity.

Design Issue: Focus of Control

- There are three approaches that can be followed to protect a distributed application:

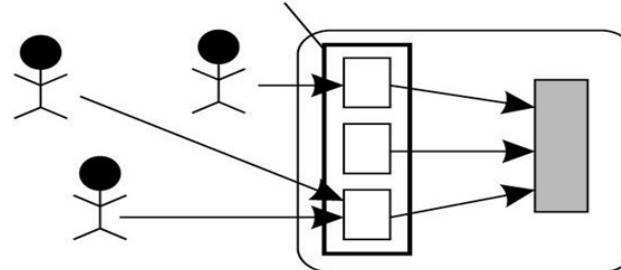
1. Protection against invalid operations on secure data

Data is protected against wrong or invalid operation

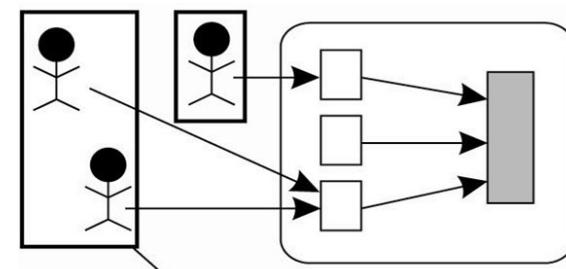


2. Protection against unauthorized invocations

Data is protected against unauthorized invocations

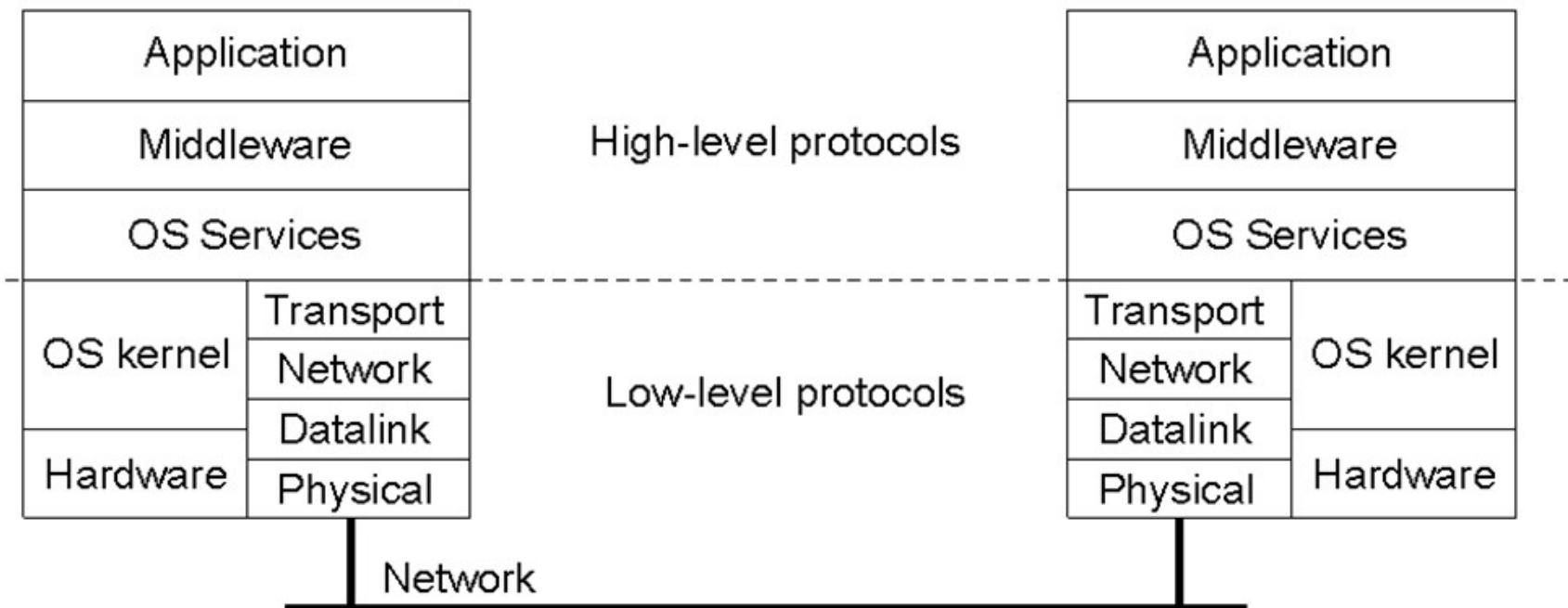


3. Protection against unauthorized users



Design Issue: Layering of Security Mechanisms

- ▶ The logical organization of a distributed system into several layers.
- ▶ A decision is required as to where in the model security the mechanism is to be placed.

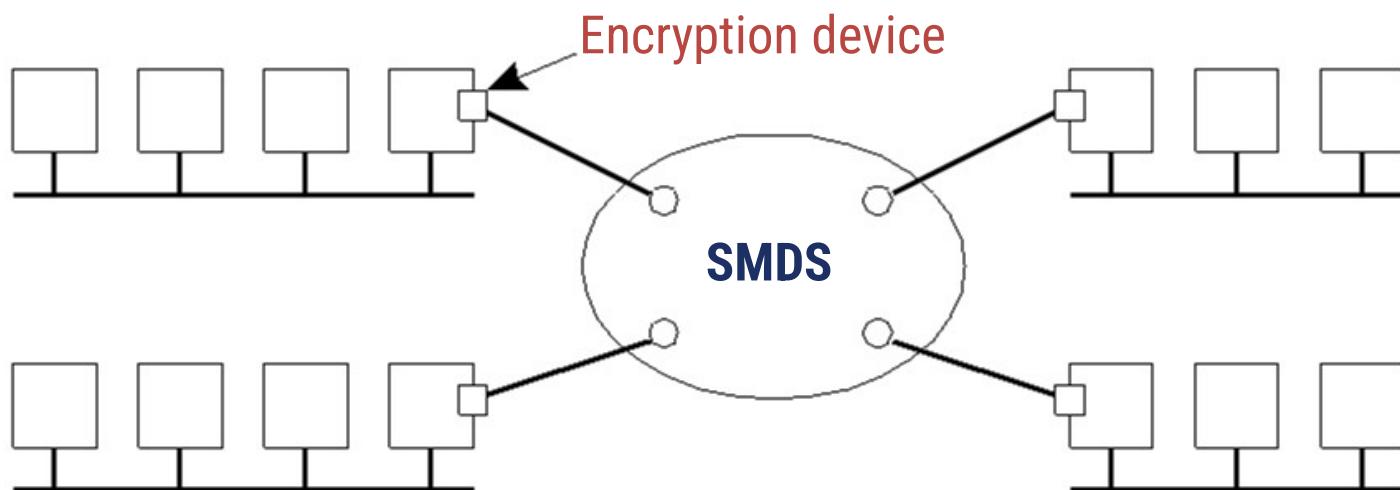


Trust and Security

- ▶ A system is either secure or it is not.
- ▶ Whether a client considers a systems to be secure is a matter of trust.
- ▶ In which layer security mechanisms are placed depends on the trust a client has in how secure the services are in any particular layer.

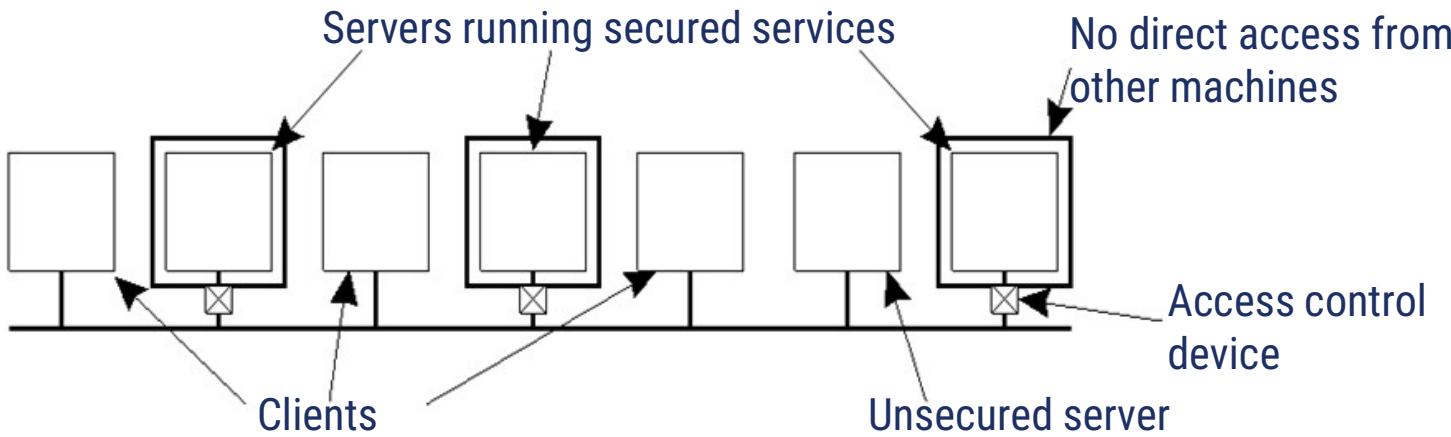
Trust and Security

- ▶ Several sites connected through a wide-area backbone service.
- ▶ Switched Multi-megabit Data Service (SMDS)
- ▶ If the encryption mechanisms cannot be trusted, additional mechanisms can be employed by clients to provide the level of trust required (e.g., using SSL at the Application Layer).



Distribution of Security Mechanisms

- ▶ The **Trusted Computing Base (TCB)** is the set of services/mechanisms within a distributed system required to support a security policy.
- ▶ **Reduced Interfaces for Secure System Components (RISSC)**
 - Prevents clients and their applications from direct access to critical services.
 - Any security-critical server is placed on a separate machine isolated from end-user systems using low-level secure network interfaces.
 - Clients and their applications run on different machines and can access the secured server only through these network interface



Design Issue: Simplicity

- ▶ Designing a secure computer system is considered a difficult task, regardless of whether it is also a DS.
- ▶ The use of a few simple mechanisms that are easily understood and trusted to work would be the ideal situation.
- ▶ Unfortunately, the real world is not this clear cut, as introducing security mechanisms to an already complex system can often matters worse.

Cryptographic Hash Functions

- ▶ A hash function is a unique identifier for any given piece of content. It's also a process that takes plaintext data of any size and converts it into a unique ciphertext of a specific length.
- ▶ A hash function $H(m)$ maps an input message m to a hash value h
- ▶ Message m is of any arbitrary length
- ▶ Hash h is fixed length $\textcolor{red}{h=H(m)}$
- ▶ Often, h is called as the “message digest” of m
- ▶ How does a cryptographic hash function differ from a regular hash function?



Properties of Cryptographic Hash Functions

1. One-Way Function

- Finding hash h from m is easy, but not vice-versa
- Computationally infeasible to find m that corresponds to a given hash h

2. Weak-collision resistance

- Given a message, it is hard to find another message that has the same hash value
- Given m and $h=H(m)$, it is computationally infeasible to find $m' \neq m$ such that $H(m)=H(m')$

3. Strong-collision resistance

- Given a hash function, it is hard to find two messages with the same hash value
- Given $H(.)$, it is computationally infeasible to find two messages m and m' such that $H(m)=H(m')$

Encryption/Decryption Functions

- ▶ Recall: An encryption function $E_K(m_p)$ encrypts a plain-text message m_p to an cipher-text message m_c using a key K

$$m_c = E_K(m_p)$$

- ▶ Similarly, decryption function $D_K(m_c)$:

$$m_p = D_K(m_c)$$

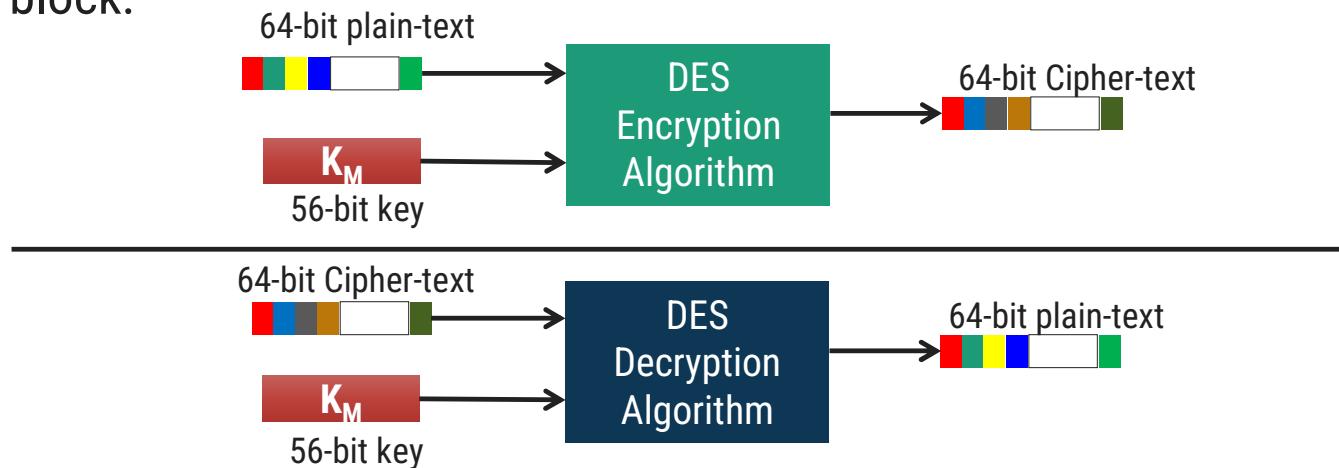
- ▶ Encryption/Decryption Functions have the same properties as Cryptographic Hash Functions
 - One-way functions
 - Weak and Strong collision resistance

Protocols in Cryptosystems

- ▶ **Data Encryption Standard (DES)** : Encryption/Decryption in **Symmetric** Cryptosystems
- ▶ **RSA protocol** : Encryption/Decryption in **Public-Key** cryptosystems
- ▶ **Hybrid Cryptographic protocol** : A combination of **Symmetric** and **Public-Key** based system

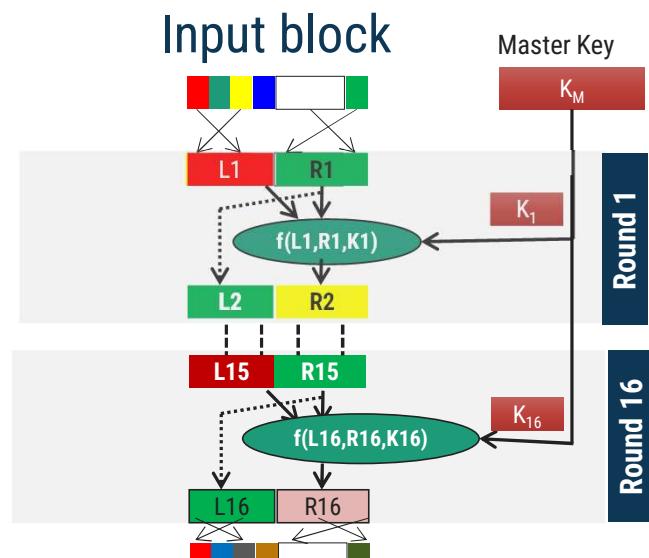
DES Protocol

- ▶ Purpose: Used for **Encryption/Decryption in Symmetric Cryptosystems**
- ▶ DES is designed to operate on 64-bit blocks of data.
- ▶ A block is transformed into an encrypted (64 bit) block of output in 16 rounds, where each round uses a different 48-bit key for encryption.
- ▶ Each of these 16 keys is derived from a 56-bit master key
- ▶ Before an input block starts its 16 rounds of encryption, it is first subject to an initial permutation, of which the inverse is later applied to the encrypted output leading to the final output block.



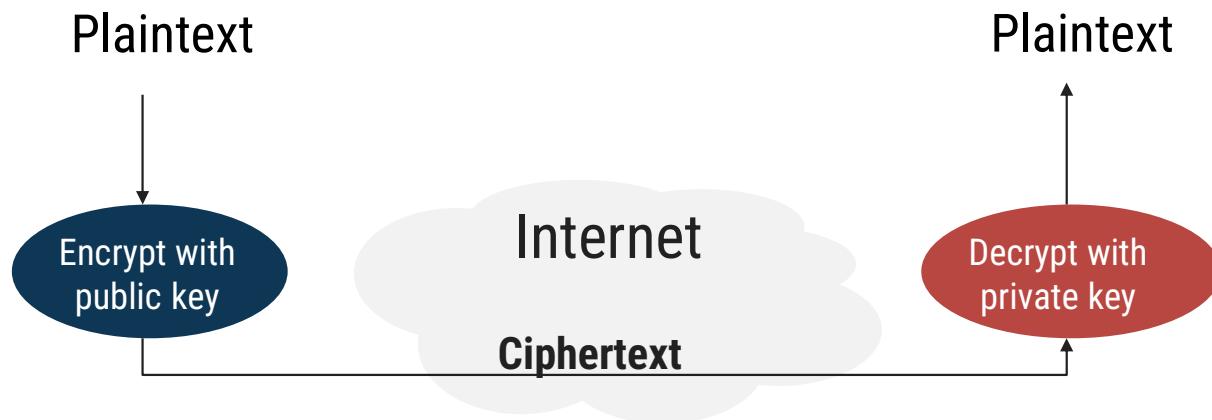
DES Encryption Algorithm

1. Permute a 64-bit block
2. 16 rounds of identical operations
3. In each round i
 - i. Divide the block into 2 halves L_i and R_i
 - ii. Extract 48-bit key K_i from K_M
 - iii. Mangle the bits in L_i and R_i using K_i to produce R_{i+1}
 - iv. Extract R_i as L_{i+1}
4. Perform an inverse permutation on the block $L_{16}-R_{16}$ to produce the encrypted output block



RSA protocol

- ▶ Sender uses a public key : Advertised to everyone
- ▶ Receiver uses a private key



RSA encryption algorithm

1. Select two large prime numbers, p and q .
2. Multiply these numbers to find $n = p \times q$, where n is called the modulus for encryption and decryption.
3. Choose a number e less than n , such that n is relatively prime to $(p - 1) \times (q - 1)$.
 - It means that e and $(p - 1) \times (q - 1)$ have no common factor except 1.
 - Choose "e" such that $1 < e < \varphi(n)$, e is prime to $\varphi(n)$
 - $\gcd(e, \varphi(n)) = 1$
4. If $n = p \times q$, then the public key is $\langle e, n \rangle$.
 - A plaintext message m is encrypted using public key $\langle e, n \rangle$.
 - To find ciphertext from the plain text following formula is used to get ciphertext C .
 - $C = m^e \text{ mod } n$ Here, m must be less than n .
 - A larger message ($>n$) is treated as a concatenation of messages, each of which is encrypted separately.

RSA encryption algorithm

5. To determine the private key, we use the following formula to calculate the d such that:
 - $D_e \text{ mod } \{(p - 1) \times (q - 1)\} = 1$ Or
 - $D_e \text{ mod } \varphi(n) = 1$
6. The private key is $\langle d, n \rangle$. A ciphertext message c is decrypted using private key $\langle d, n \rangle$.
 - To calculate plain text m from the ciphertext c following formula is used to get plain text m .
 - $m = c^d \text{ mod } n$

RSA encryption algorithm- An Example

► **Step 1:** Select two large prime numbers, p, and q.

- p = 7
- q = 11

► **Step 2:** Multiply these numbers to find $n = p \times q$, where n is called the modulus for encryption and decryption.

- First, we calculate
- $n = p \times q$
- $n = 7 \times 11$
- $n = 77$

RSA encryption algorithm- An Example

► **Step 3:** Choose a number e less than n , such that n is relatively prime to $(p - 1) \times (q - 1)$. It means that e and $(p - 1) \times (q - 1)$ have no common factor except 1. Choose "e" such that $1 < e < \varphi(n)$, e is prime to $\varphi(n)$, $\text{gcd}(e, \varphi(n)) = 1$.

- Second, we calculate
- $\varphi(n) = (p - 1) \times (q - 1)$
- $\varphi(n) = (7 - 1) \times (11 - 1)$
- $\varphi(n) = 6 \times 10$
- $\varphi(n) = 60$
- Let us now choose relative prime e of 60 as 7.
- Thus the public key is $\langle e, n \rangle = (7, 77)$

RSA encryption algorithm- An Example

► **Step 4:** A plaintext message m is encrypted using public key $\langle e, n \rangle$. To find ciphertext from the plain text following formula is used to get ciphertext C.

- To find ciphertext from the plain text following formula is used to get ciphertext C.
- $C = m^e \text{ mod } n$
- $C = 9^7 \text{ mod } 77$
- $C = 37$

► **Step 5:** The private key is $\langle d, n \rangle$. To determine the private key, we use the following formula d such that:

- $D_e \text{ mod } \{(p - 1) \times (q - 1)\} = 1$
- $7^d \text{ mod } 60 = 1$, which gives $d = 43$
- The private key is $\langle d, n \rangle = (43, 77)$

RSA encryption algorithm- An Example

► **Step 6:** A ciphertext message c is decrypted using private key $\langle d, n \rangle$. To calculate plain text m from the ciphertext c following formula is used to get plain text m .

- $m = c^d \bmod n$
- $m = 37^{43} \bmod 77$
- $m = 9$
- In this example, Plain text = 9 and the ciphertext = 37

RSA encryption algorithm- Overview

- ▶ Choose two large prime numbers p & q
- ▶ Compute $n=pq$ and $z=(p-1)(q-1)$
- ▶ Choose number e, less than n, which has no common factor (other than 1) with z
- ▶ Find number d, such that $ed - 1$ is exactly divisible by z
- ▶ Keys are generated using n, d, e
 - Public key is (n,e)
 - Private key is (n, d)
- ▶ Encryption: **$c = m^e \text{ mod } n$**
 - m is plain text
 - c is cipher text
- ▶ Decryption: **$m = c^d \text{ mod } n$**
- ▶ Public key is shared and the private key is hidden

RSA protocol

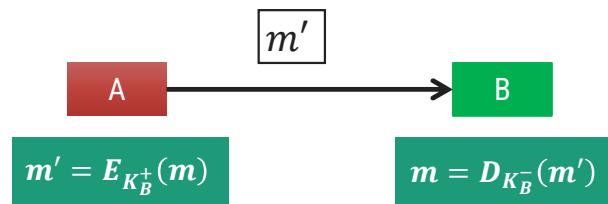
- ▶ Invented by Rivest, Shamir and Adleman (RSA) as a protocol for Public-key systems
- ▶ Approach:
 1. Choose two very large prime numbers, p and q
 2. Compute $n = pq$
 3. Compute $z = (p-1)(q-1)$
 4. Choose a number e that is relatively prime to z
 - e is co-prime to z
 5. Compute the number d such that $de \% z = 1$
 - This is equivalent to finding $de = 1 + kz$ for some integer k
- ▶ Depending on the requirement, d and e can be used as public and private keys
 - d is used for decryption; e is used for encryption

Example
$p=7; q=19$
$n = 7*19 = 133$
$z = 6*18 = 108$
$e=5$
$d=65$ for $m=325$

RSA protocol for encryption- An Example

- ▶ Scenario: Alice (A) wants to send to Bob (B)
- ▶ Problem: Only Bob should be able to decrypt the message
- ▶ Given d and e are the two keys computed by RSA, which row indicates correct choice of keys?

Correct/ Incorrect	Alice		Bob	
	Private K_A^-	Public K_A^+	Private K_B^-	Public K_B^+
X	d	e		
X	e	d		
Correct			d	e
X			e	d



RSA protocol for encryption- An Example

At the sender:

- ▶ Split the message into fixed-length blocks of size s ($0 \leq s < n$)
- ▶ For each block m_i
 - Sender calculates the encrypted message c_i such that $c_i = m_i^e \pmod{n}$
 - Send c_i to the receiver

Calculated Values
 $p=7; q=19; n=133; d=65; e=5$

Example
$s = 132$
$m_i = 6$
$c_i = 6^5 \pmod{133}$ = 62
$m_i = 62^{65} \pmod{133}$ = 6

At the receiver:

- ▶ Receive c_i from sender
- ▶ For each block c_i
 - Compute actual message $m_i = c_i^d \pmod{n}$
- ▶ Merge all c_i 's to obtain the complete message

Secure Channels

- ▶ A Secure Channel is an abstraction of **secure communication** between communication parties in a DS
- ▶ A Secure Channel protects senders and receivers against:
 - **Interception** : By ensuring confidentiality of the sender and receiver
 - **Modification and Fabrication of messages** : By providing mutual authentication and message integrity protocols

Authentication

- ▶ Consider a scenario where Alice wants to set up a secure channel with Bob
 - Alice sends a message to Bob (or trusted third party) for mutual authentication
 - Message integrity should be ensured for all communication between Alice and Bob
 - Generate a “session key” to be used between Alice and Bob
 - Session-keys ensure integrity and confidentiality
 - When the channel is closed, the session key is destroyed

Mutual Authentication Protocols

Three types of Mutual Authentication Protocols:

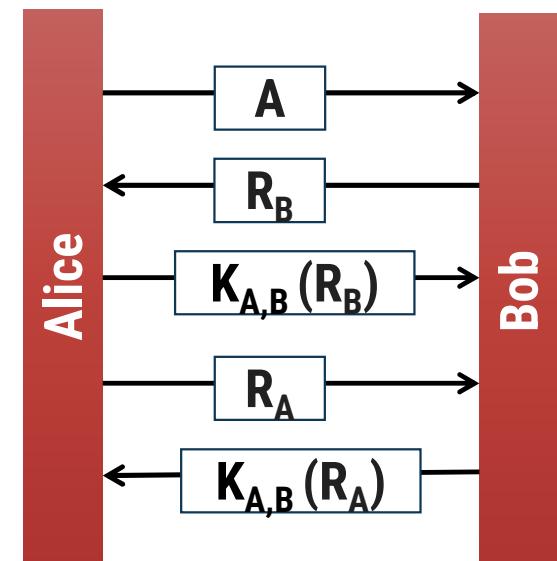
1. Shared Secret Key based Authentication
2. Authentication using a Key Distribution Center
3. Authentication using Public-key Cryptography

Shared Secret Key based Authentication

- ▶ Ensure data message integrity exchanged after authentication use secret-key cryptography with session keys.
- ▶ The scheme is also known as “*Challenge-Response protocol*”
- ▶ Let $K_{A,B}$ be the shared secret key between Alice and Bob

The Challenge-Response Protocol

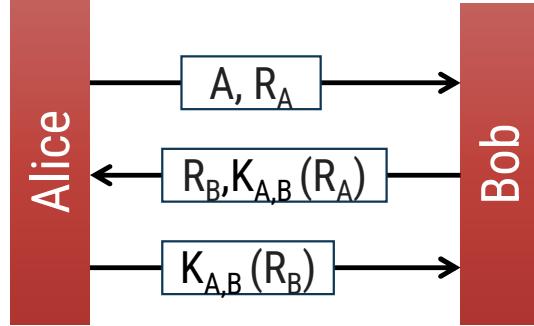
1. 'A' sends her identity to 'B'
2. 'B' sends a challenge R_B back to 'A'
3. 'A' responds to the challenge by encrypting R_B with $K_{A,B}$ (denoted by $K_{A,B}(R_B)$), and sending it back to 'B'
4. 'A' challenges 'B' by sending R_A
5. 'B' responds to the challenge by sending the encrypted message $K_{A,B}(R_A)$



A and B are mutually authenticated

Optimization of the Authentication based on a shared secret key

- ▶ Session key - is a shared (secret) key that is used to encrypt messages for integrity
- ▶ Generally used only for as long as the channel exists



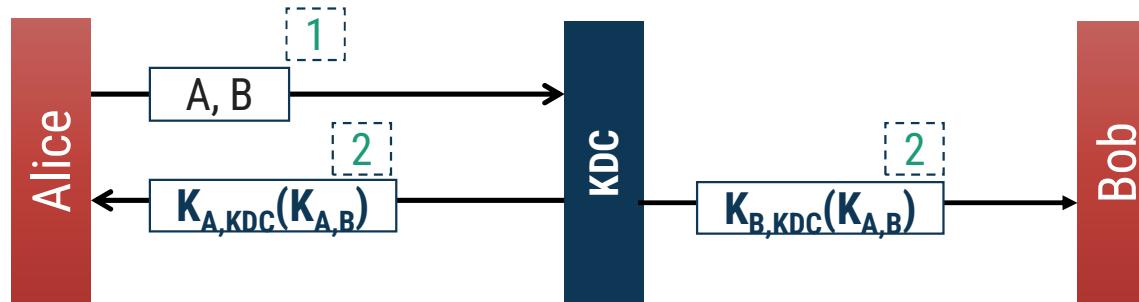
Bob and Alice – Taking Shortcuts

Authentication Using a Key Distribution Center (KDC)

- ▶ Shared secret key based authentication is not scalable
 - Each host has exchange keys with every other host
 - Complexity: $O(N^2)$
- ▶ The complexity is reduced by electing one node as “Key Distribution Center” (KDC)
 - KDC shares a secret-key with every host
 - No pair of hosts need to share a key
 - Complexity: $O(N)$

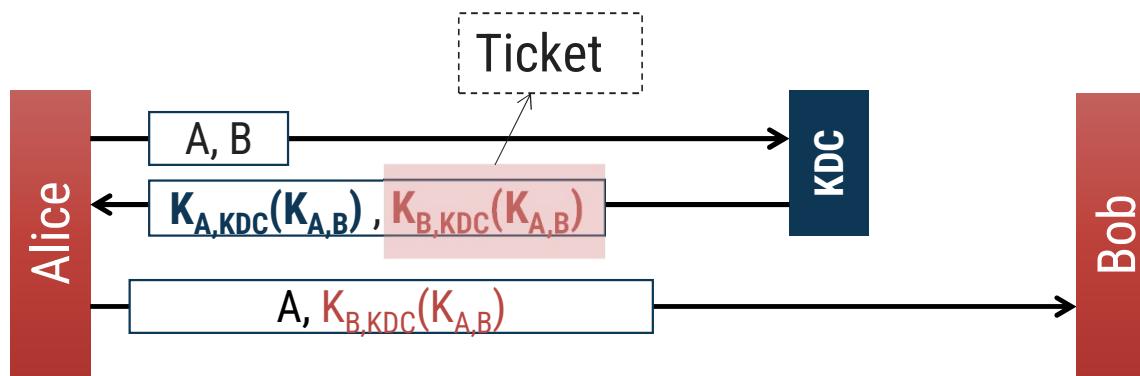
Authentication Using a Key Distribution Center

- ▶ If Alice wants to set up a secure channel with Bob, she can do so with the help of a (trusted) KDC.
- ▶ The KDC hands out a key to both Alice and Bob that they can use for communication



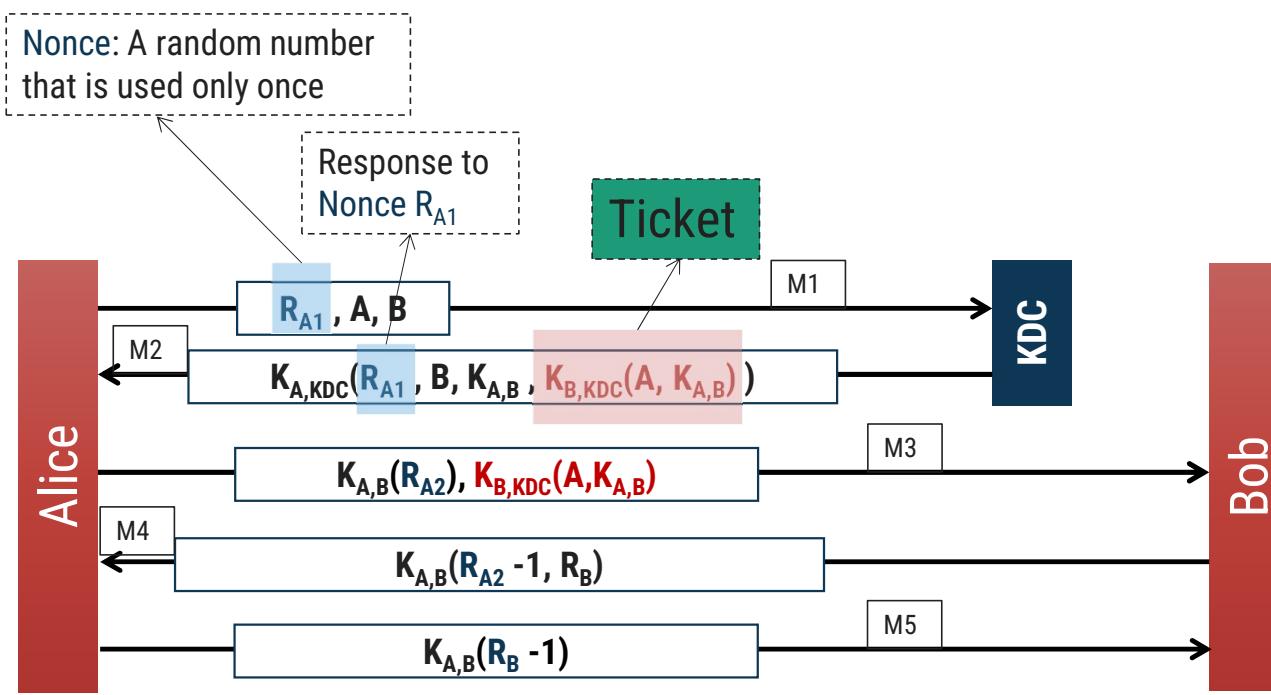
1. Alice sends a message to the KDC, telling it that she wants to talk to Bob.
2. The KDC returns a message containing a shared secret key KA,B that she can use.
 - The message is encrypted with the secret key KA,KDC that Alice shares with the KDC
3. The KDC sends KA,B to Bob, but now encrypted with the secret key KB,KDC it shares with Bob

Authentication Using a Key Distribution Center using Ticket



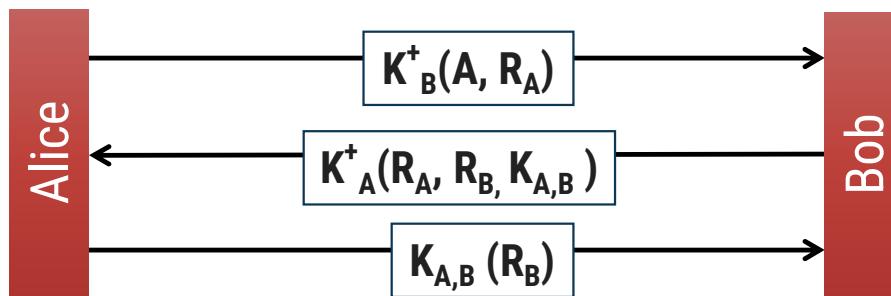
Needham-Schroeder Authentication Protocol

- ▶ A multi-way challenge response protocol
- ▶ Protocol is a variant of a well-known example of an authentication protocol using a KDC, known as the Needham-Schroeder authentication protocol.



Authentication Using Public-Key Cryptography

- ▶ Mutual authentication in a public-key cryptosystem.
- ▶ Note that the KDC is missing
- ▶ But, this assumes that some mechanism exists to verify everyone's public key.
- ▶ Recall:
 - K_N^+ : Public key of user N
 - K_N^- : Private key of user N

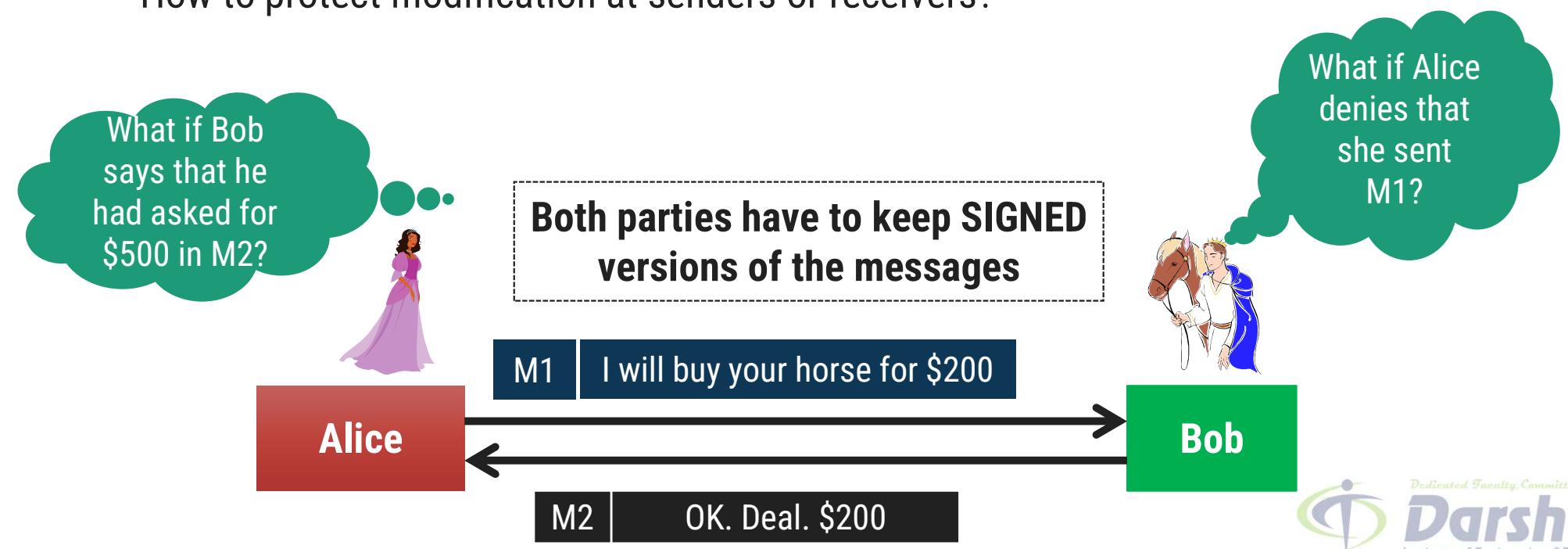


Session Keys

- ▶ During the establishment of a secure channel, after the authentication phase has completed, the communicating parties generally use a unique shared session key for confidentiality.
- ▶ The session key is safely discarded when the channel is no longer used.
- ▶ Why not using the same keys for confidentiality as those that are used for setting up the secure channel?
 - Cryptographic keys are subject to “wear and tear” just like ordinary keys.
 - Protection against replay attacks
 - If a key is compromised, only a single session is affected

Message Integrity and Confidentiality

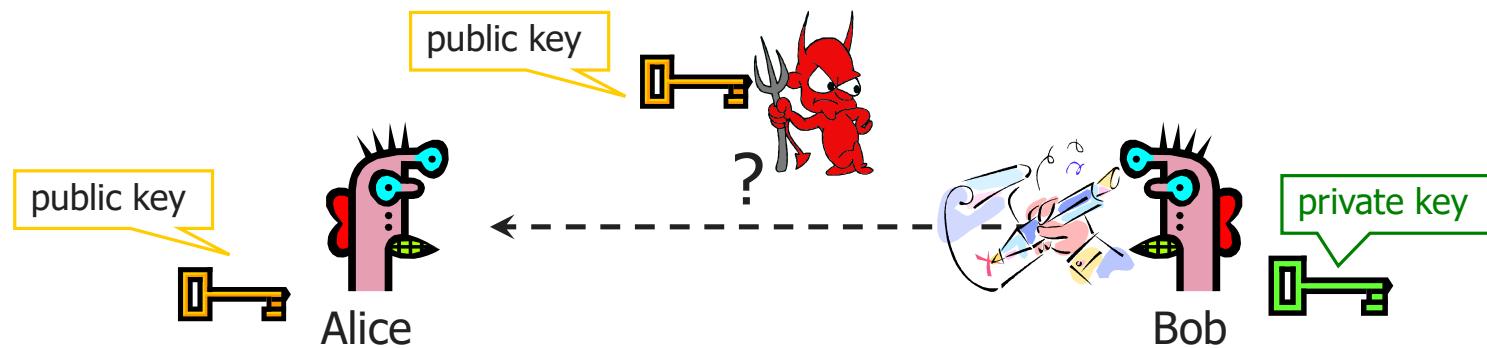
- ▶ Encryption is used for providing Confidentiality
- ▶ But, how to provide Message Integrity?
 - Encryption protects message modification by third party adversaries
 - How to protect modification at senders or receivers?



Message Integrity and Confidentiality

- ▶ Besides authentication, a secure channel should also provide guarantees for message integrity and confidentiality
- ▶ Confidentiality is easily established by simply encrypting a message before sending it.
- ▶ Protecting a message against modifications is somewhat more complicated
- ▶ Digital Signatures:
 - Digitally sign a message in such a way that the signature is uniquely tied to its content
 - Several ways to place digital signatures:
 - ✓ Use a public-key cryptosystem such as RSA
 - ✓ Use a message digest

Digital Signatures: Basic Idea



► Given:

- Everybody knows Bob's public key
- Only Bob knows the corresponding private key

► Goal: Bob sends a “digitally signed” message

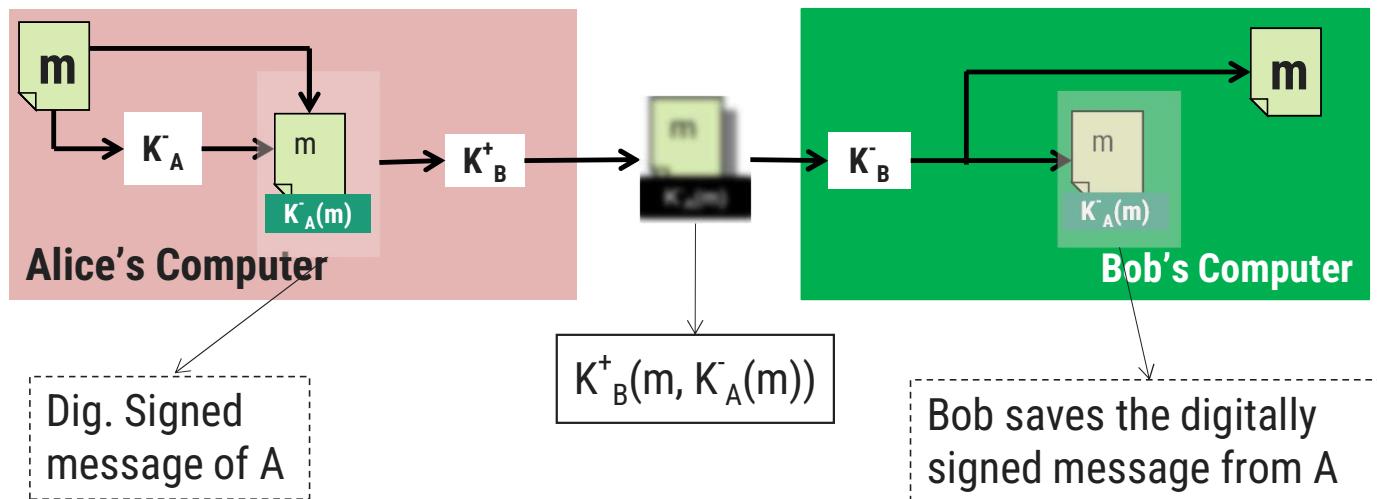
1. To compute a signature, must know the private key
2. To verify a signature, only the public key is needed

Digital Signatures

- ▶ Digital signatures are the public-key primitives of message authentication.
- ▶ In the physical world, it is common to use handwritten signatures on handwritten or typed messages.
- ▶ They are used to bind signatory to the message.
- ▶ Digital signature is a technique that binds a person/entity to the digital data.
- ▶ This binding can be independently verified by receiver as well as any third party.
- ▶ Digital signature is a cryptographic value that is calculated from the data and a secret key known only by the signer.

Digital Signatures

- ▶ The sender and the receiver will keep digitally signed copies of the messages
- ▶ One way of creating digital signatures is by using RSA

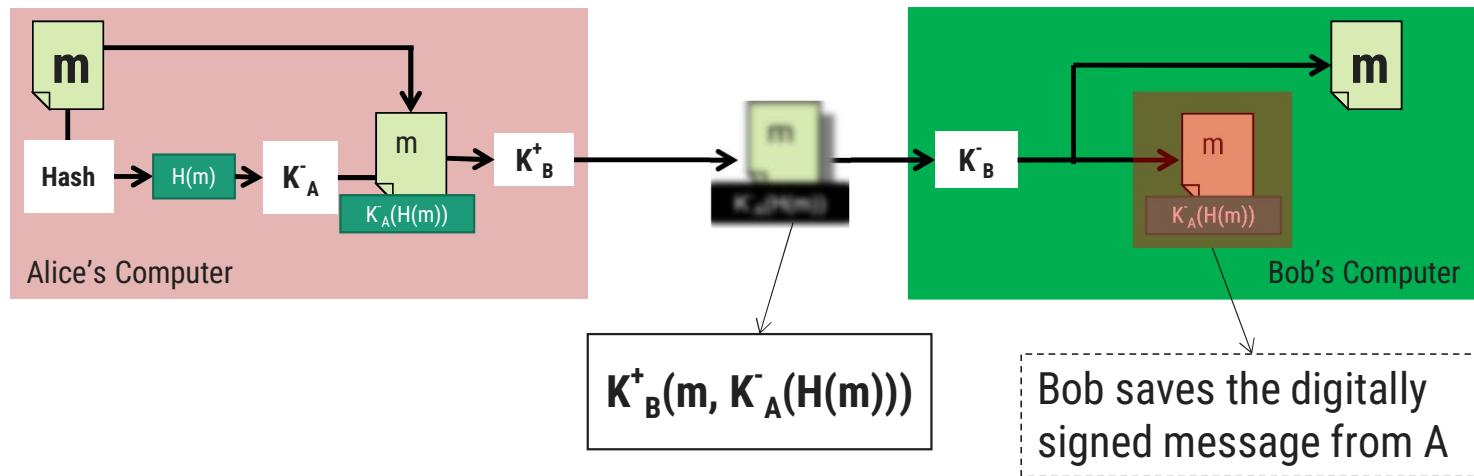


Issues with Naïve Digital Signatures

- ▶ The validity of the digitally signed message holds as long as A's private key remains a secret
 - 'A' can claim that her private key was stolen
- ▶ If 'A' changes her private key, the signed message becomes invalid
 - A centralized authority can keep track of when keys are changed
- ▶ Encrypting the entire message with private key is costly
 - RSA is slower, and encryption of long data is not preferred
 - Any ideas on how to improve?

Digital Signature with Message Digest

- ▶ Instead of attaching the encrypted message, attach the message digest



Bob can verify the signature by comparing the hash of received message, and the hash attached in the digitally signed message

Secure Group Communication

- ▶ Scenario: Client has to communicate with a group of replicated servers
 - Replication can be for fault-tolerance or performance improvement
- ▶ Client expects that the response is secure
- ▶ Naïve Solution:
 - Client collects the response from each server
 - Client authenticates every response
 - Client takes a majority vote
- ▶ Drawback of naïve approach
 - Replicated servers should be transparent to the client
 - Reiter et al. proposed a transparent and secure group authentication protocol (Secure Group Authentication Protocol)

Secure Group Authentication Protocol

- ▶ Problem: Authenticate a group of replicated servers at the client in a transparent way
- ▶ Main Idea:
 - Multiple servers share a secret
 - None of them know the entire secret
 - Secret can be revealed only if all/most of them cooperate
- ▶ The approach is similar to k-fault tolerant systems
 - Tolerate intrusion (faults) of c servers out of N servers if $c \leq N$

Secure Replicated Servers

► Secure and Transparent Replicated Servers

► Example:

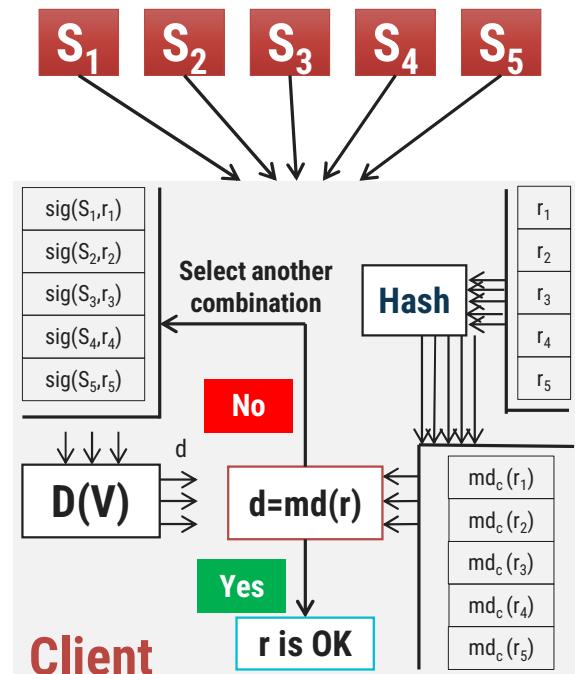
- Given a securely replicated group of servers
- Each server accompanies its response with a digital signature.
- If r_i is the response from server S_i , let $md(r_i)$ denote the message digest computed by server S_i .
- This digest is signed with server S_i 's private key k_i

► Want to protect the client against at most c corrupted servers => the server group should be able to tolerate corruption by at most c servers, and still be capable of producing a response that the client can put its trust in

Secure Replicated Servers

- ▶ For each client call:
 - Each server S_i returns the response (r_i)
 - In addition, it sends a signed message digest: $\text{sig}(S_i, r_i) = K_{S_i}(\text{md}(r_i))$
- ▶ For each response, client computes $\text{md}_c(r_i) = H(r_i)$
- ▶ Hence, client has N triplets $R_i = < r_i, \text{md}_c(r_i), \text{sig}(S_i, r_i) >$
- ▶ Client takes $c+1$ combinations of triplets
- ▶ For each combination vector [R, R' and R'']
 - Computes and creates a single digest using a special decryption function D(V)
 - $d = D(V) = D(\text{sig}(S, r), \text{sig}(S', r'), \text{sig}(S'', r''))$
 - NOTE: d is a vector of responses
 - If $d[x] = \text{respective } \text{md}_c(r_i)$, then r_i is correct

Parameters: $N=5; c=2$

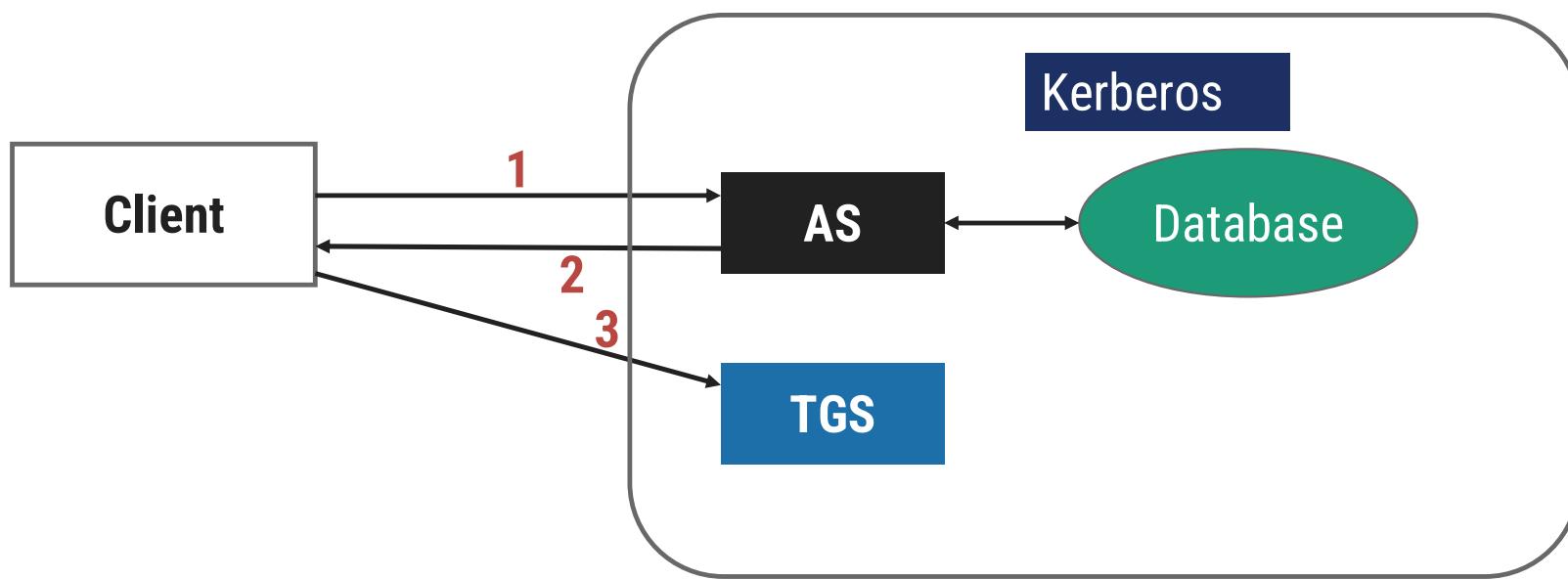


Authentication in Kerberos

- ▶ Network authentication protocol.
- ▶ Developed at MIT in the mid 1980s.
- ▶ A secret key based service for providing authentication in open networks.
- ▶ Provides strong authentication for client-server applications.
- ▶ There are four parties involved in the Kerberos protocol:
 - User – Alice who uses the client workstation
 - Real Server - The server (Bob) provides services for the user (Alice).
 - Authentication Server (AS) – It is the KDC in the Kerberos protocol. Each user registers with the AS and is granted a user identity and a password. The AS verifies the user, issues a session key to be used between Alice and TGS and sends a ticket to TGS.
 - Ticket Granting Server (TGS) – Issues a ticket for the real server. It provides the session key(KAB) between Alice and Bob.

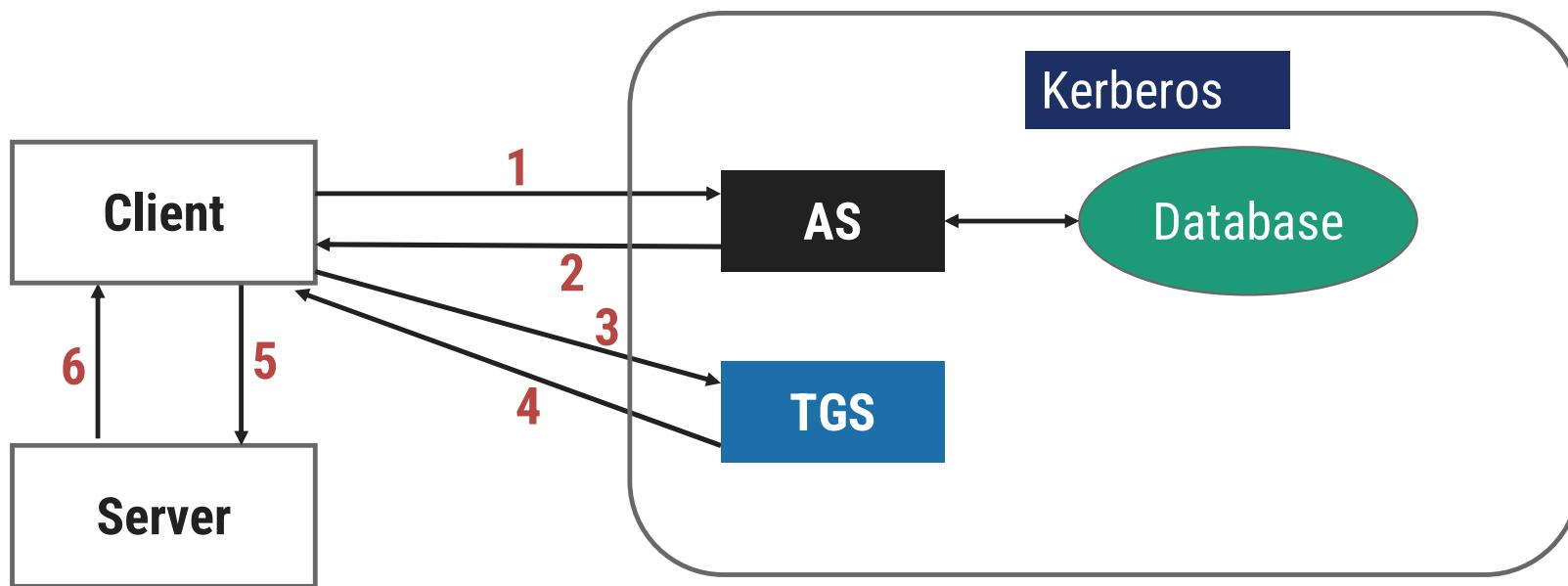
Authentication in Kerberos

- ▶ Step-1: User logon and request services on host. Thus user request for ticket-granting-service.
- ▶ Step-2: Authentication Server verifies user's access right using database and then gives ticket-granting-ticket and session key. Results are encrypted using Password of user.
- ▶ Step-3: Decryption of message is done using the password then send the ticket to Ticket Granting Server. The Ticket contain authenticators like user name and network address.

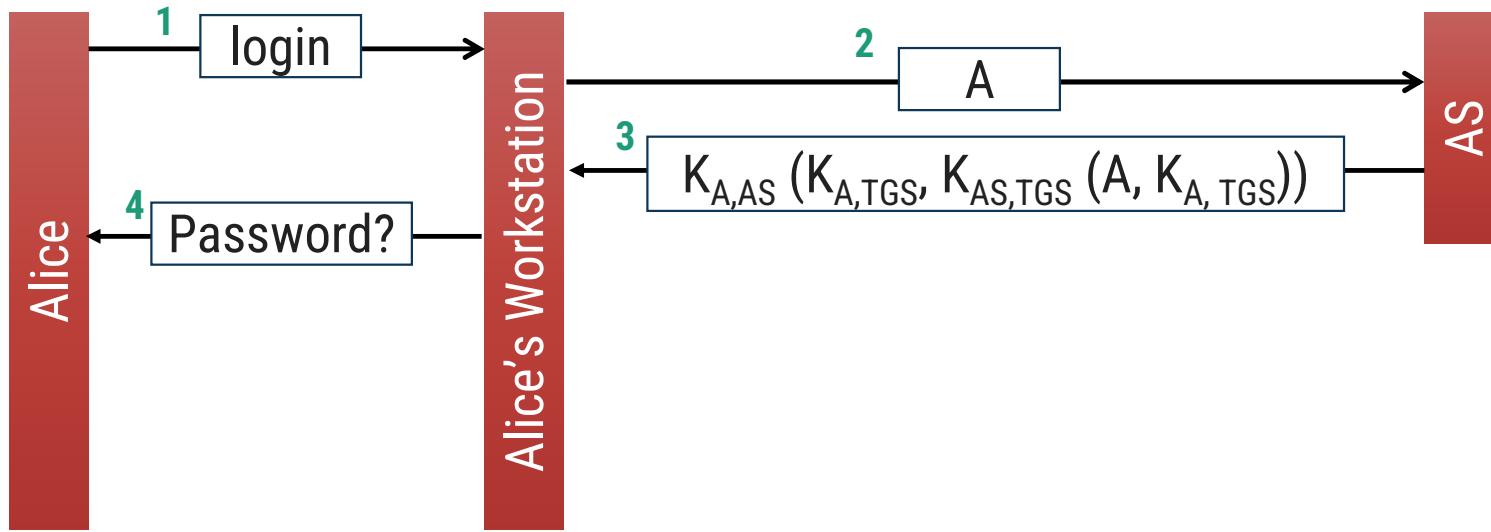


Authentication in Kerberos

- ▶ Step-4: Ticket Granting Server decrypts the ticket send by User and authenticator verifies the request then creates the ticket for requesting services from the Server.
- ▶ Step-5: User send the Ticket and Authenticator to the Server.
- ▶ Step-6: Server verifies the Ticket and authenticators then generate the access to the service. After this User can access the services.

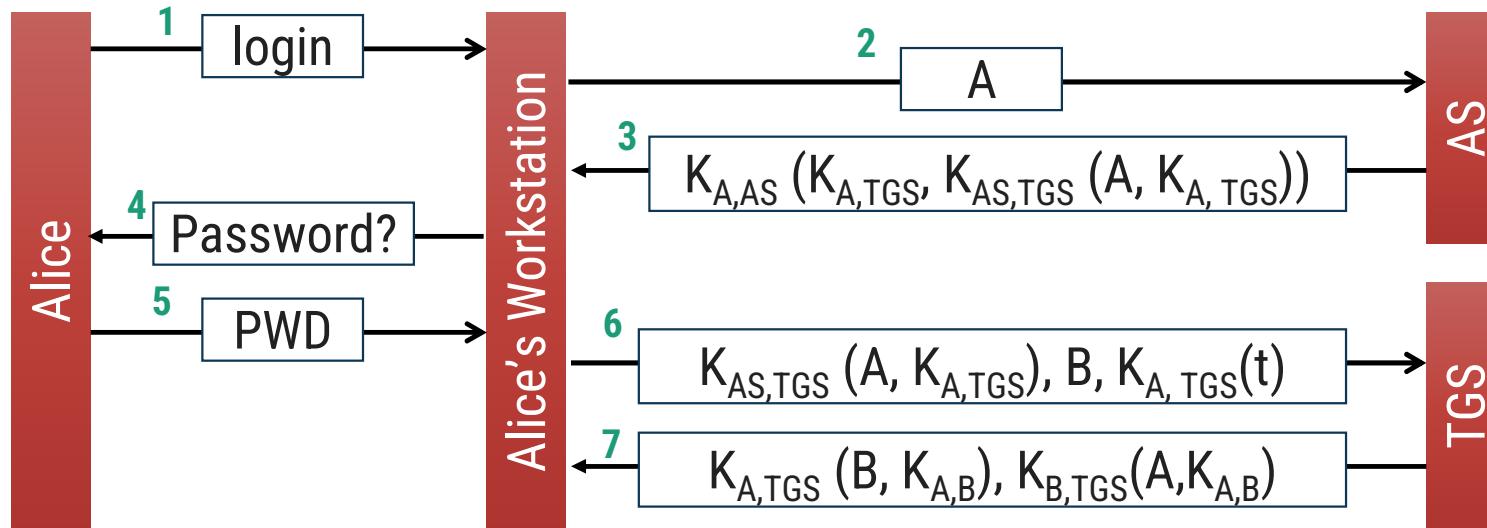


Authentication in Kerberos



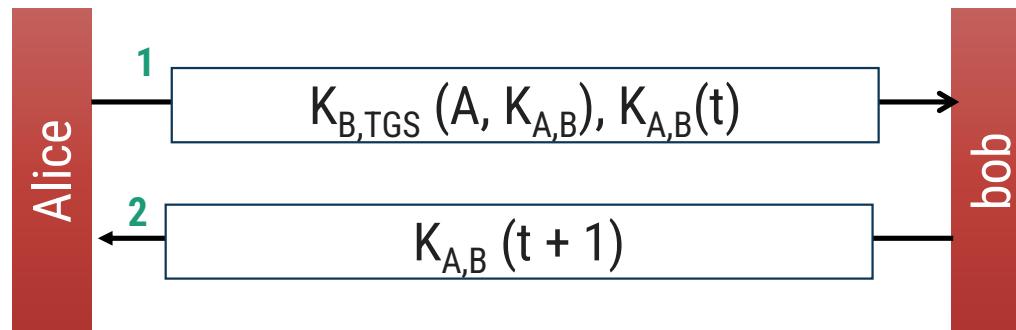
- ▶ Message 1 - Alice types in her login name at a workstation.
- ▶ Message 2 - contains login name and is sent to the AS.
- ▶ Message 3 - contains the session key $K_{A,TGS}$ and the ticket $K_{AS,TGS}(A, K_{A,TGS})$.
 - To ensure privacy, message 3 is encrypted with the secret key $K_{A,AS}$ shared between Alice and the AS.
- ▶ Message 4 - workstation prompts Alice for her password

Authentication in Kerberos



- ▶ Message 5 – PWD returned to workstation which generates shared key $K_{A,AS}$ and find session key $K_{A,TGS}$.
- ▶ Message 6 –to talk to Bob, she requests the TGS to generate a session key for Bob.
 - The fact that Alice has the ticket $K_{AS,TGS}(A, K_{A,TGS})$ proves that she is Alice.
 - Message also contains a timestamp, t, encrypted with the secret key shared between Alice and the TGS.

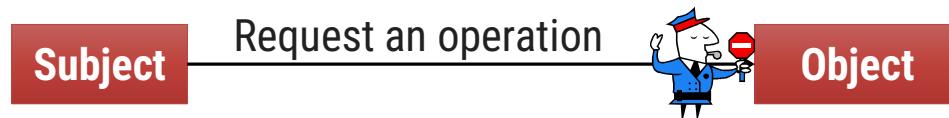
Setting up a secure channel in Kerberos



1. Alice sends to Bob a message containing the ticket she got from the TGS, along with an encrypted timestamp.
2. When Bob decrypts the ticket, he notices that Alice is talking to him, because only the TGS could have constructed the ticket
 - He also finds the secret key $K^{A,B}$, allowing him to verify the timestamp.
 - At that point, Bob knows he is talking to Alice and not someone maliciously replaying message 1.
 - By responding with $K^{A,B}(t + 1)$, Bob proves to Alice that he is indeed Bob.

Access Control

- ▶ When a secure channel is set up between a client and a server, the client can issue requests to the server.
- ▶ A request can be carried out only if the client has sufficient access rights for that invocation.
- ▶ Access Control (AC) verifies the access rights of a subject requesting an access to an object



- ▶ A General model for Access Control



- ▶ How is Access Control different from Authorization?
 - **Authorization** = granting access rights
 - **Access Control** = verifying access rights

Access Control

► Many access control models:

- Access Control Matrix
- Access Control List (Capability List)
- Firewalls

Access Control Matrix

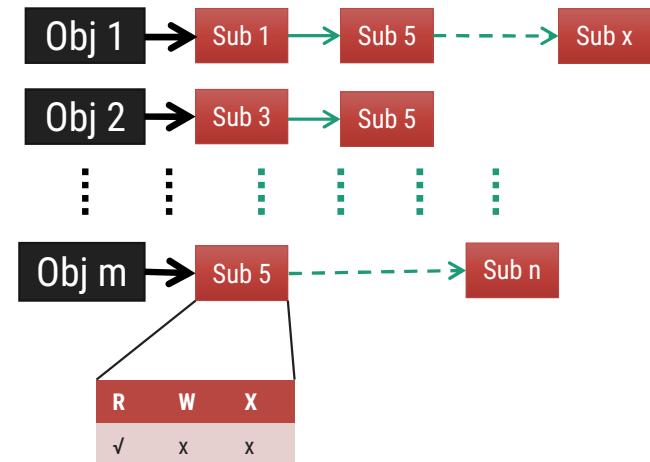
- ▶ A matrix M that keeps track of which operations can a subject invoke on an object
- ▶ $M[s,o]$ lists which operations subject s can perform on object o
- ▶ The reference monitor will look up AC Matrix before authorizing the request
- ▶ Scalability of AC Matrix
- ▶ For a large DS with many users, most of the entries are empty
- ▶ Two implementations of AC Matrix:
 1. Access Control Lists
 2. Capabilities

		Objects		
		01	...	0m
Subjects	S1			
	S2			
Sn				
	R	W	X	
	✓	X	X	

Access Control List and Capabilities

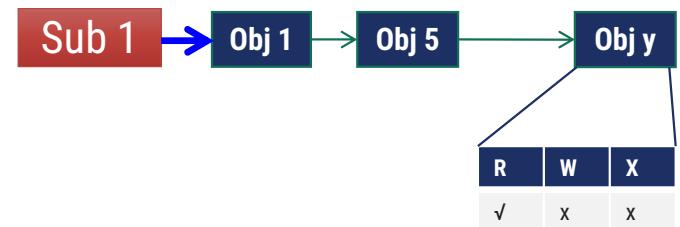
Access Control List

- ▶ AC Matrix is stored as a list
 - Similar to how a graph of nodes and edges are stored as an adjacency list



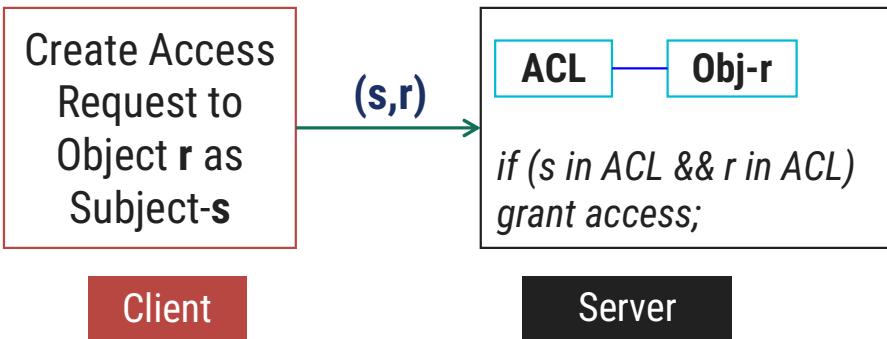
Capabilities

- ▶ AC Matrix can also be decomposed row-wise
- ▶ Capability for a user is a list of objects and the associated permissions for a given user
- ▶ The 'capability' object can be issued by Reference Monitor to a user, and can reside at the user side
 - It should be protected (digital signature)

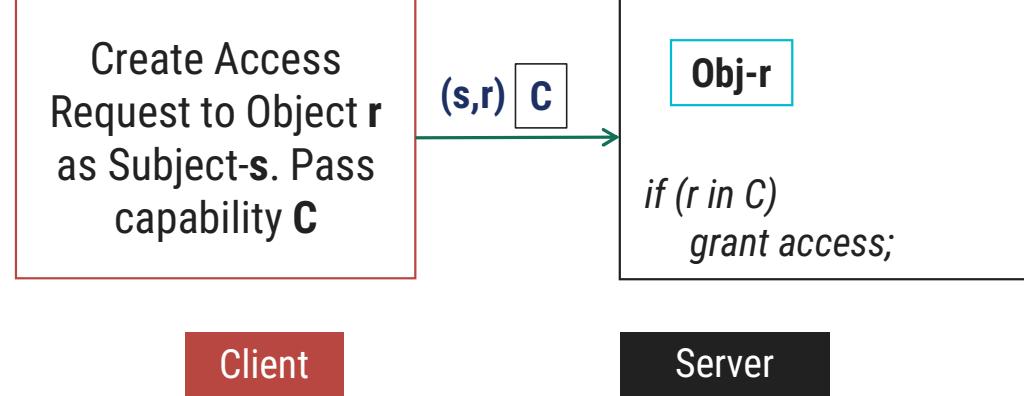


Comparison of ACL and Capabilities

Using an ACL (Access Control List)



Using Capabilities (CL- Capability List)



- The server stores the ACL

- The server is not interested in whether it knows the client
- Server should check Capability

Protection Domains

- ▶ Access Control using ACL and Capabilities is still not scalable in very large DS
- ▶ Reduce information by means of protection domains
- ▶ Organizing Permissions:
 - Object permission is given to a protection domain (PD)
 - Each PD row consists of (object, permission) pair
- ▶ User Management:
 - Group uses into a “protection domain”
 - Users belong to one or more protection domain
 - E.g., employees, admin, guests
 - Before granting the access, check permissions from user's PDs

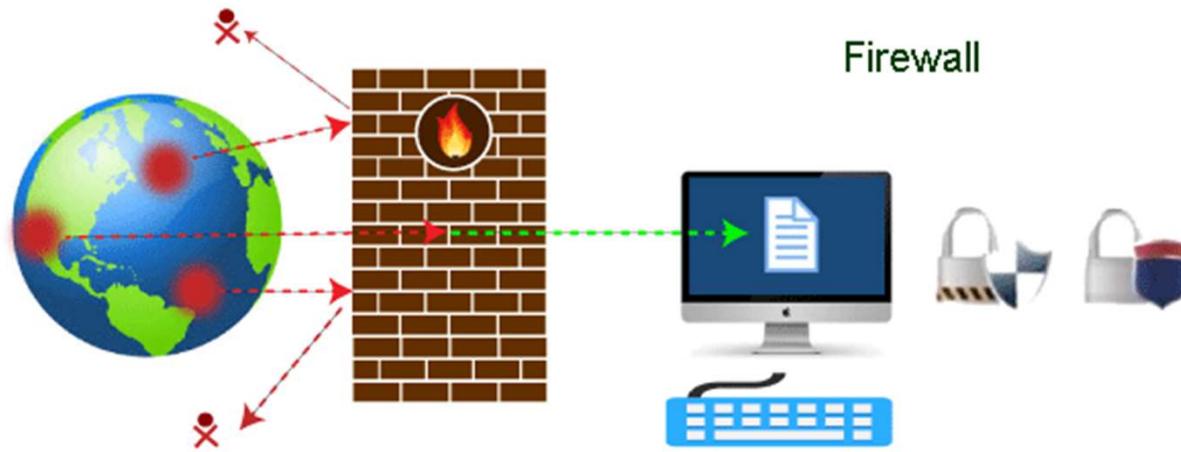
Protection Domains	Objects		
	01	0m	
	PD1		
	PD2		
PDn		1	
PD: Admin			
R	W	X	
✓	✓	✓	
PD: Employees			
R	W	X	
✓	x	✓	
PD: Guests			
R	W	X	
x	x	x	

Controlling Outsider Attacks

- ▶ Above discussed schemes are good for preventing against small number of users who belong to a DS
- ▶ But, what if any user on the Internet can access the resources of DS
 - Search engines, Sending mails, ...
- ▶ Above approaches are complicated, inefficient and time-consuming when a DS is exposed to the rest-of-the-world
- ▶ Two coarse grained Access Control schemes
 - Firewall
 - Controlling Denial of Service (DoS) attacks

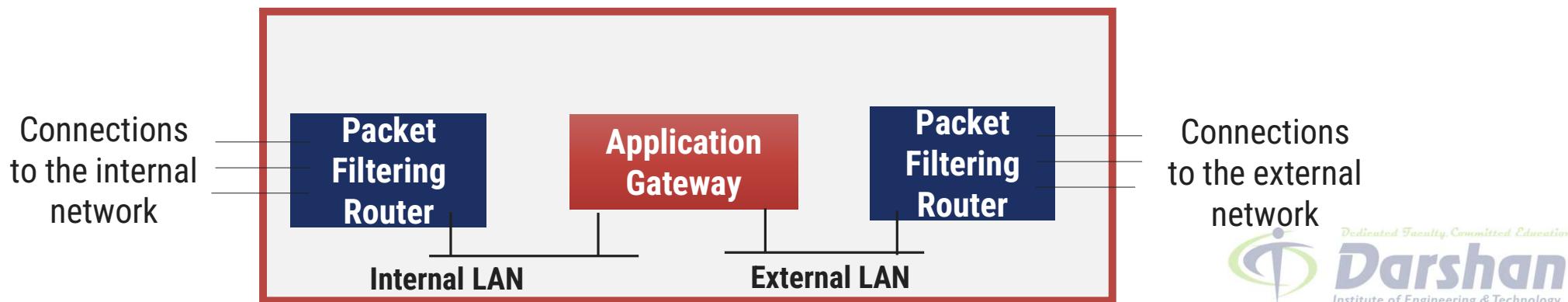
Firewall

- ▶ A firewall can be defined as a special type of network security device or a software program that monitors and filters incoming and outgoing network traffic based on a defined set of security rules.
- ▶ It acts as a barrier between internal private networks and external sources (such as the public Internet).
- ▶ The primary purpose of a firewall is to allow non-threatening traffic and prevent malicious or unwanted data traffic for protecting the computer from viruses and attacks.



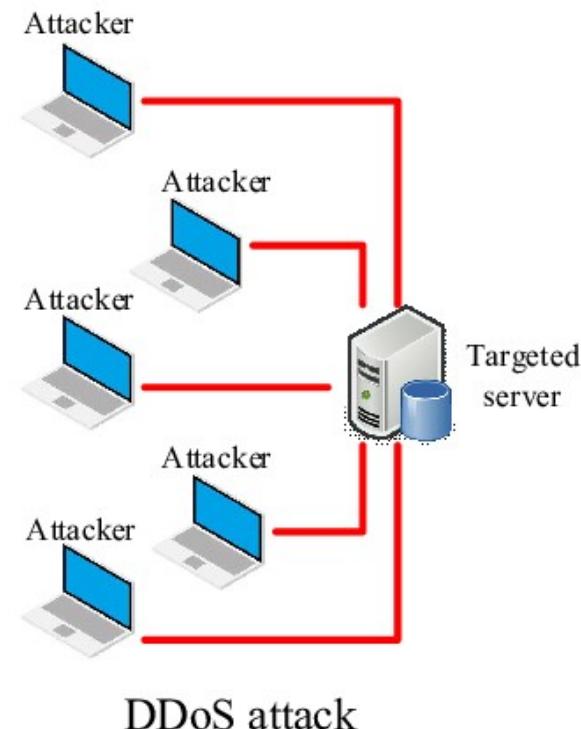
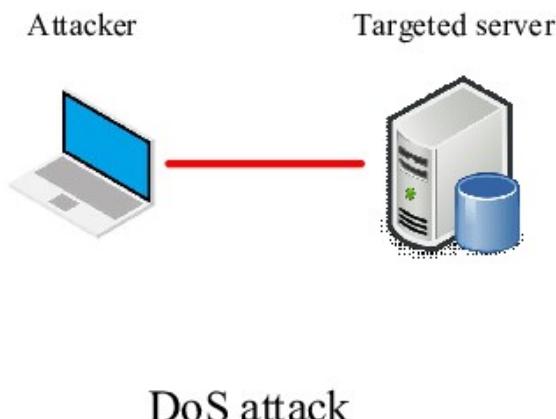
Firewall

- ▶ A Firewall is a special kind reference monitor to control external access to any part of a distributed system.
- ▶ Firewall provides access control by regulating the flow of network traffic to/from a LAN
- ▶ Firewall disconnects any part of the DS from outside world
 - The internal LANs route packets through Packet Filtering Routers (PFR) and Application Gateway (AG)
 - Incoming/outgoing packet contents (e.g., src/dest IP) are examined by PFRs
 - PFRs selectively forward the incoming/outgoing packets to/from AG



Denial of Service Attacks

- ▶ Denial of service (DoS) usually refers to an attack that attempts to make a computer resource unavailable to its intended users by flooding a network or server with requests and data.
- ▶ It can also simply refer to a resource, such as e-mail or a Web site, that is not functioning as usual.



Denial of Service Attacks

- ▶ Denial of Service (DoS) attacks maliciously prevents authorized processes from accessing the resources
 - Example: Flood a search server with TCP requests
- ▶ DoS attacks from a single source can be easily prevented
- ▶ Triggering DoS attacks from multiple sources (Distributed DoS or DDoS) is hard to detect
 - Attackers hijack a large group of machines
 - DDoS is hard to detect
- ▶ Two Types of DoS Attacks

Denial of Service Attacks

1. **Bandwidth Depletion:** Send many messages to a server
 - Normal messages cannot reach the server
2. **Resource Depletion :** Trick the server to allocate large amount of resources
 - Example: TCP SYN flooding
 - Initiate a lot of TCP connections on server (by sending TCP SYN packets)
 - Do not close the connections
 - Each connection request blocks some memory for some time

Preventing DoS Attacks

1. Monitor at ingress routers:

- Monitor incoming traffic for large flow towards a destination
- Drawback: Too late to detect since regular traffic is already blocked

2. Monitor at egress routers:

- Monitor traffic when packets leave organization's network
- e.g., Drop packets whose source IP does not belong to the org's network

3. Monitor at core internet routers:

- Monitor at routers at the core of the Internet Service Providers (ISPs)
- e.g., Drop packets when rate of traffic to a specific node is disproportionately larger than the rate of traffic from that node

Key Distribution

- ▶ Two types
 - 1. Secret-key distribution
 - 2. Public-key distribution

Securing Mobile Code

- ▶ Two main issues:
 - Securing against malicious agents that attempt to damage a mobile agent environment.
 - Securing a mobile agent from a malicious environment that attempts to interfere with the working of the mobile agent.
- ▶ It has proved impossible to protect an agent against all types of attack.
- ▶ The emphasis is on being able to detect modifications (or tampering) to an agent.
- ▶ The Ajanta System uses public-key technologies to implement this idea:
 - Read-only state.
 - Append-only logs.
 - Selective Revealing

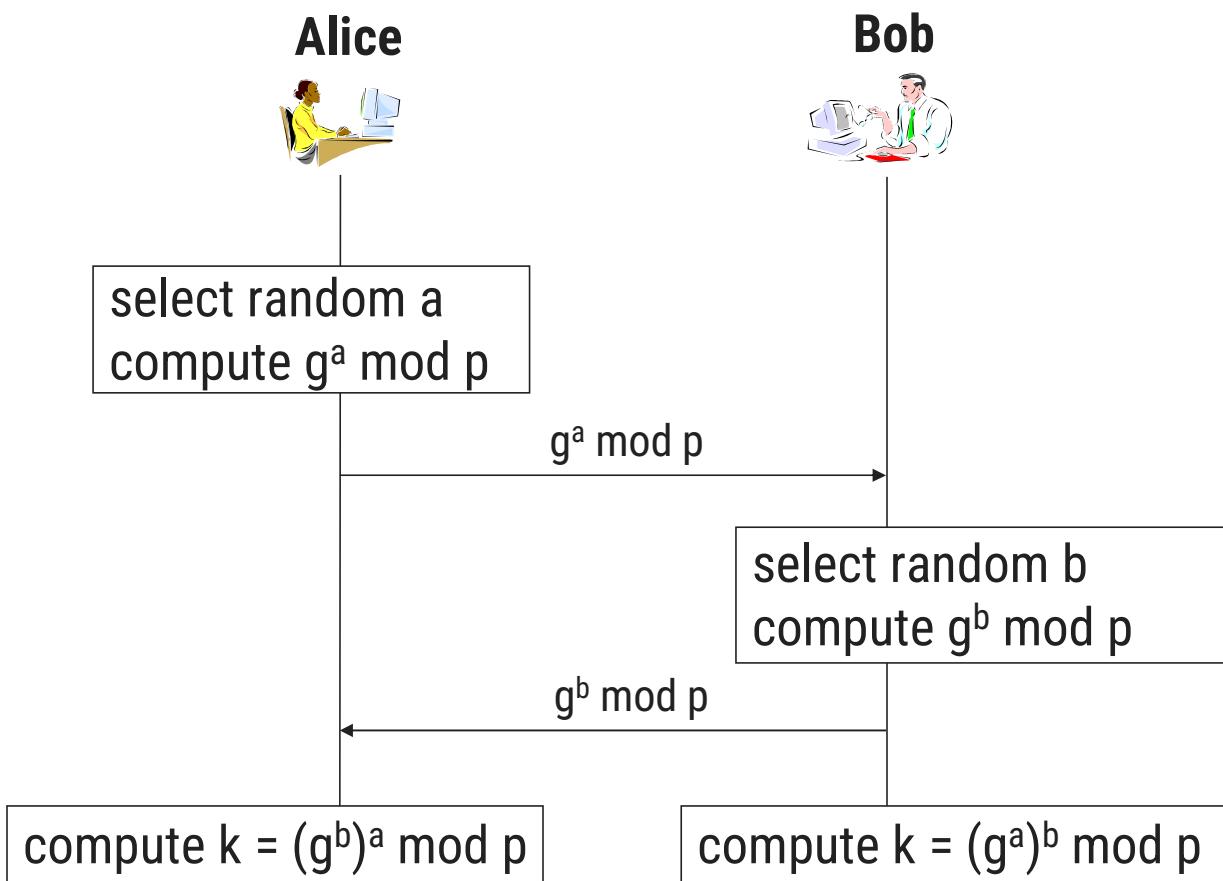
Security Management

- ▶ Security is implemented using symmetric and asymmetric keys.
- ▶ There are three main issues:
 1. How to manage cryptographic keys.
 2. How to manage securely a group of servers such that a malicious process is not added to the group.
 3. Authorization management by capabilities and attribute certificates.

Key Management

- ▶ Examples of key-generation that we studied:
 - Session key : Generated and distributed by public-private key pair
 - Public-Private Key : Generated by algorithms. How is it securely distributed?
- ▶ How to boot-strap secure key generation and distribution in a DS?
- ▶ There are two main approaches in key management:
 - Diffie-Hellman Algorithm (For key generation over insecure channels)
 - Key distribution

Diffie-Hellman Algorithm for Key Establishment



Diffie-Hellman Algorithm for Key Establishment

- ▶ Suppose Alice and Bob want to agree on a shared symmetric key.
- ▶ Alice and Bob, and everyone else, already know the values of p and g .
- ▶ Alice generates a random **private value a** and Bob generates a random **private value b**.
- ▶ Both a and b are drawn from the set of integers $\{1, \dots, p-1\}$.
- ▶ Alice and Bob derive their corresponding public values—the values they will send to each other unencrypted—as follows.
- ▶ Alice's public value is $g^a \text{ mod } p$ and Bob's public value is $g^b \text{ mod } p$
- ▶ They then exchange their public values.
- ▶ Finally, Alice computes $g^{ab} \text{ mod } p = (g^b \text{ mod } p)^a \text{ mod } p$
- ▶ Bob computes $g^{ba} \text{ mod } p = (g^a \text{ mod } p)^b \text{ mod } p$.

Diffie-Hellman Algorithm- An example

► Step 1 –Publicly shared information

- Alice & Bob publicly agree to a number called the generator, or g , which has a primitive root relationship with p .
- In our example we'll assume
- $p = 17$
- $g = 3$

► Step 2 – Select a secret key

- Alice selects a secret key, which we will call a .
- Bob selects a secret key, which we will call b .
- For our example assume: $a = 54$ and $b = 24$

Diffie-Hellman Algorithm- An example

► Step 3 – Combine secret keys with public information

- Alice combines her secret key of a with the public information to compute A .
 - $A = g^a \text{ mod } p$
 - $A = 354 \text{ mod } 17$
 - $A = 15$
- Bob combines his secret key of b with the public information to compute B .
 - $B = g^b \text{ mod } p$
 - $B = 354 \text{ mod } 17$
 - $B = 16$

► Step 4 – Share combined values

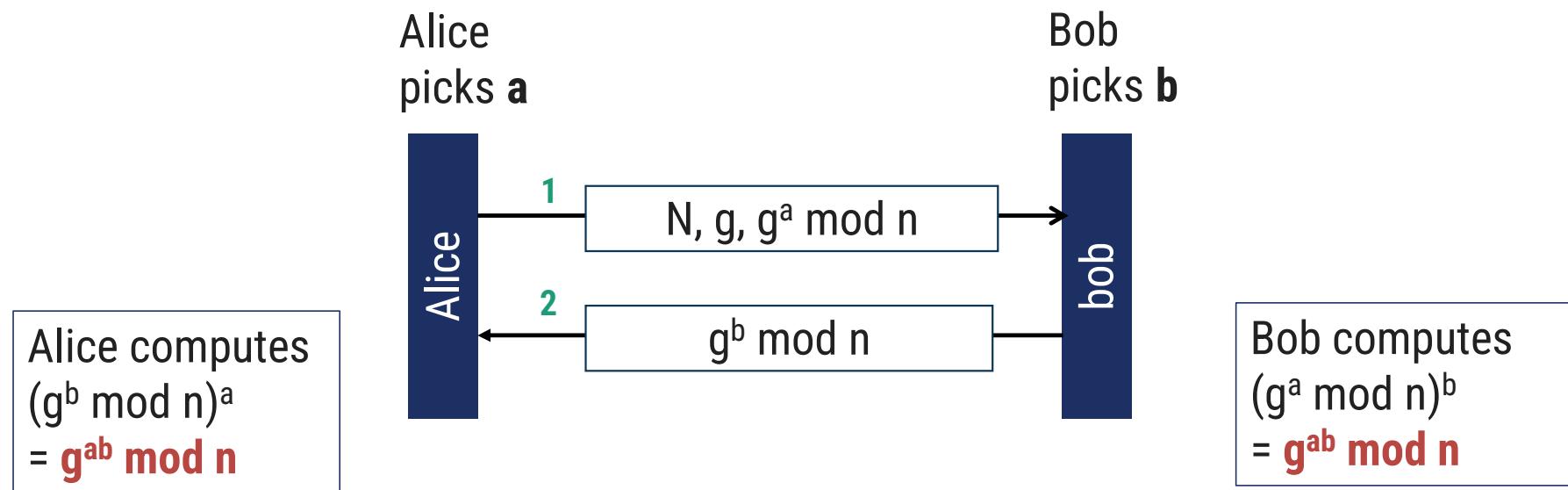
- Alice shares her combined value, A , with Bob. Bob shares his combined value, B , with Alice.
- Sent to Bob => $A = 15$
- Sent to Alice => $B = 16$

Diffie-Hellman Algorithm- An example

► Step 5 – Compute Shared Key

- Alice computes the shared key.
 - $s = (B \text{ mod } p)^a \text{ mod } p$
 - $s = g^{b*a} \text{ mod } p$
 - $s = 3^{54*24} \text{ mod } 17$
 - **s = 1**
- Bob computes the shared key.
 - $s = (A \text{ mod } p)^a \text{ mod } p$
 - $s = g^{a*b} \text{ mod } p$
 - $s = 3^{24*54} \text{ mod } 17$
 - **s = 1**

Diffie-Hellman Algorithm for Key Establishment



Unit-7

Categories of Distributed System



Prof. Umesh H. Thoraya
Computer Engineering Department
Darshan Institute of Engineering & Technology, Rajkot

 umesh.thoriya@darshan.ac.in
 9714233355



Topics to be covered

- Distributed Object-based System
- Distributed File System
- Distributed Web-based System
- Distributed Coordination based System



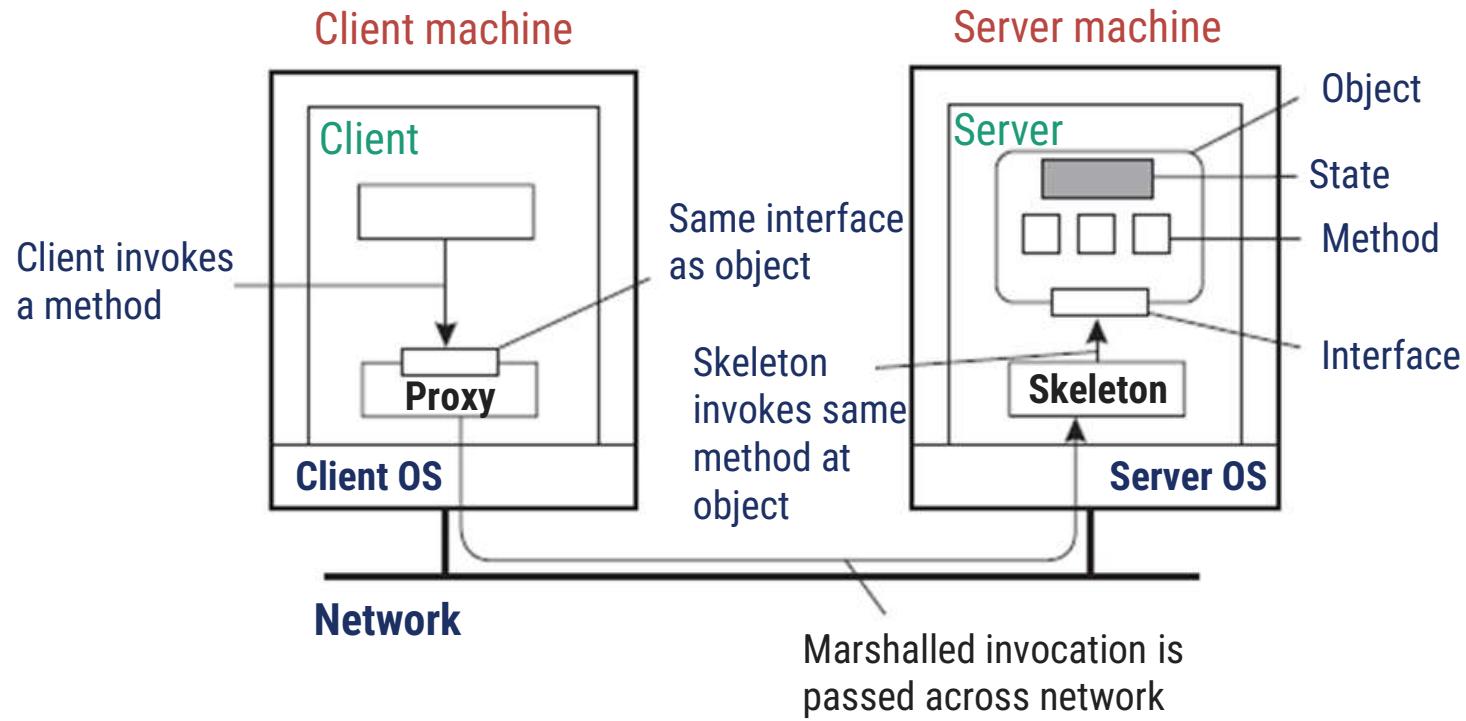
Categories of Distributed System

Distributed Object-based System

Distributed Object-based System

- ▶ In distributed object-based systems, the notion of an object plays a key role in establishing distribution transparency.
- ▶ In principle, everything is treated as an object and clients are offered services and resources in the form of objects that they can invoke.
- ▶ Few examples of Object based systems are CORBA, Java-based systems, and Globe

Distributed Objects



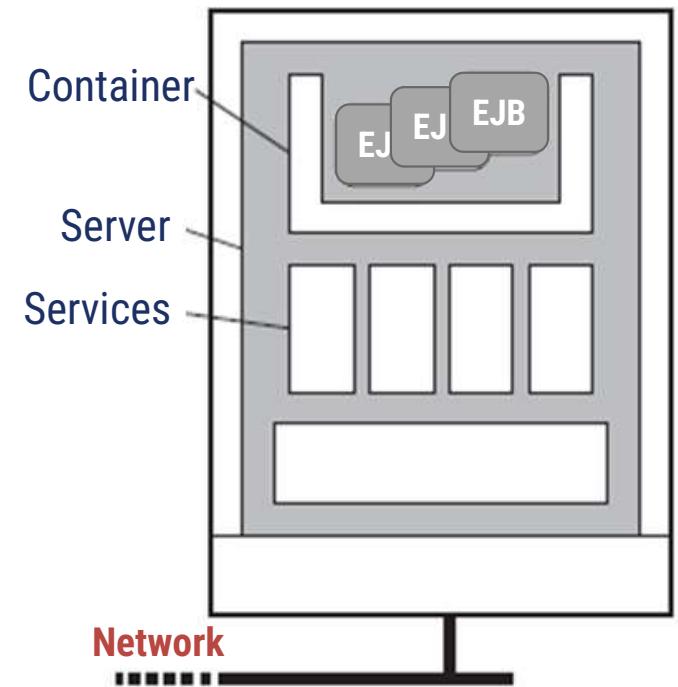
Common organization of a remote object with client-side proxy.

Distributed Objects

- ▶ Data and operations **encapsulated** in an object
- ▶ Operations implemented as **methods** grouped into **interfaces**
- ▶ Object offers only its **interface** to clients
- ▶ **Object server** is responsible for a collection of objects
- ▶ Client stub (**proxy**) implements interface
- ▶ Server skeleton handles (un)marshaling and object invocation

Example: Enterprise Java Beans

- ▶ An EJB is essentially a **Java object** that is hosted by a special server offering different ways for remote clients to invoke that object
- ▶ The important issue is that an EJB is **embedded inside a container** which effectively provides interfaces to underlying services that are implemented by the application server.
- ▶ The container can more or less automatically bind the EJB to these services, meaning that the correct references are readily available to a programmer
- ▶ Typical services include those for remote method invocation (RMI), database access (JDBC), naming (JNDI), and messaging (JMS).



General architecture of an Enterprise Java Beans

Types of EJBs(Enterprise Java Beans)

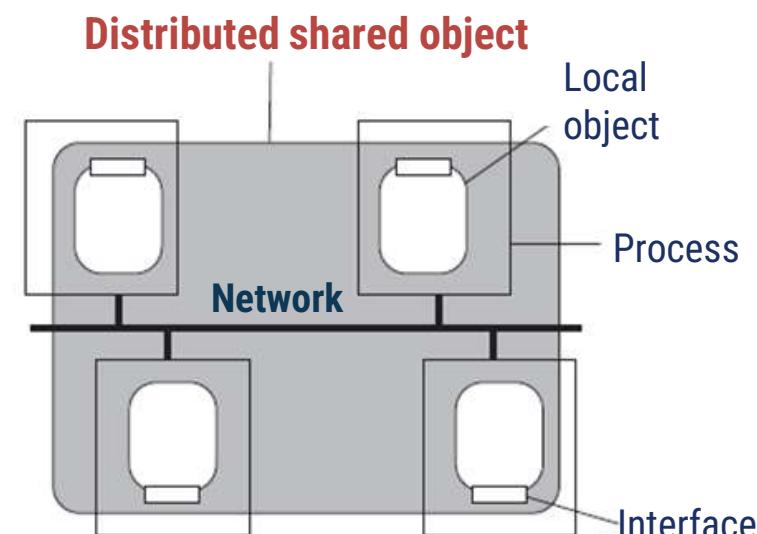
1. Stateless session beans
2. Stateful session beans
3. Entity beans
4. Message-driven beans

Types of EJBs(Enterprise Java Beans)

- ▶ **Stateless session bean:** Transient object, called once, does its work and is done.
 - Example: execute an SQL query and return result to caller.
- ▶ **Stateful session bean:** Transient object, but maintains client-related state until the end of a session.
 - Example: shopping cart.
- ▶ **Entity bean:** Persistent, state-ful object, can be invoked during different sessions.
 - Example: object maintaining client info on last number of sessions.
- ▶ **Message-driven bean:** Reactive objects, often triggered by message types.
 - Used to implement publish/subscribe forms of communication.

Globe Distributed Shared Objects

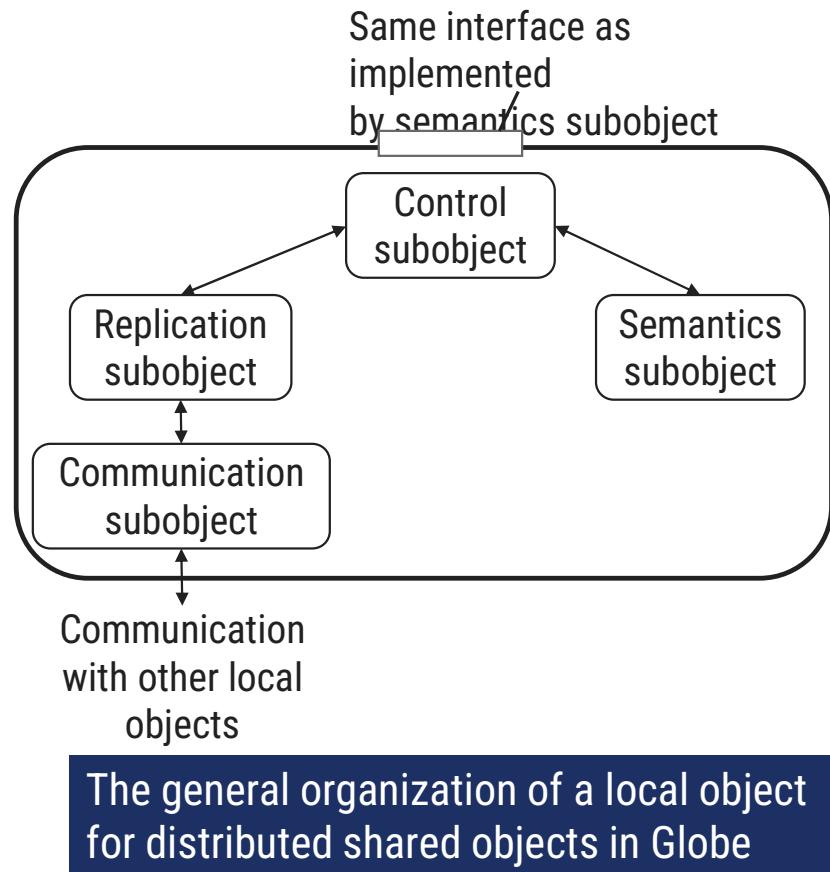
- ▶ A Globe object is a physically distributed shared object: the object's state may be physically distributed across several machines
- ▶ **Local object:** A non-distributed object residing a single address space, often representing a distributed shared object
- ▶ **Contact point:** A point where clients can contact the distributed object; each contact point is described through a contact address



The organization of a Globe distributed shared object

Globe Distributed Shared Objects

- ▶ Globe attempts to separate functionality from distribution by distinguishing different local sub-objects
- ▶ **Semantics sub-object:** Contains the methods that implement the functionality of the distributed shared object
- ▶ **Communication sub-object:** Provides a (relatively simple), network-independent interface for communication between local objects
- ▶ **Replication sub-object:** Contains the implementation of an object-specific consistency protocol that controls exactly when a method on the semantics sub-object may be invoked
- ▶ **Control sub-object:** Connects the user-defined interfaces of the semantics sub-object to the generic, predefined interfaces of the replication sub-object

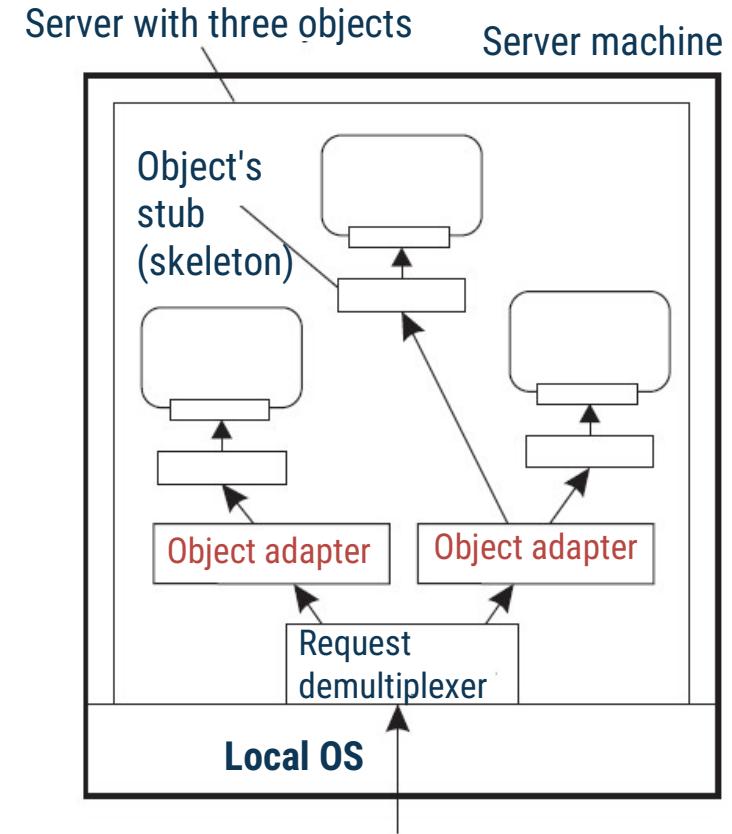


Processes: Object servers

- ▶ Servant The actual implementation of an object, sometimes containing only method implementations:
- ▶ Skeleton: Server-side stub for handling network I/O:
 - Un-marshalls incoming requests, and calls the appropriate servant code
 - Marshalls results and sends reply message
 - Generated from interface specifications
- ▶ The “**manager**” of a set of objects:
 - Inspects (as first) incoming requests
 - Ensures referenced object is activated (requires identification of servant)
 - Passes request to appropriate skeleton, following specific activation policy
 - Responsible for generating object references

Object Adapter

- ▶ The role of the object adapter is to bridge the gap between CORBA objects with IDL interfaces and the programming language interfaces of the corresponding servant classes.
- ▶ An object adapter has the following tasks
 - It creates remote object references for CORBA objects
 - It dispatches each RMI via skeleton to the appropriate servant
 - It activates and deactivates servants



Organization of an object server supporting different activation policies

Communication: Client-to-object binding

- ▶ Object reference: Having an object reference allows a client to bind to an object
 - Reference denotes server, object, and communication protocol
 - Client loads associated stub code
 - Stub is instantiated and initialized for specific object
- ▶ Two ways of binding
 1. **Implicit**: Invoke methods directly on the referenced object
 2. **Explicit**: Client must first explicitly bind to object before invoking it

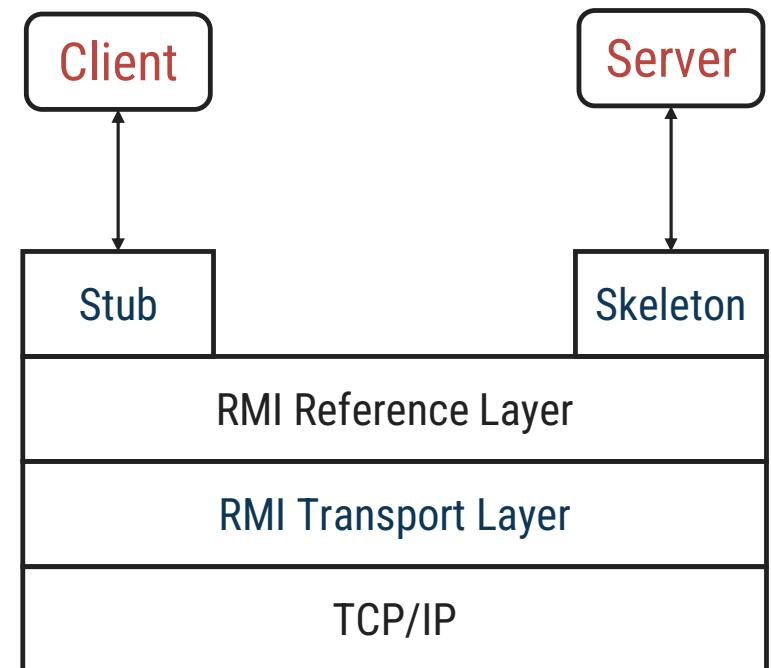
Java Remote Method Invocation (RMI)

- ▶ The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM.
- ▶ Java RMI provides applications with transparent and lightweight access to remote objects. RMI defines a high-level protocol and API.
- ▶ Programming distributed applications in Java RMI is simple.
 - It is a single-language system.
 - The programmer of a remote object must consider its behavior in a concurrent environment.

Remote Method Invocation (RMI)

RMI Architecture

- ▶ **Stub:** lives on the client; pretends to be the remote object
- ▶ **Skeleton:** lives on the server; talks to the true remote object
- ▶ **Reference Layer:** determines if referenced object is local or remote
- ▶ **Transport Layer:** packages remote invocations; dispatches messages between stub and skeleton



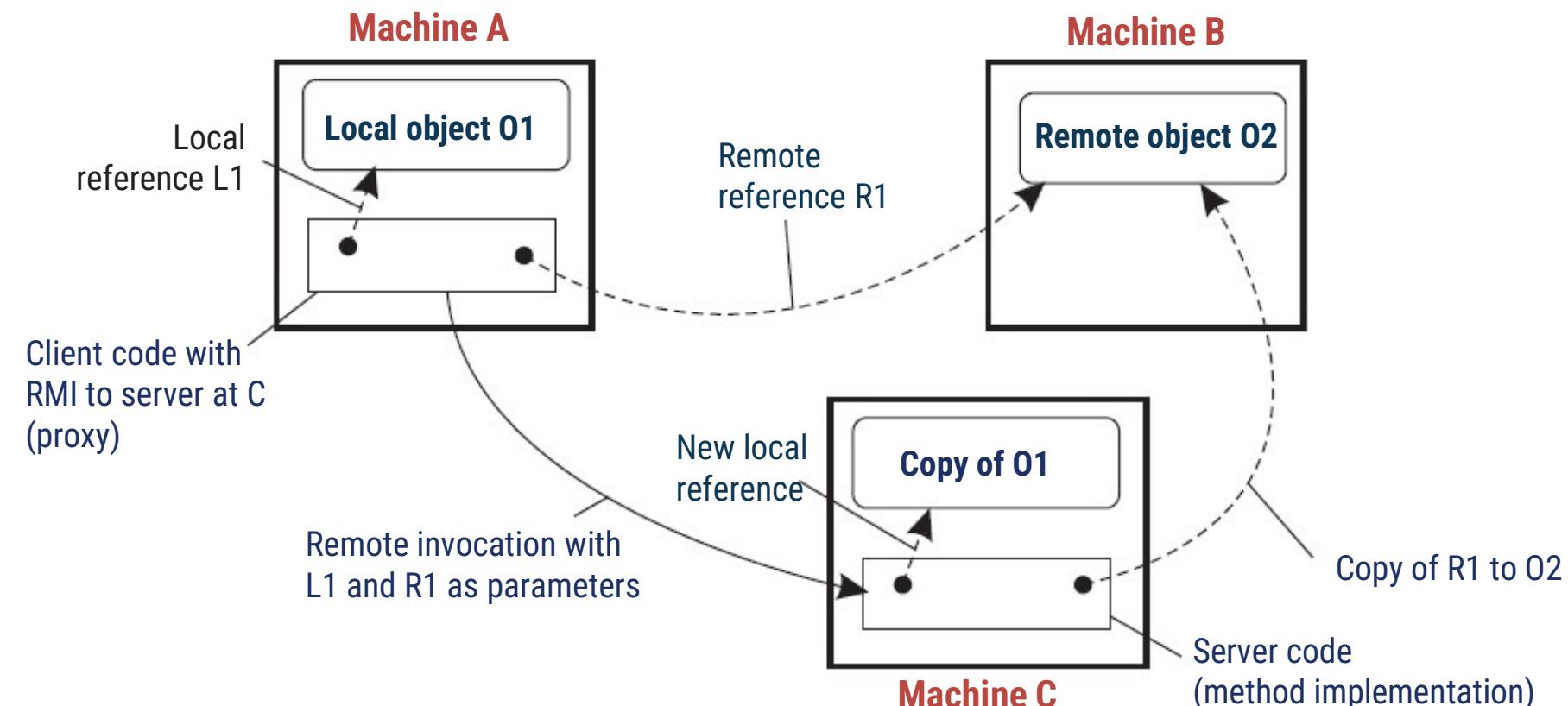
Remote Method Invocation (RMI)

- ▶ Client invokes method at stub
- ▶ Stub marshals request and sends it to server
- ▶ Server ensures referenced object is active:
 - Create separate process to hold object
 - Load the object into server process
- ▶ Request is unmarshaled by object's skeleton, and referenced method is invoked
- ▶ If request contained an object reference, invocation is applied recursively (i.e., server acts as client)
- ▶ Result is marshaled and passed back to client
- ▶ Client stub unmarshals reply and passes result to client application

RMI: Parameter passing

- ▶ Object reference: Much easier than in the case of RPC
 - Server can simply bind to referenced object, and invoke methods
 - Unbind when referenced object is no longer needed
- ▶ Object-by-value: A client may also pass a complete object as parameter value
 - An object has to be marshaled:
 - ✓ Marshall its state
 - ✓ Marshall its methods, or give a reference to where an implementation can be found
 - Server un-marshals object. Note that we have now created a copy of the original object.
 - Object-by-value passing tends to introduce nasty problems

RMI: Parameter passing



The situation when passing an object by reference or by value

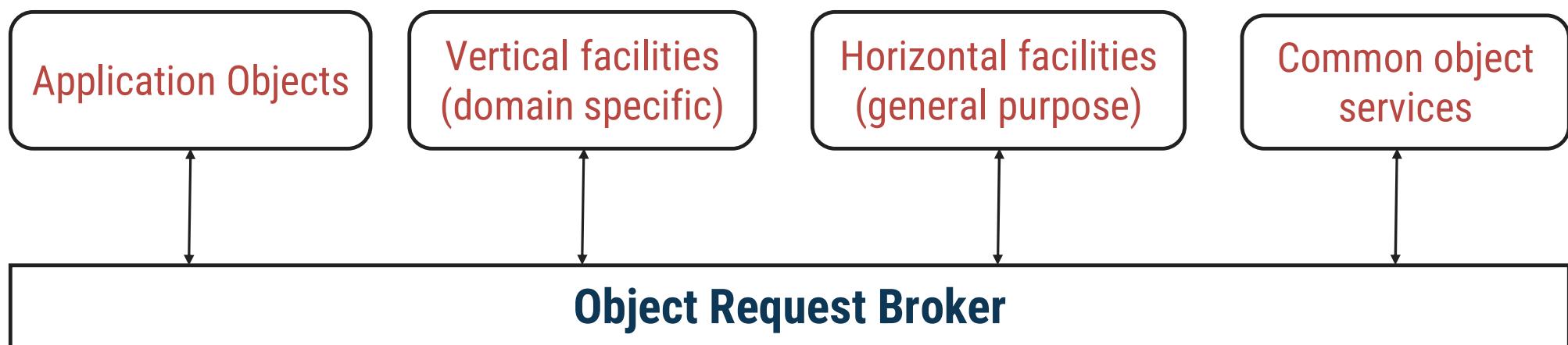
CORBA: Overview

- ▶ **OMG** : The Object Management Group (OMG) was formed in 1989 and now has over 800 members.
- ▶ Its aims were:
 - To make better use of distributed systems
 - To use object-oriented programming
 - To allow objects in different programming languages to communicate with one another
 - The object request broker (ORB) enables clients to invoke methods in a remote object
 - CORBA (Common Object Request Broker Architecture) is a specification of an architecture supporting this (rather than an implementation such as RMI).

CORBA: Overview

- ▶ CORBA (Common Object Request Broker Architecture) is a specification of an architecture supporting this (rather than an implementation such as RMI).
- ▶ **ORB(Object Request Broker)** : the runtime system responsible for communication between objects and their clients
- ▶ CORBA facilities: collections of classes and objects that provide general-purpose capabilities that are useful in many applications.
 - Horizontal facilities: user interfaces, information management, system management, task management
 - Vertical facilities: e.g., electronic commerce, banking, manufacturing, healthcare, telecommunications

CORBA: Overview

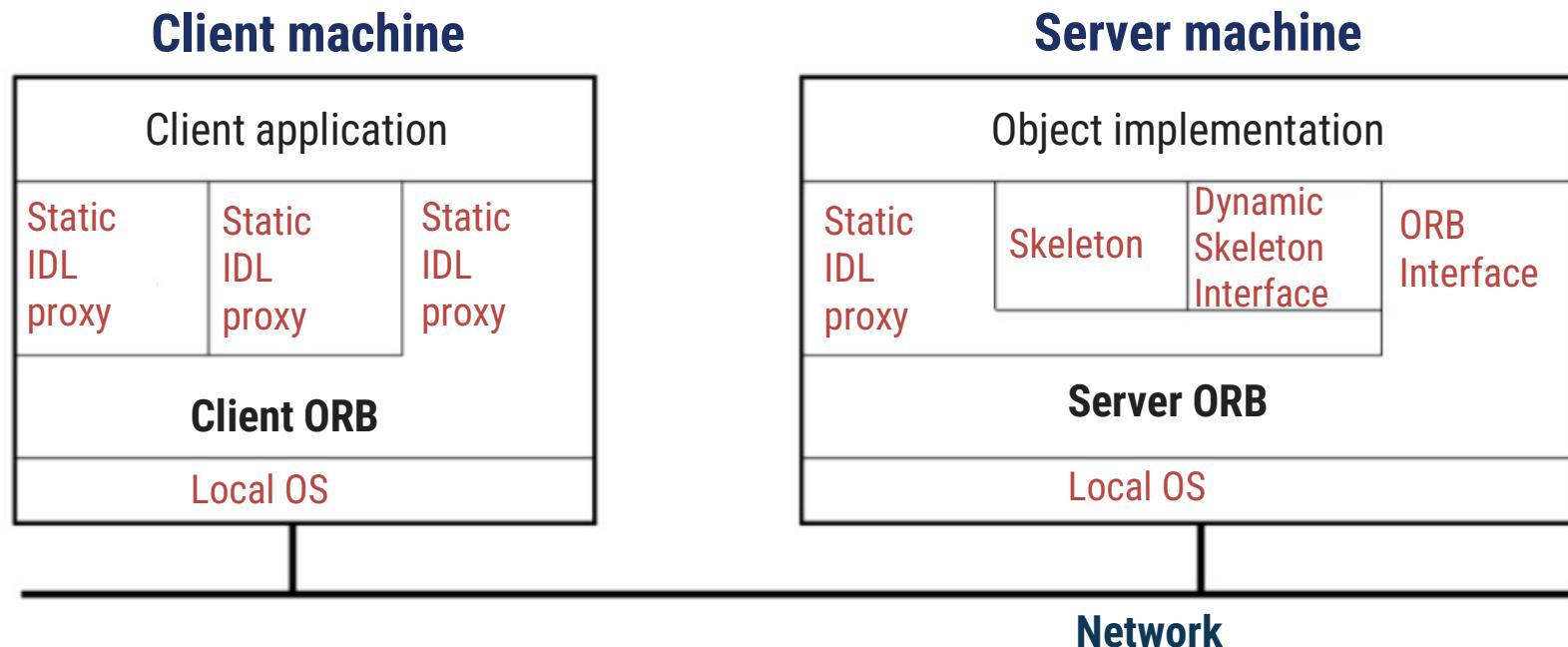


- ▶ **ORB**: the runtime system responsible for communication between objects and their clients
- ▶ **CORBA facilities**: collections of classes and objects that provide general-purpose capabilities that are useful in many applications.
 - **Horizontal facilities**: user interfaces, information management, system management, task management
 - **Vertical facilities**: e.g., electronic commerce, banking, manufacturing, healthcare, telecommunications

CORBA: Components (Main CORBA Components)

- ▶ An Interface Definition Language (IDL) and its mapping onto the implementation language (C, C++, Java, Smalltalk, Ada, COBOL)
 - specifies the syntax of objects and services
 - cannot be used to describe semantics
 - object can be implemented by a language without classes
- ▶ An Interface Repository (IR) representing the interfaces of all objects available in the distributed system.
- ▶ A fully dynamic calling mechanism allowing:
 - run-time discovery of objects
 - discovery of available interfaces from an IR
 - construction of message requests
 - sending of message requests
- ▶ Object Adapters: an abstraction mechanism for removing implementation details from the message substrate.

CORBA: Organisation



ORB offers:

- Basic communication
- Object references
- Initial finding of services

Interfaces:

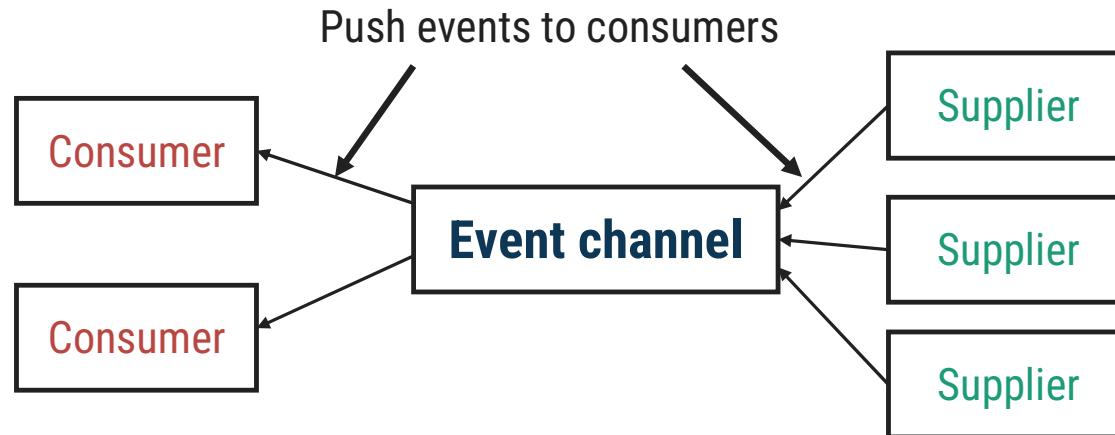
- Static (described in IDL)
- Run-time creation (Dynamic Invocation Interface)

CORBA: Communication

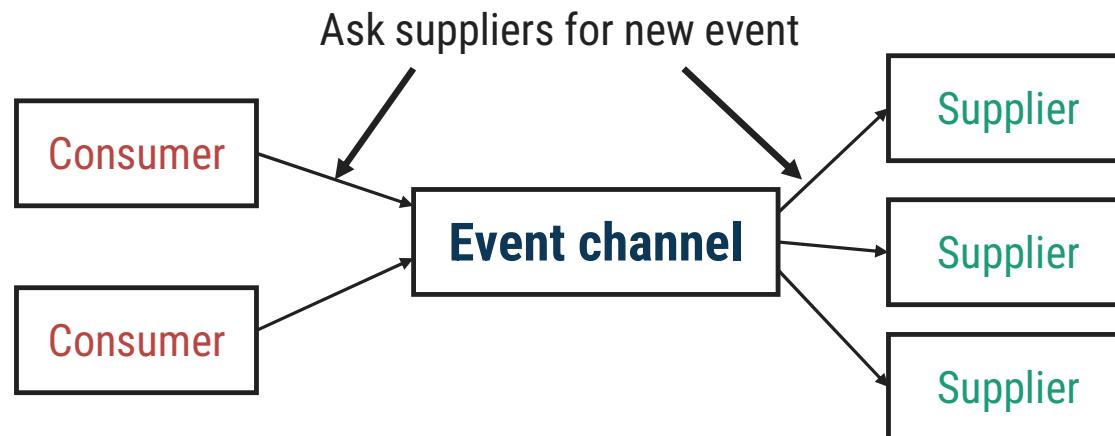
Request type	Failure semantics	Description
Synchronous	At-most-once	Caller blocks until a response is returned or an exception is raised
One-way	Best effort delivery	Caller continues immediately without waiting for any response from the server
Deferred Synchronous	At-most-once	Caller continues immediately and can later block until response is delivered

CORBA: Event and Notification Services

Push-style model



Pull-style model

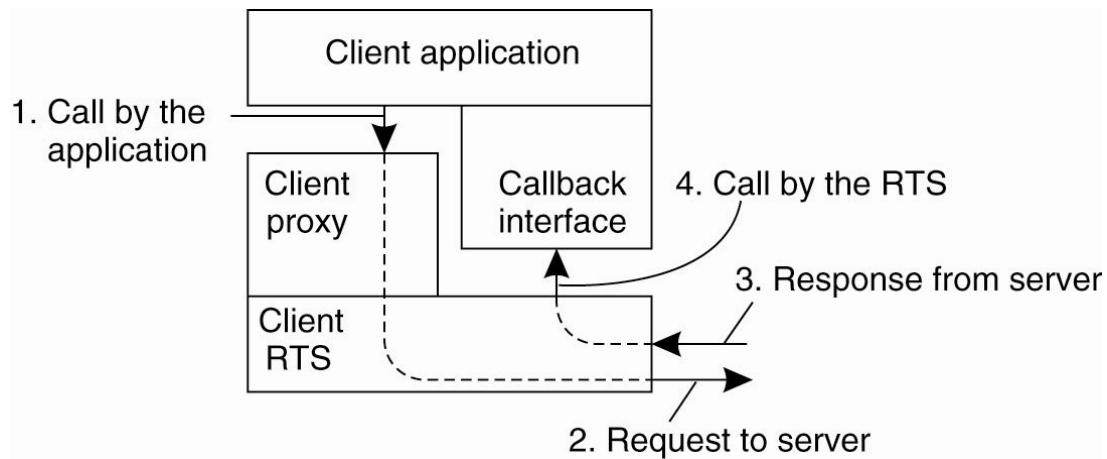


CORBA: Persistent Communication

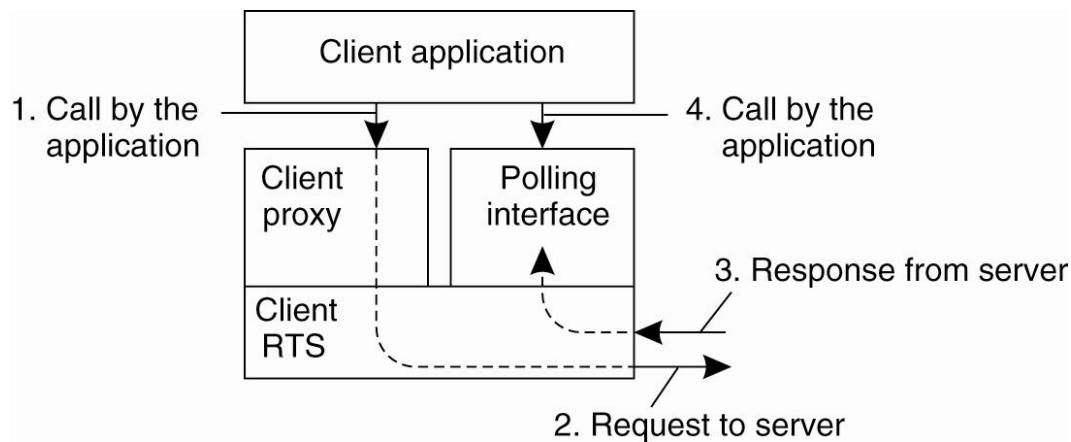
► CORBA messaging service:

- Object-oriented approach to communication
- All communication takes place by invoking an object
- May not be possible to get an immediate reply
- Callback model: A client features two interfaces: method and callback
- Polling model: A client is offered a collection of operations to poll its ORB for incoming results
- Messages sent are stored by the underlying system in case the client or server is not yet running

CORBA: Messaging



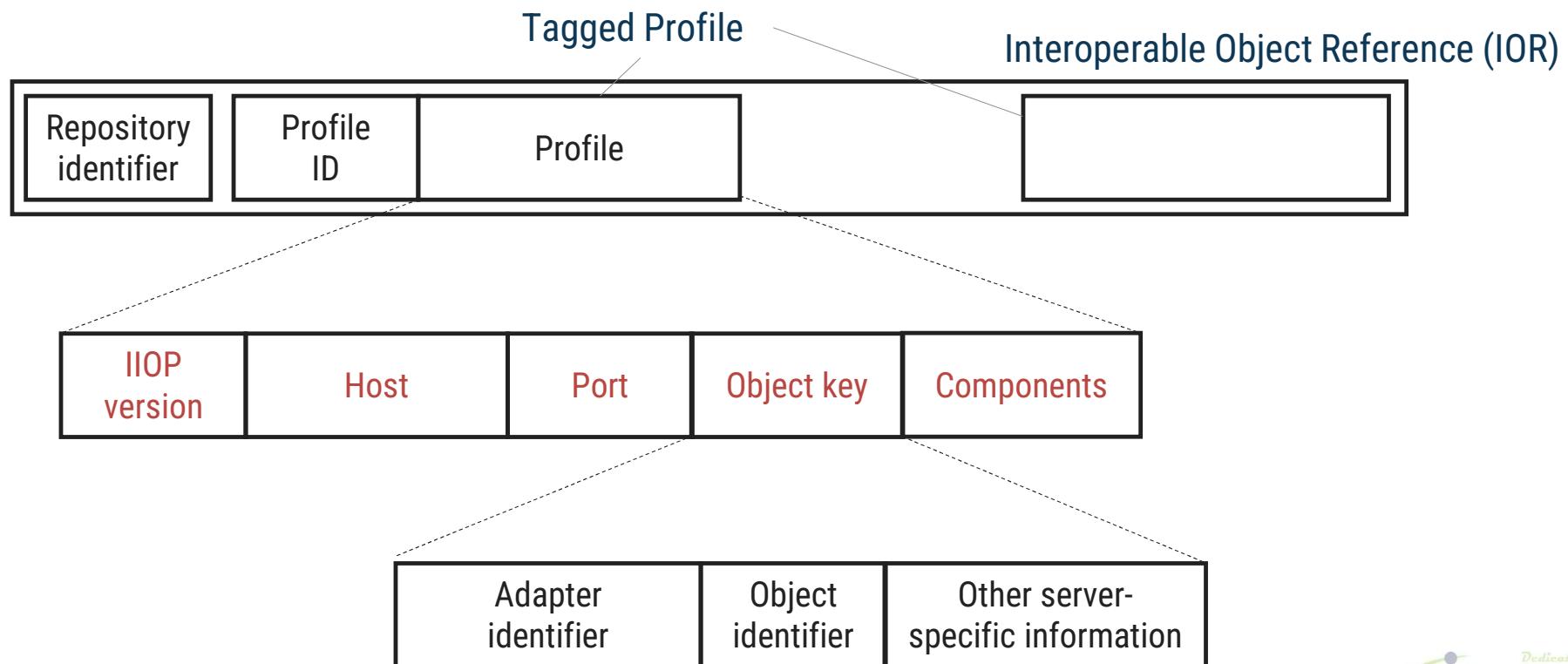
CORBA's callback model for asynchronous method invocation



CORBA's polling model for asynchronous method invocation

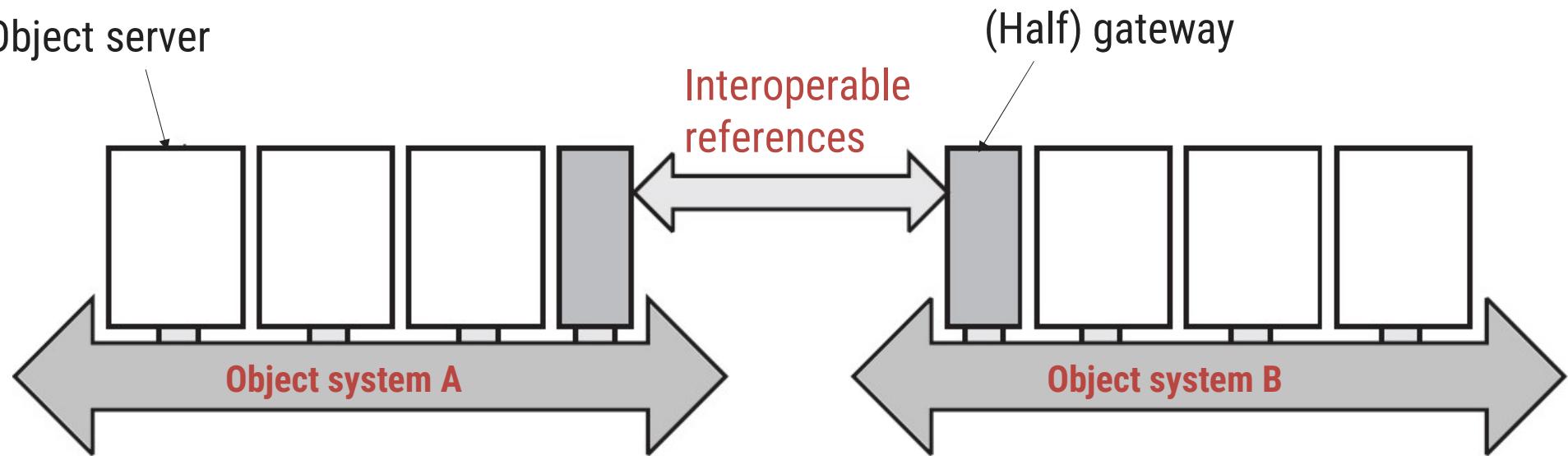
CORBA: Object references

- In order to invoke remote objects, we need a means to uniquely refer to them.



CORBA: Object references

- It is not important how object references are implemented per object-based system, as long as there is a standard to exchange them between systems.



CORBA: Interoperability

- ▶ Interoperability is achieved by the introduction of a standard inter-ORB protocol (**General Inter-ORB Protocol GIOP**) which:
 - Defines an external data representation, called the **Common Data Representation (CDR)**
 - Specifies formats for the messages in a request-reply protocol including messages for enquiring about the location of an object, for canceling requests and for reporting errors.
- ▶ The **Internet Inter-ORB protocol (IIOP)** defines a standard form for remote object references.
 - IIOP is GIOP implemented in TCP/IP

CORBA: Clients and Servers

▶ Clients:

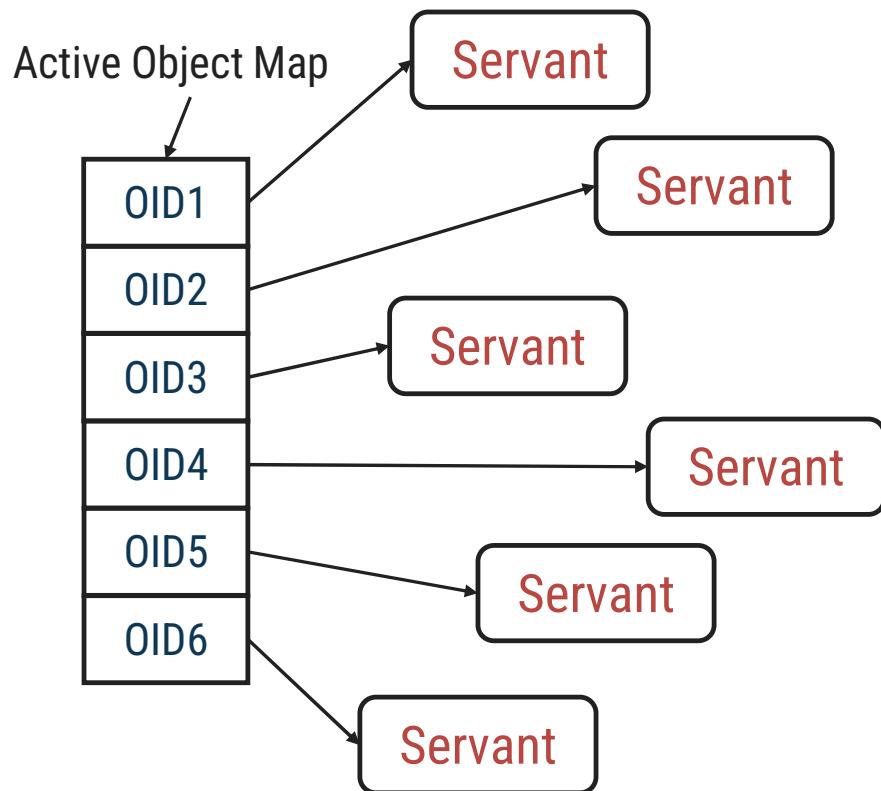
- The IDL specification is compiled into a **proxy**
- Proxy connects a client application to the underlying ORB

▶ Servers:

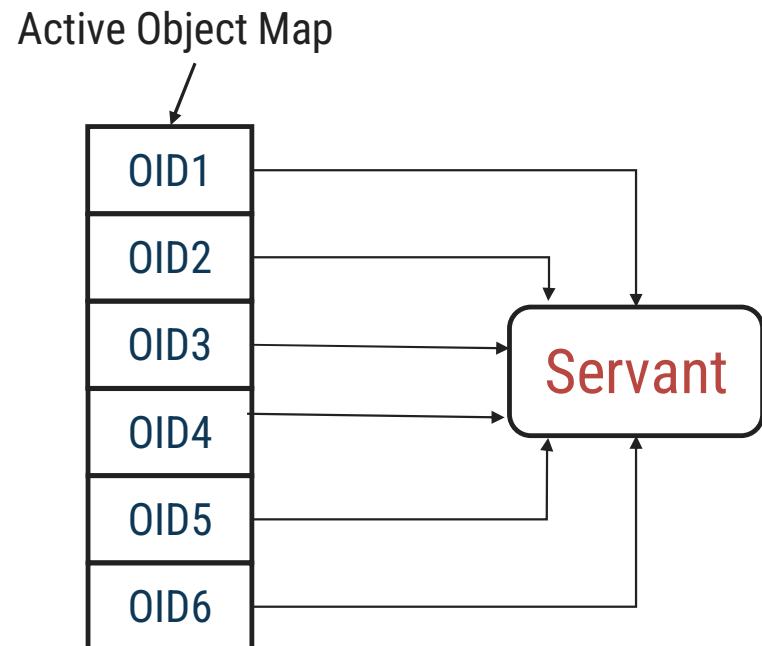
- The **Portable Object Adapter (POA)** makes code appear as CORBA objects to clients
- Creates remote object references for CORBA objects
- Activates objects
- Implements a specific **activation policy** (e.g., thread-per-request, threadper-connection, thread-per-object policy,...)
- Methods are implemented by means of **servants**.

CORBA: Clients and Servers

- ▶ Mapping of CORBA object identifiers to servants.



The POA supports multiple servants

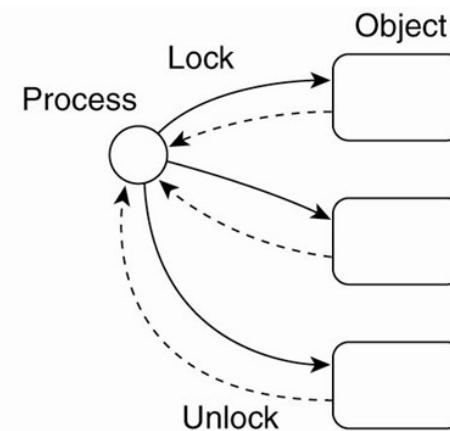
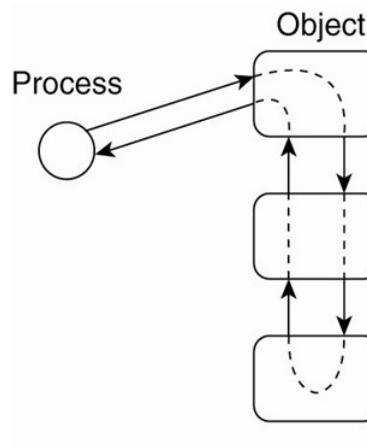


The POA supports a single servant

Synchronization and Locking

► Problem: Implementation details are hidden behind interfaces

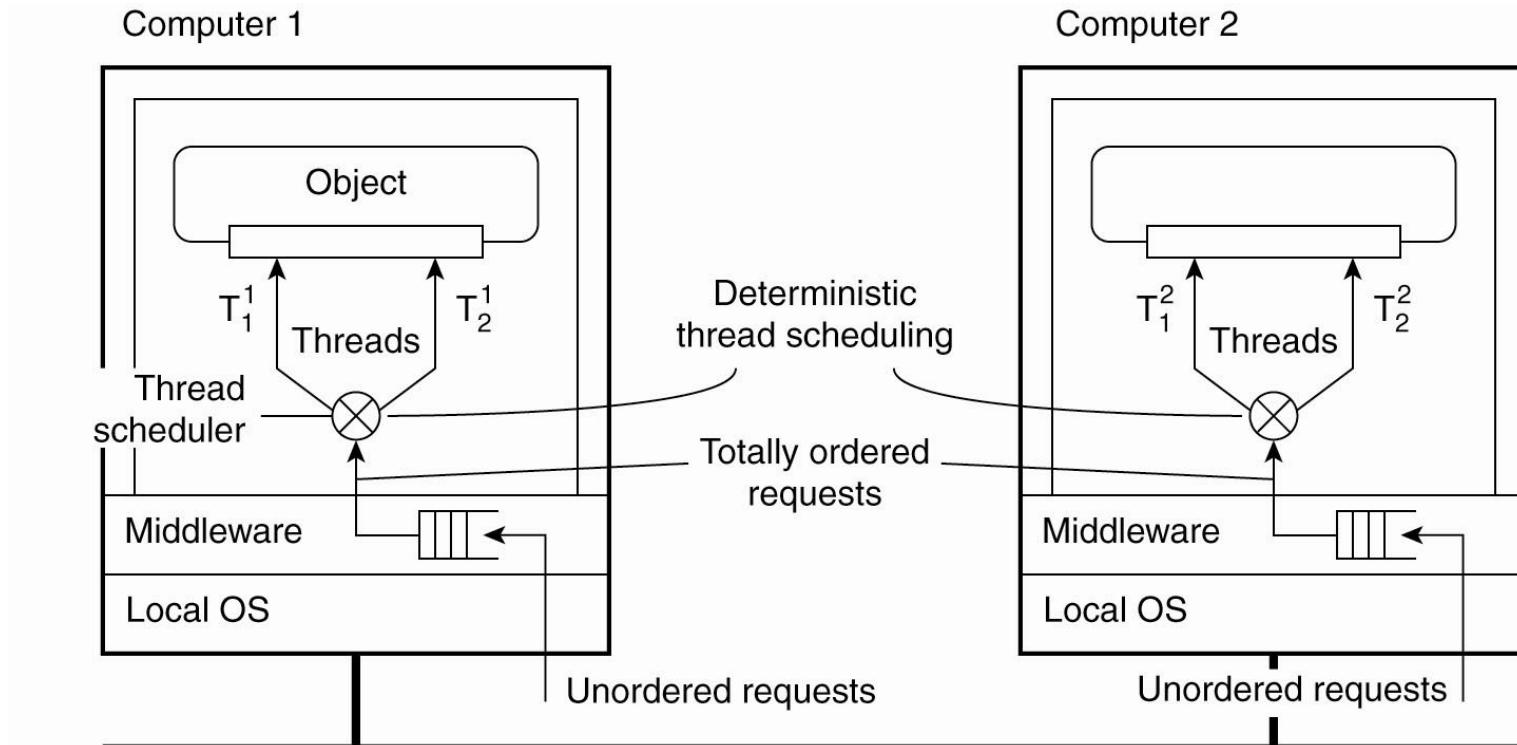
- When a process invokes a remote object, it has no knowledge whether that invocation will lead to invoking other objects.
 - ✓ If an object is protected against concurrent access, we may have cascading set of locks the invoking process is unaware of
- When dealing with data resources such as files or database tables that are protected by locks, the pattern for the control flow is visible to the process.



Consistency and Replication

- ▶ General approach to replicated objects:
 - Traditional approaches treating them as containers of data with their own operations
- ▶ Consistency: Data-centric consistency comes in the form of entry consistency
 1. No multiple invocations on the same object allowed
 2. Ensure that all the changes to a replicated object are the same.

Entry Consistency

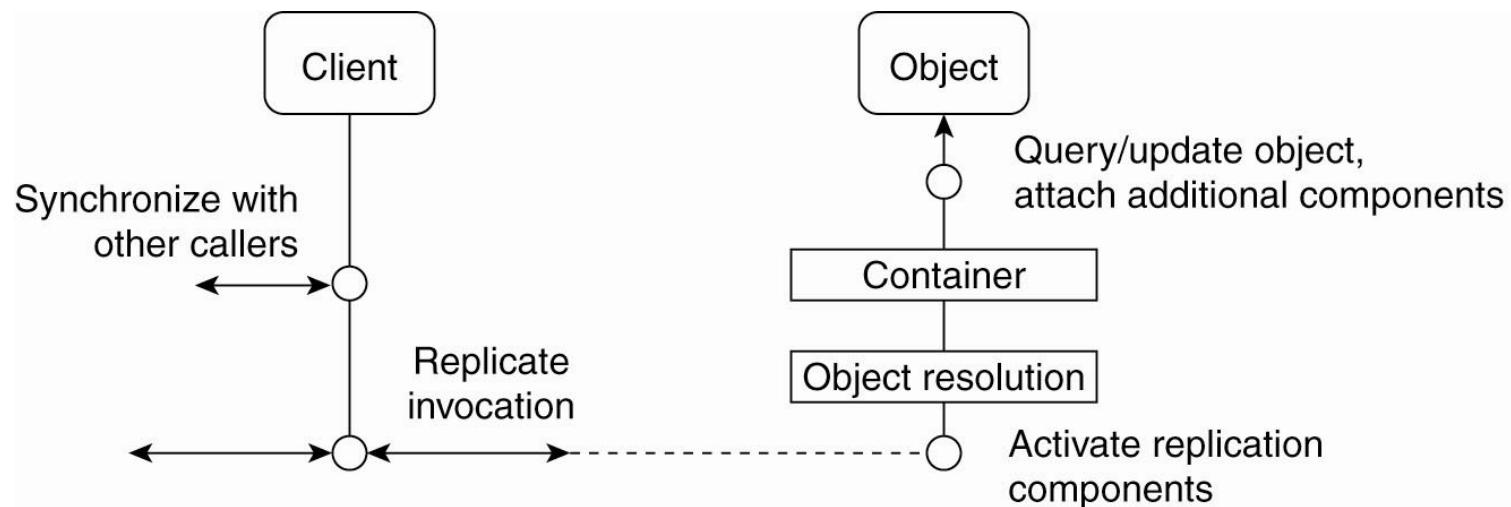


Deterministic thread scheduling for replicated object servers

Replication Frameworks

- ▶ Invocations to objects are intercepted at three different points:
 - At the client side just before the invocation is passed to the stub.
 - Inside the client's stub, where the interception forms part of the replication algorithm.
 - At the server side, just before the object is about to be invoked.

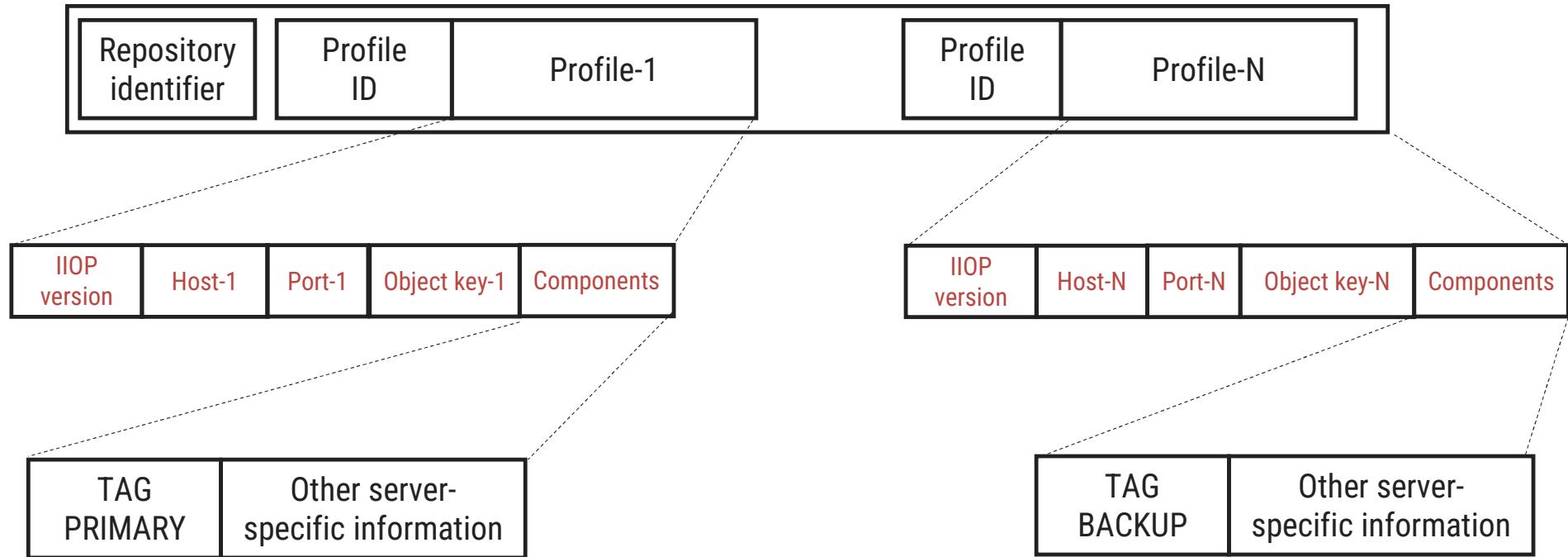
Replication Frameworks



A general framework for separating replication algorithms from objects in an EJB environment

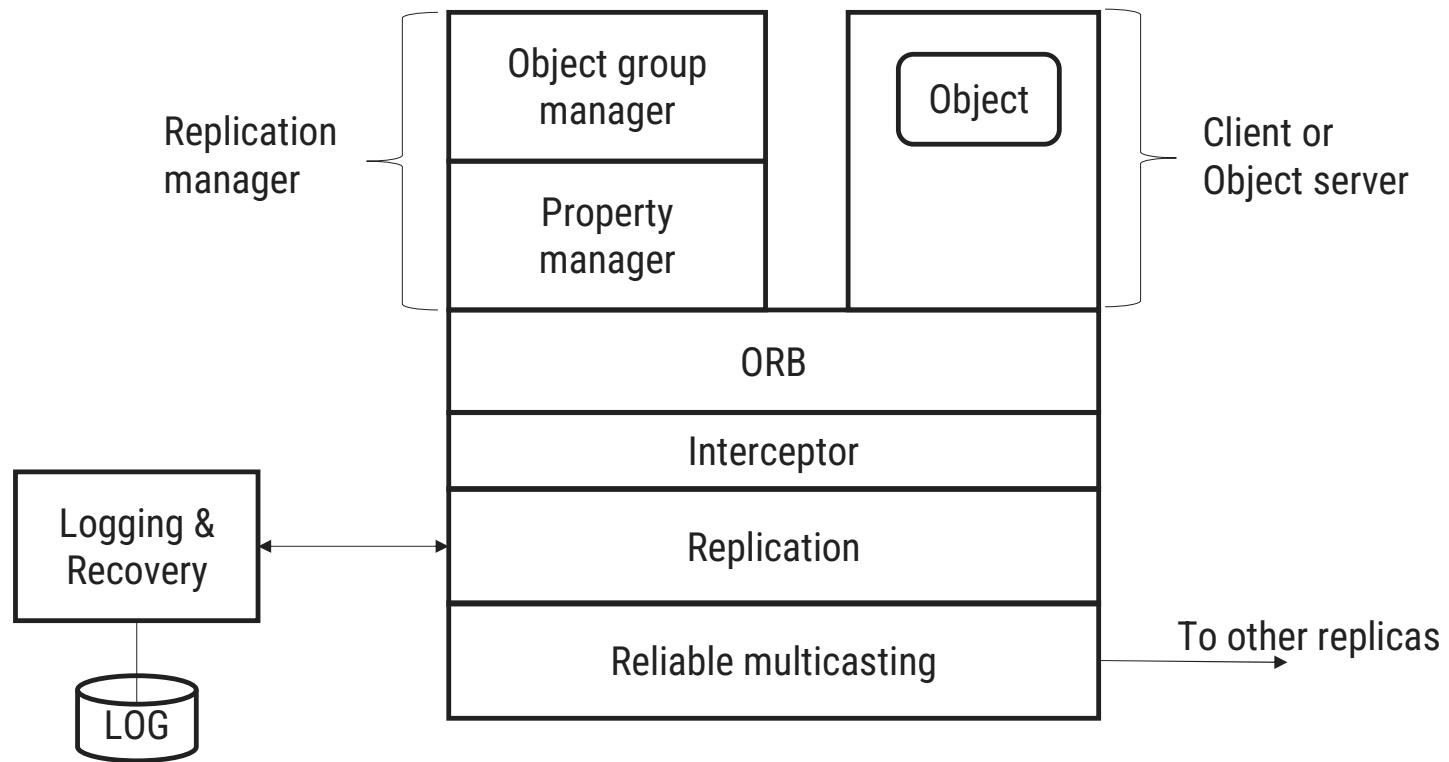
Fault-Tolerant CORBA

Interoperable Object Group Reference (IOGR)



A possible organization of an IOGR for an object group having a primary and backups

CORBA- An Example Architecture



An example architecture of a fault-tolerant CORBA system.

CORBA- An Example Architecture

- ▶ There are several components that play an important role in this architecture.
- ▶ By far the most important one is the replication manager, which is responsible for creating and managing a group of replicated objects.
- ▶ A client there is no fundamental difference between an object group and any other type of CORBA object.
- ▶ To create an object group, a client simply invokes the normal `create_object` operation as offered, in this case, by the replication manager, specifying the type of object to create. The client remains unaware of the fact that it is implicitly creating an object group.
- ▶ The number of replicas that are created when starting a new object group is normally determined by the system-dependent default value.
- ▶ The replica manager is also responsible for replacing a replica in the case of a failure, thereby ensuring that the number of replicas does not drop below a specified minimum.
- ▶ The architecture also shows the use of message-level interceptors.

Example: Fault-Tolerant Java

► Causes for nondeterministic behavior:

- JVM can execute native code, that is, code that is external to the JVM and provided to the latter through an interface.
- Input data may be subject to non-determinism.
- In the presence of failures, different JVMs will produce different output revealing that the machines have been replicated.

Overview of Globe Security

User certificate

K^+_{Alice}
$U: 0010011100$
$\text{sig}(O, \{U, K^+_{Alice}\})$

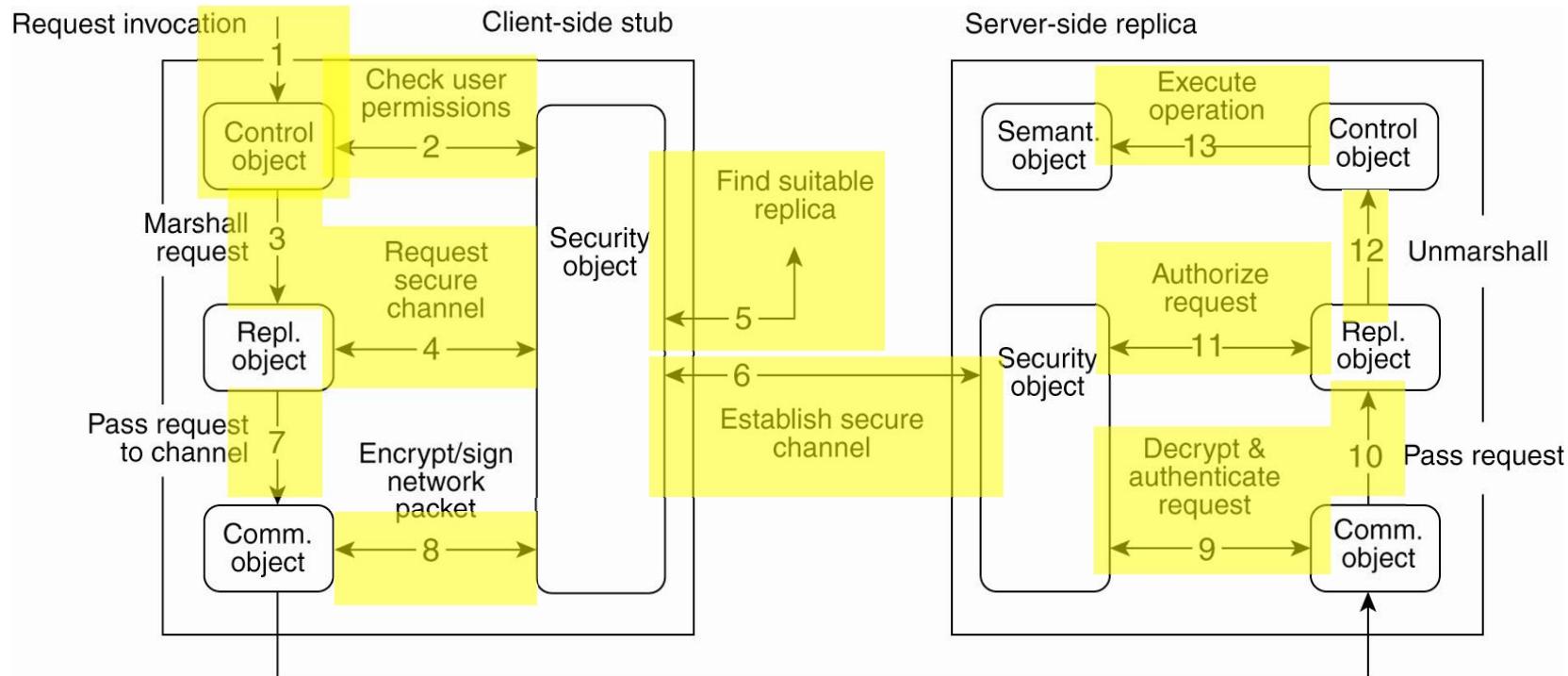
Replica certificate

K^+_{Repl}
$R: 1100011100$
$\text{sig}(O, \{R, K^+_{Repl}\})$

Administrative certificate

K^+_{Adm}
$R: 1101111100$
$U: 011001111$
$D: 1$
$\text{sig}(O, \{R, U, D, K^+_{Adm}\})$

Secure Method Invocation



①. Control object handles the check for user permissions and the initiation of the secure channel.
 ②. An interface object handles the marshaling and unmarshaling of the request information.
 ③. An interface object handles the connection with the client application.
 ④. An interface object handles the connection with the server application.
 ⑤. An interface object handles the connection with the replicated objects.
 ⑥. An interface object handles the connection with the security objects.

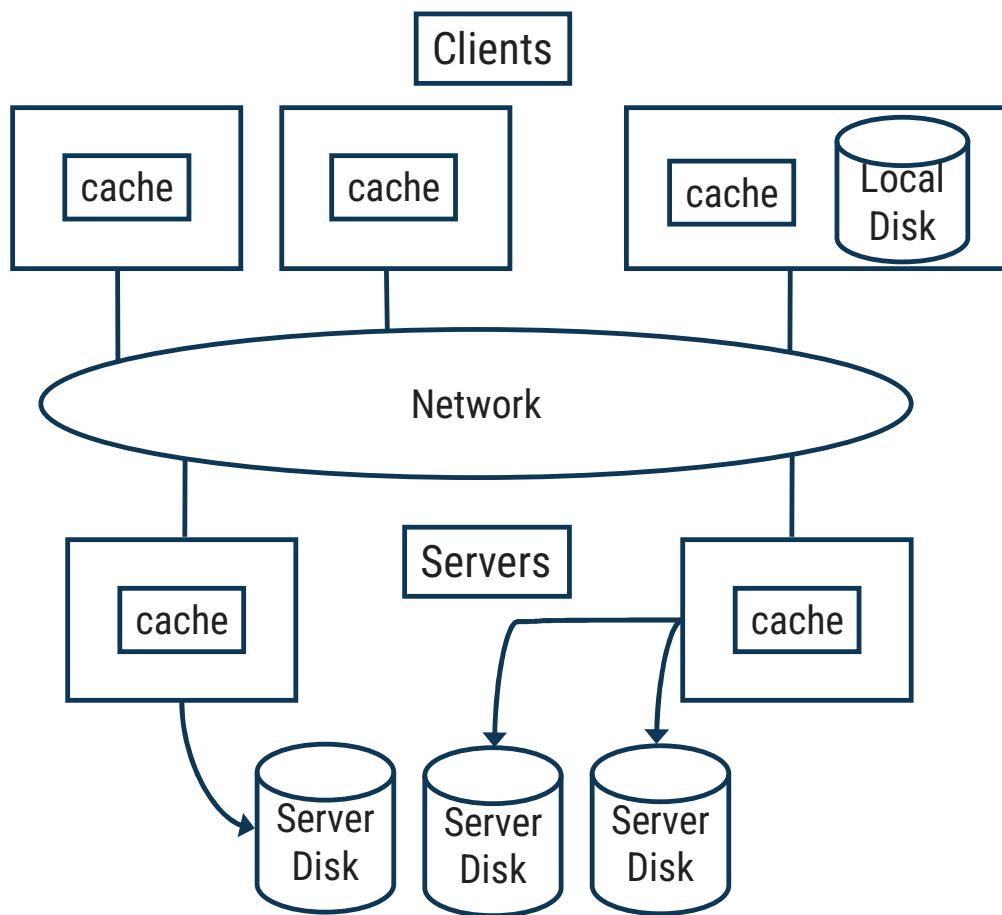
Categories of Distributed System

Distributed File System

What is Distributed File System?

- ▶ A distributed file system (DFS) is a **file system with data stored on a server**.
- ▶ The data is accessed and processed as if it was stored on the local client machine.
- ▶ The DFS makes it convenient to **share information** and files among users on a network in a controlled and authorized way.
- ▶ Server allows the client users to share files and store data just like they are storing the information locally.
- ▶ Servers have full control over the data and give access control to the clients.

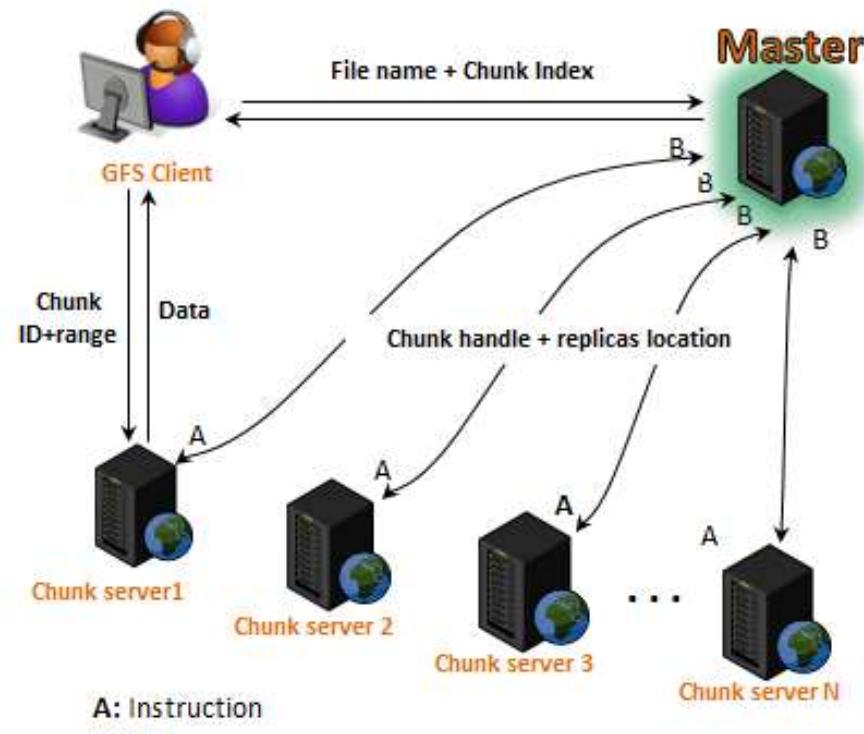
Distributed File System



Distributed file system is a part of distributed system that provides a user with a unified view of the files on the network.

Distributed File System- Example

- ▶ Google File System (GFS or GoogleFS) is a distributed file system developed by Google to provide efficient, reliable access to data using large clusters of hardware.



A: Instruction

B: State

Benefits of DFSs

- ▶ **File sharing over a network:** without a DFS, we would have to exchange files by e-mail or use applications such as the Internet's FTP
- ▶ **Transparent file accesses:** A user's programs can access remote files as if they are local. The remote files have no special APIs; they are accessed just like local ones
- ▶ **Easy file management:** managing a DFS is easier than managing multiple local file systems

DFS Component Placement

- ▶ **Access speed:** Caching information on clients improves performance considerably
- ▶ **Consistency:** If clients cache information, do all parties share the same view of it?
- ▶ **Recovery:** If one or more computers crash, to what extent are the others affected? How much information is lost?

DFS Aspects

Aspect	Description
Architecture	How are DFSs generally organized?
Processes	<ul style="list-style-type: none">Who are the cooperating processes?Are processes <i>stateful</i> or <i>stateless</i>?
Communication	<ul style="list-style-type: none">What is the typical communication paradigm followed by DFSs?How do processes in DFSs communicate?
Naming	How is naming often handled in DFSs?
Synchronization	What are the file sharing semantics adopted by DFSs?
Consistency and Replication	What are the various features of client-side caching as well as server-side replication?
Fault Tolerance	How is fault tolerance handled in DFSs?

Network File System

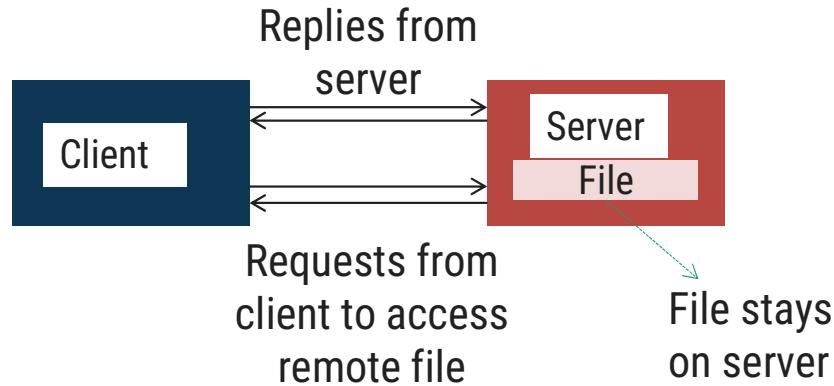
- ▶ Many distributed file systems are organized along the lines of client-server architectures
- ▶ Sun Microsystem's Network File System (NFS) is one of the most widely-deployed DFS for Unix-based systems
- ▶ NFS comes with a protocol that describes precisely how a client can access a file stored on a (remote) NFS file server
- ▶ NFS allows a heterogeneous collection of processes, possibly running on different OSs and machines, to share a common file system

Architectures

- ▶ Client-Server Distributed File Systems
- ▶ Cluster-Based Distributed File Systems
- ▶ Symmetric Distributed File Systems

Remote Access Model

- ▶ The model underlying NFS and similar systems is that of remote access model

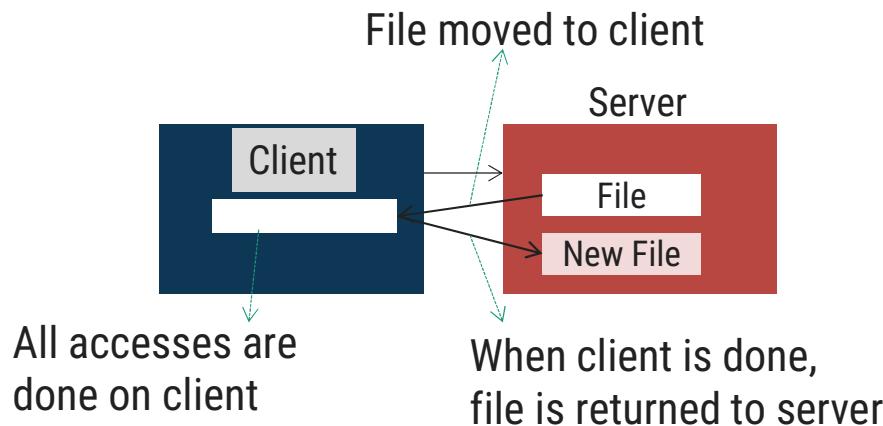


- ▶ In this model, clients:

- Are offered transparent access to a file system that is managed by a remote server
- Are normally unaware of the actual location of files
- Are offered an interface to a file system similar to the interface offered by a conventional local file system

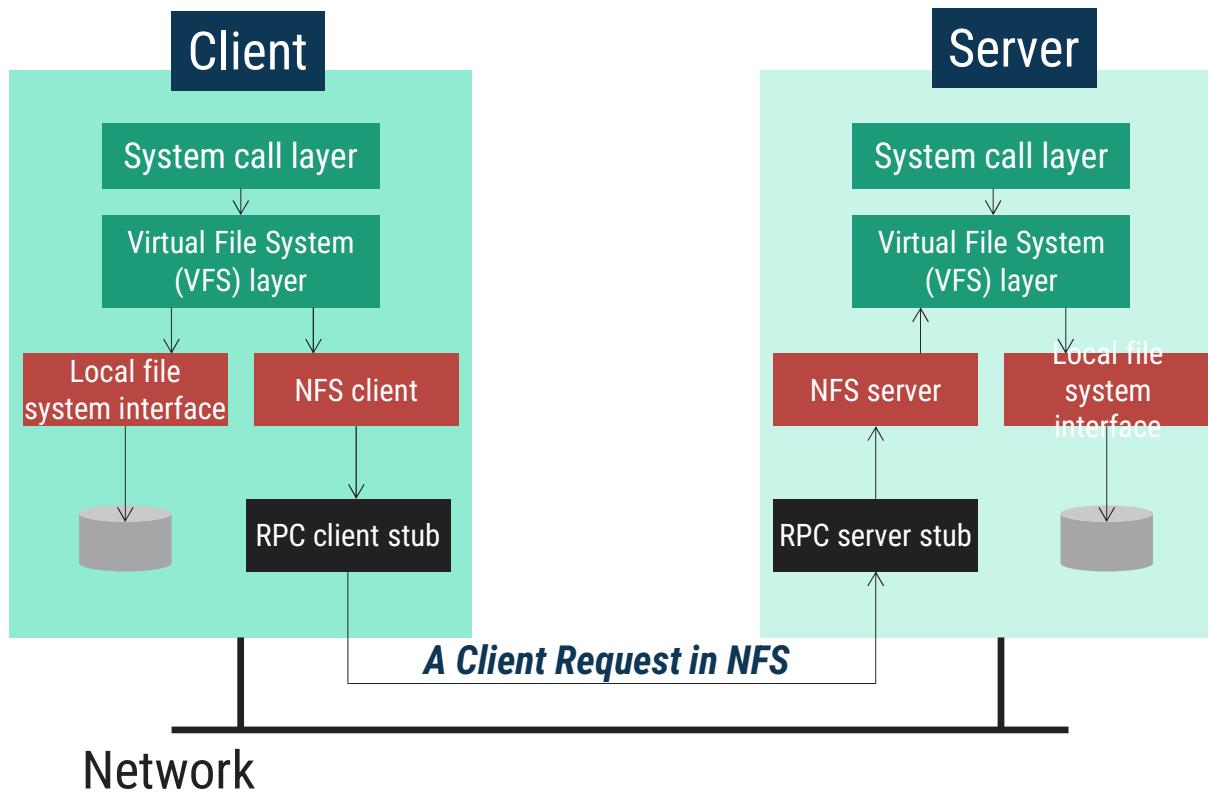
Upload/Download Model

- ▶ It attempts to reduce the amount of network traffic by taking advantage of the locality feature found in file accesses.
- ▶ A contrary model, referred to as upload/download model, allows a client to access a file locally after having downloaded it from the server



- ▶ Data is copied from the server's node to the client's node and is cached there.
- ▶ The client's request is processed on the client's node itself by using the cached data.

The Basic NFS Architecture



NFS File Operations

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory

NFS File Operations

Operation	v3	v4	Description
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Get the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

Architectures

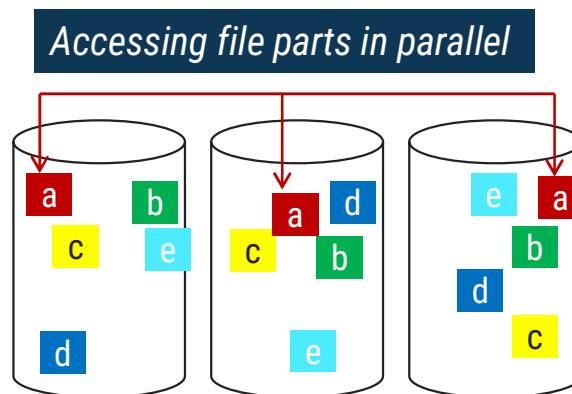
- ▶ Client-Server Distributed File Systems
- ▶ Cluster-Based Distributed File Systems
- ▶ Symmetric Distributed File Systems

Parallel Applications

- ▶ **Problems are increasingly becoming computationally challenging**
 - Typically run on large parallel machines
 - Critical to leveraging parallelism in all phases
- ▶ Data access is a huge challenge
 - We can also use parallelism in accessing data to obtain performance gains
- ▶ **File striping techniques**

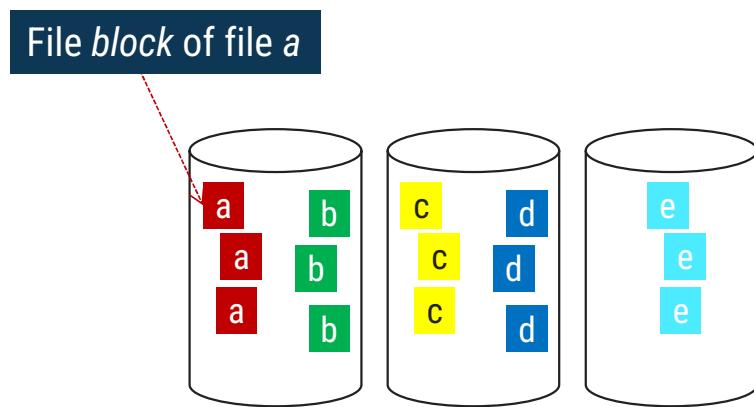
File Striping Techniques

- ▶ Server clusters are often used for parallel applications and their associated file systems are adjusted to satisfy their requirements
- ▶ One well-known technique is to deploy file-striping techniques, by which a single file is distributed across multiple servers
- ▶ Hence, it becomes possible to fetch different parts in parallel



General Purpose Applications

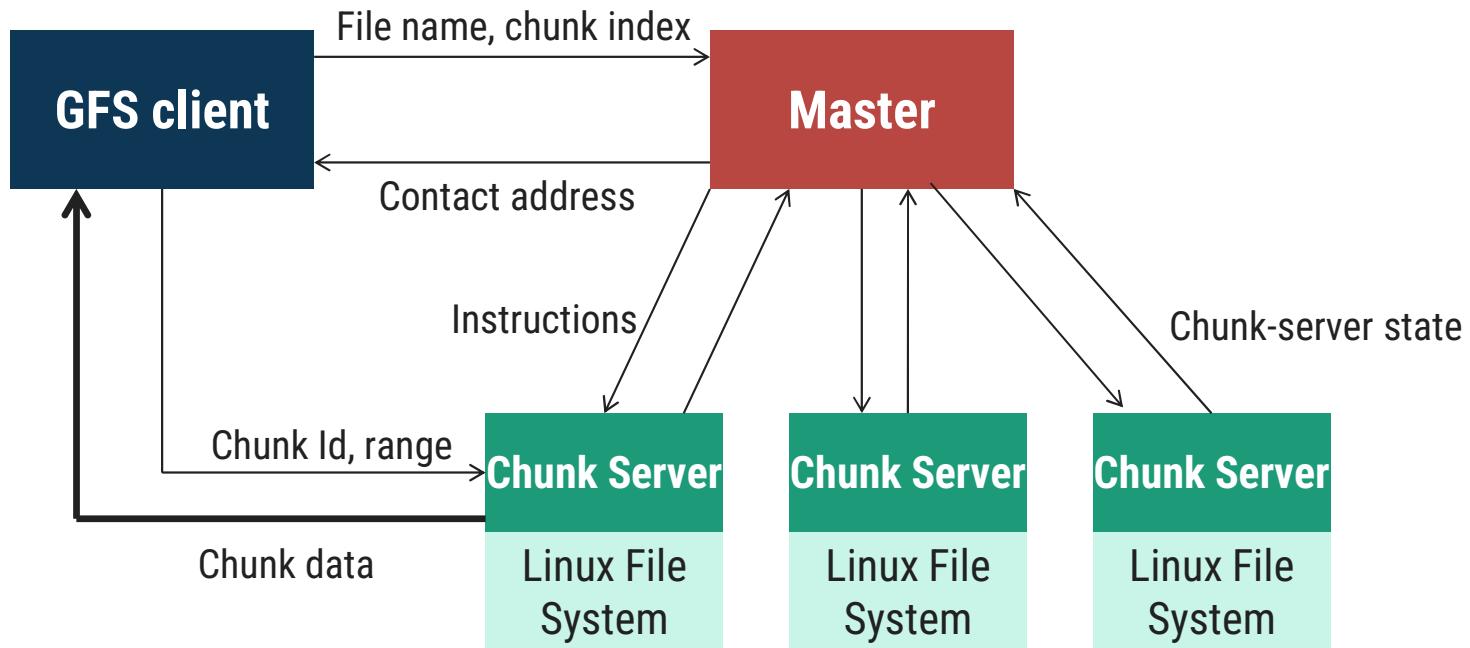
- ▶ For general-purpose applications, or those with irregular or many different types of data structures, file striping may not be effective
- ▶ In those cases, it is often more convenient to partition the file system as a whole and simply store files on different servers



Google File System

- ▶ Companies like Amazon and Google offer services to Web clients resulting in reads and updates to a massive number of files distributed across literally tens of thousands of computers
- ▶ In such environments, the traditional assumptions concerning distributed file systems might no longer hold
 - Files tend to be very large (TBs or PBs)
 - For example, we can expect that at any single moment there will be a computer malfunctioning
- ▶ To address these problems, Google, for example, has developed its own Google File System (GFS)

Google File System



GFS Mechanics

- ▶ Each GFS cluster consists of a single master along with multiple chunk servers
- ▶ Each GFS file is divided up into chunks of 64MB
- ▶ These chunks are distributed across chunk servers (i.e., using file-striping technique)
- ▶ The GFS master allocates chunks to chunk servers
- ▶ The master keeps track of where a chunk is located
- ▶ The GFS master is contacted for metadata information
- ▶ The GFS master maintains a name space, along with a mapping from file name to chunks
- ▶ The chunk servers keep an account of what they have stored
- ▶ Chunks are replicated (**server-side- No data caching; but it does cache metadata**) to handle failures (**No RAID, No SAN**)
- ▶ Replicas are updated serially

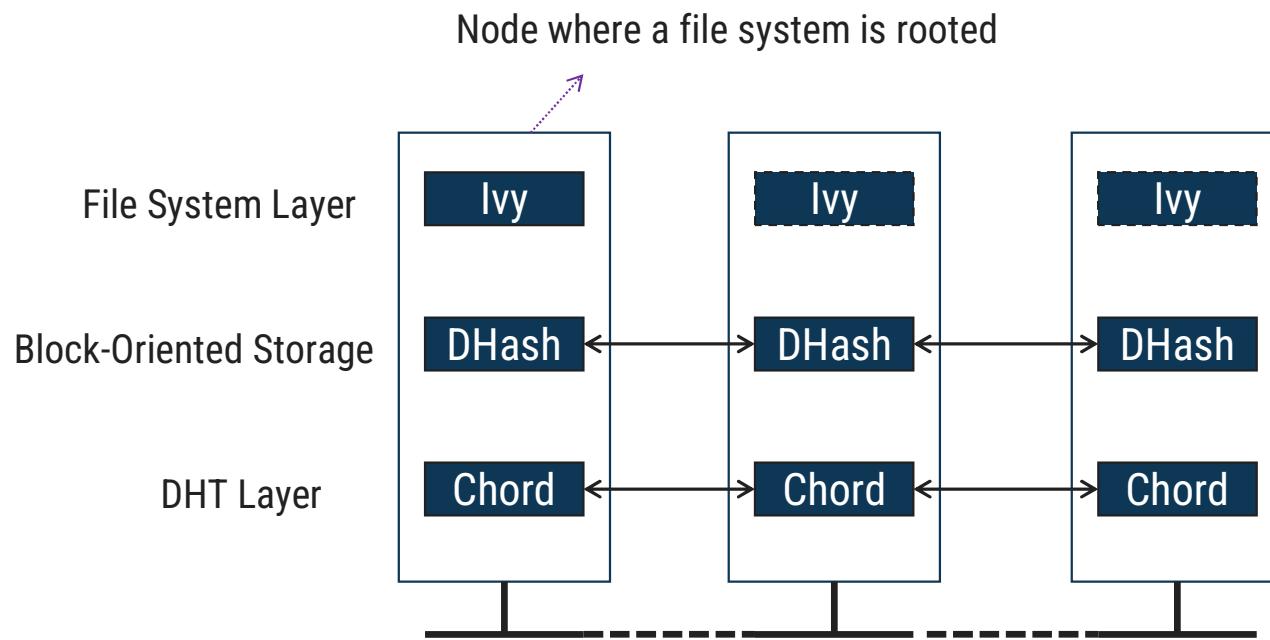
Architectures

- ▶ Client-Server Distributed File Systems
- ▶ Cluster-Based Distributed File Systems
- ▶ Symmetric Distributed File Systems

- ▶ Fully symmetric organizations that are based on peer-to-peer technology also exist
- ▶ All current proposals use a DHT-based system for distributing data, combined with a key-based lookup mechanism
- ▶ As an example, Ivy is a distributed file system that is built using a Chord DHT-based system
- ▶ Data storage in Ivy is realized by a block-oriented (i.e., blocks are distributed over storage nodes) distributed storage called DHash

Ivy Architecture

Ivy consists of 3 separate layers:

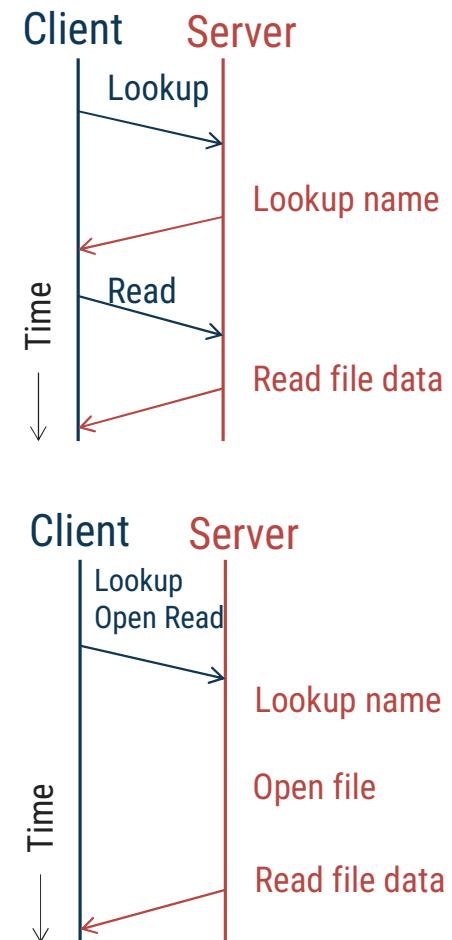


Communication

- ▶ Communication in DFSs is mainly based on remote procedure calls (**RPCs**)
- ▶ For example, in NFS, all communication between a client and server proceeds along the Open Network Computing RPC (**ONC RPC**)
- ▶ The main reason for choosing RPC is to make the system independent from underlying OSs, networks, and transport protocols
- ▶ Every NFS operation can be implemented as a single RPC call to a file server

RPCs in NFS

- ▶ Up until NFSv4, the client was made responsible for making the server's life as easy as possible by keeping requests simple
- ▶ The drawback becomes apparent when considering the use of NFS in a wide-area system
- ▶ In that case, the extra latency of a second RPC leads to performance degradation
- ▶ To circumvent such problems, NFSv4 supports compound procedures

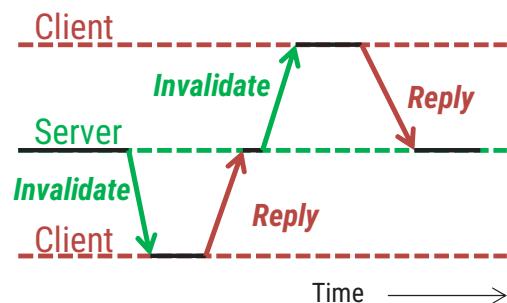


RPC in Coda (Content Delivery Architecture)

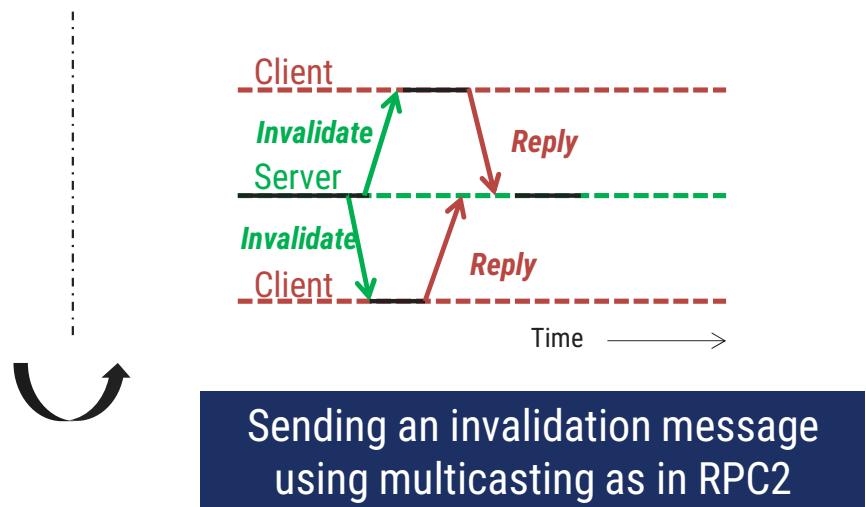
- ▶ Another enhancement to RPCs has been developed as part of the Coda file system (Kistler and Satyanarayanan, 1992) and referred to as RPC2
- ▶ RPC2 is a package that offers reliable RPCs on top of the (unreliable) UDP protocol
- ▶ Each time an RPC is made, the RPC2 client code starts a new thread that sends an invocation request to the server then blocks until it receives an answer
- ▶ As request processing may take an arbitrary time to complete, the server regularly sends back messages to the client to let it know it is still working on the request

RPC in Coda

- ▶ RPC2 supports multicasting
- ▶ An important design issue in Coda is that servers keep track of which clients have a local copy of a file (i.e., stateful servers)
- ▶ When a file is modified, a sever invalidates local copies at clients in parallel using multicasting



Sending an invalidation message one at a time

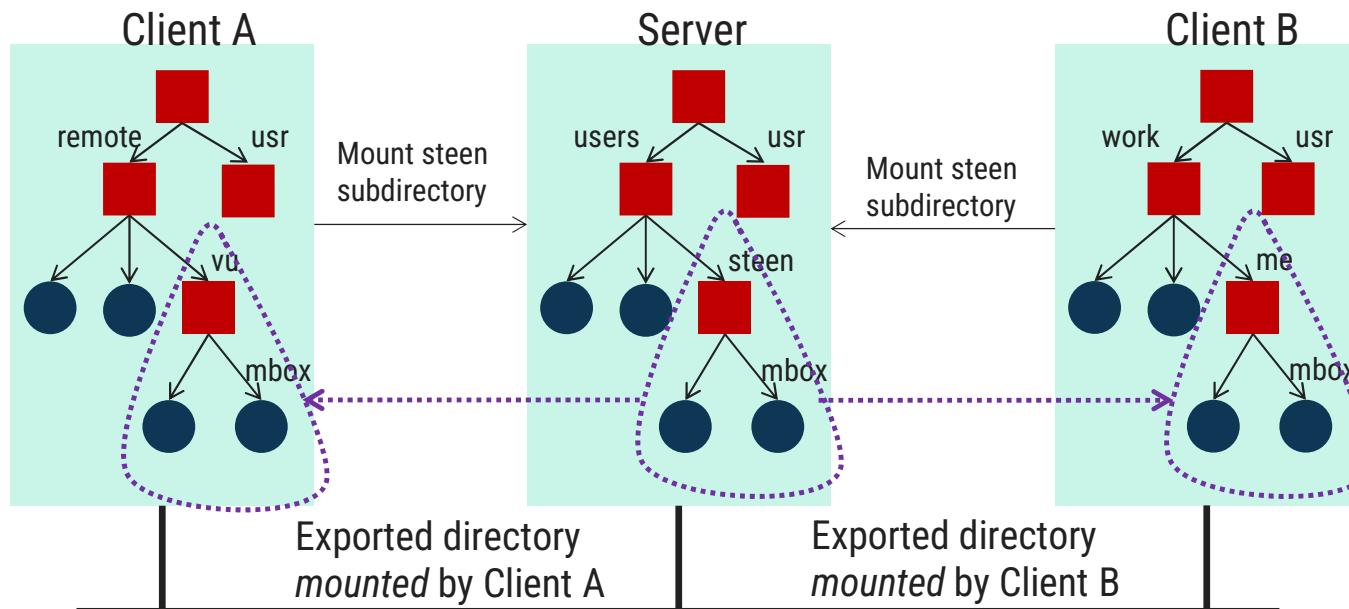


Sending an invalidation message using multicasting as in RPC2

Naming In NFS

- ▶ The fundamental idea underlying the NFS naming model is to provide clients with complete transparency
- ▶ Transparency in NFS is achieved by allowing a client to mount a remote file system into its own local file system
- ▶ However, instead of mounting an entire file system, NFS allows clients to mount only part of a file system
- ▶ A server is said to export a directory to a client when a client mounts a directory, and its entries, into its own name space

Mounting in NFS



The file named /remote/vu/mbox
at Client A

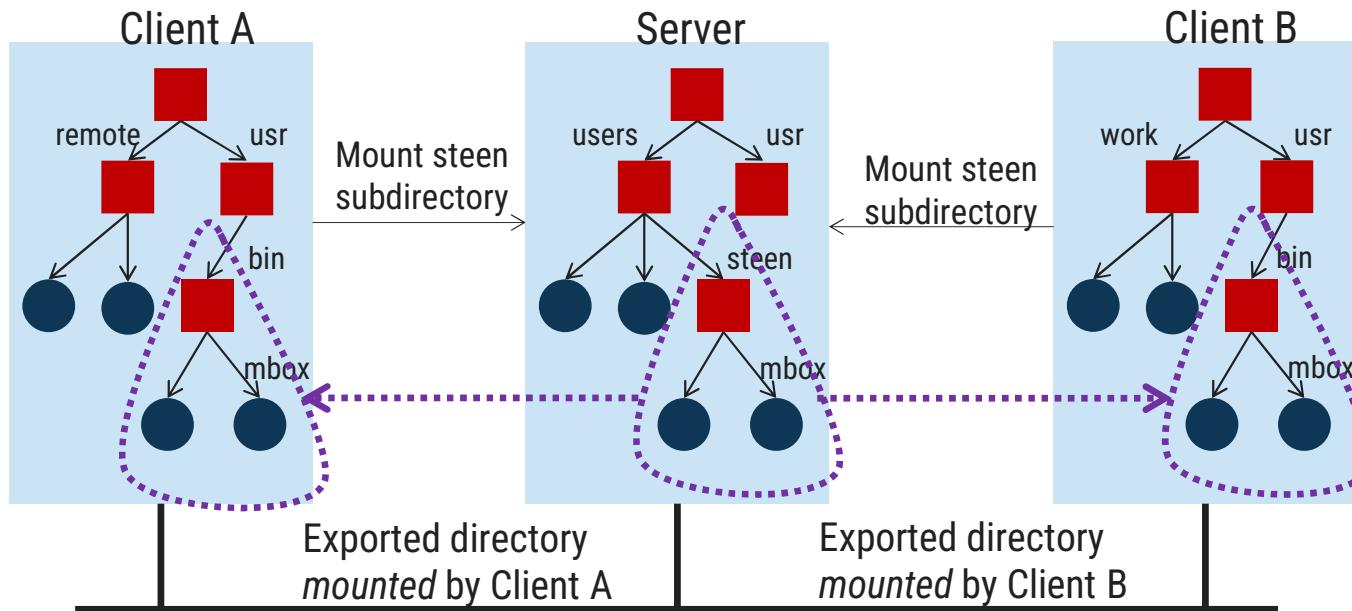
The file named /work/vu/mbox
at Client B

Sharing files becomes harder

Sharing Files In NFS

- ▶ A common solution for sharing files in NFS is to provide each client with a name space that is partly standardized
- ▶ For example, each client may by using the local directory /usr/bin to mount a file system
- ▶ A remote file system can then be mounted in the same manner for each user

Sharing Files In NFS



The file named `/usr/bin/mbox`
at Client A

Sharing files resolved

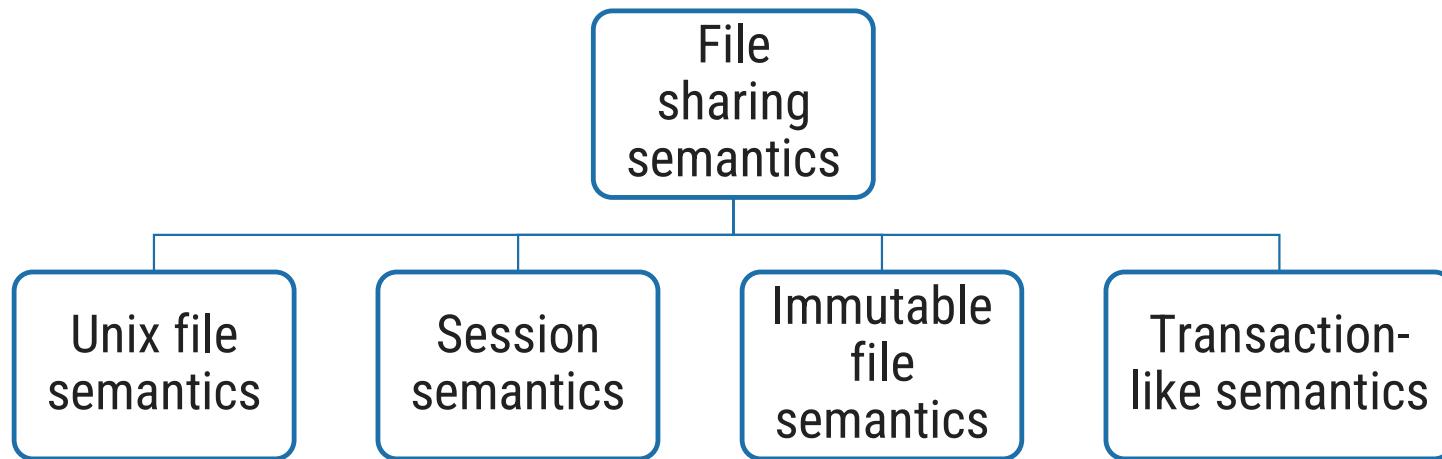
The file named `/usr/bin/mbox`
at Client B

Synchronization In DFSs

1. File Sharing Semantics
2. Lock Management

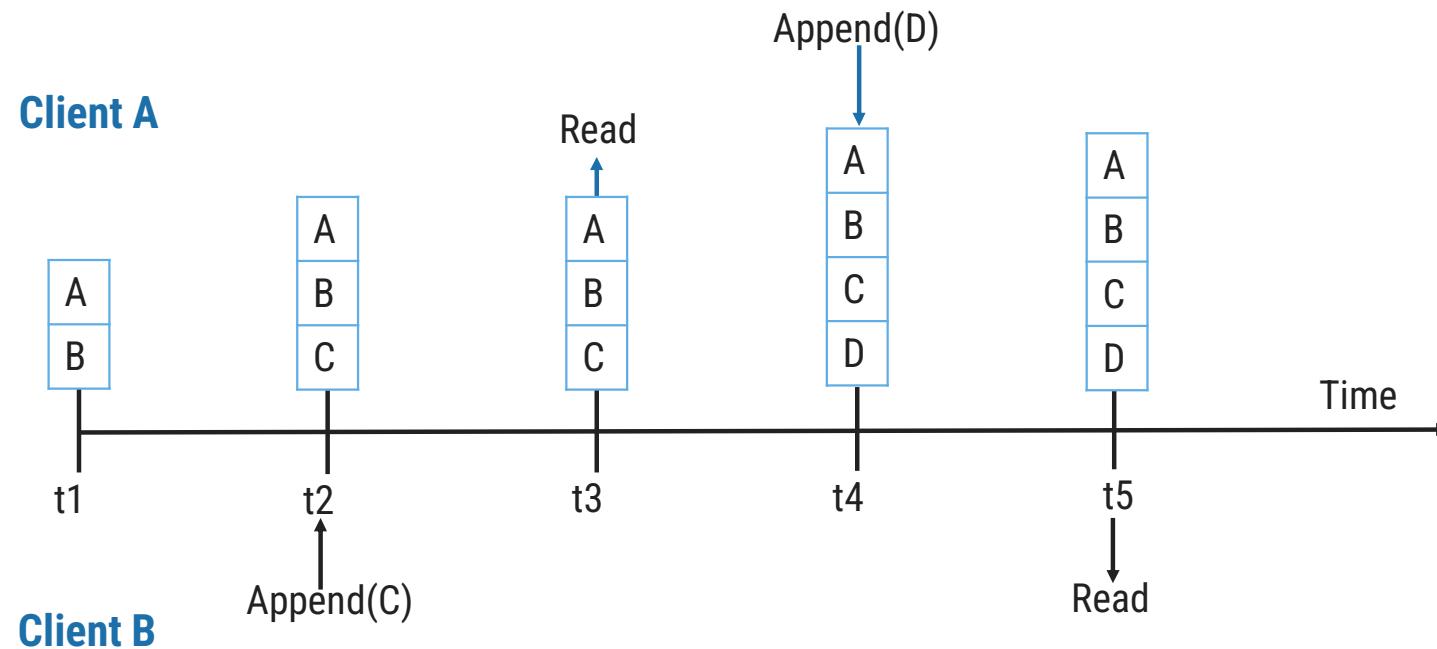
File Sharing Semantics

- ▶ A shared file may be simultaneously accessed by multiple users.
- ▶ In such a situation, an important design issue for any file system is to clearly define when modifications of file data made by a user are observable by other users.



File Sharing Semantics - Unix File Semantics

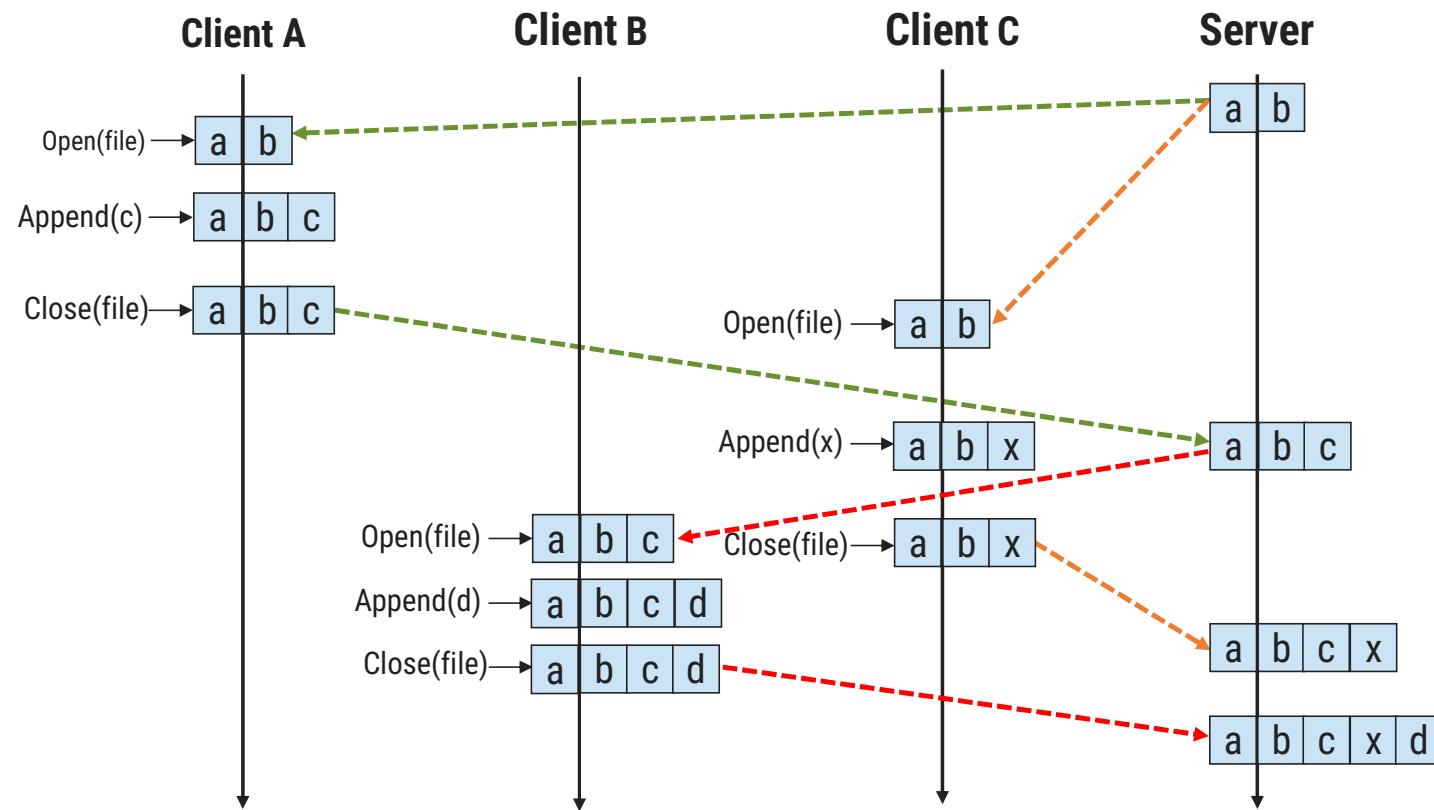
- Every read operation on a file sees the effects of all previous write operations performed on that file.



File Sharing Semantics - Session Semantics

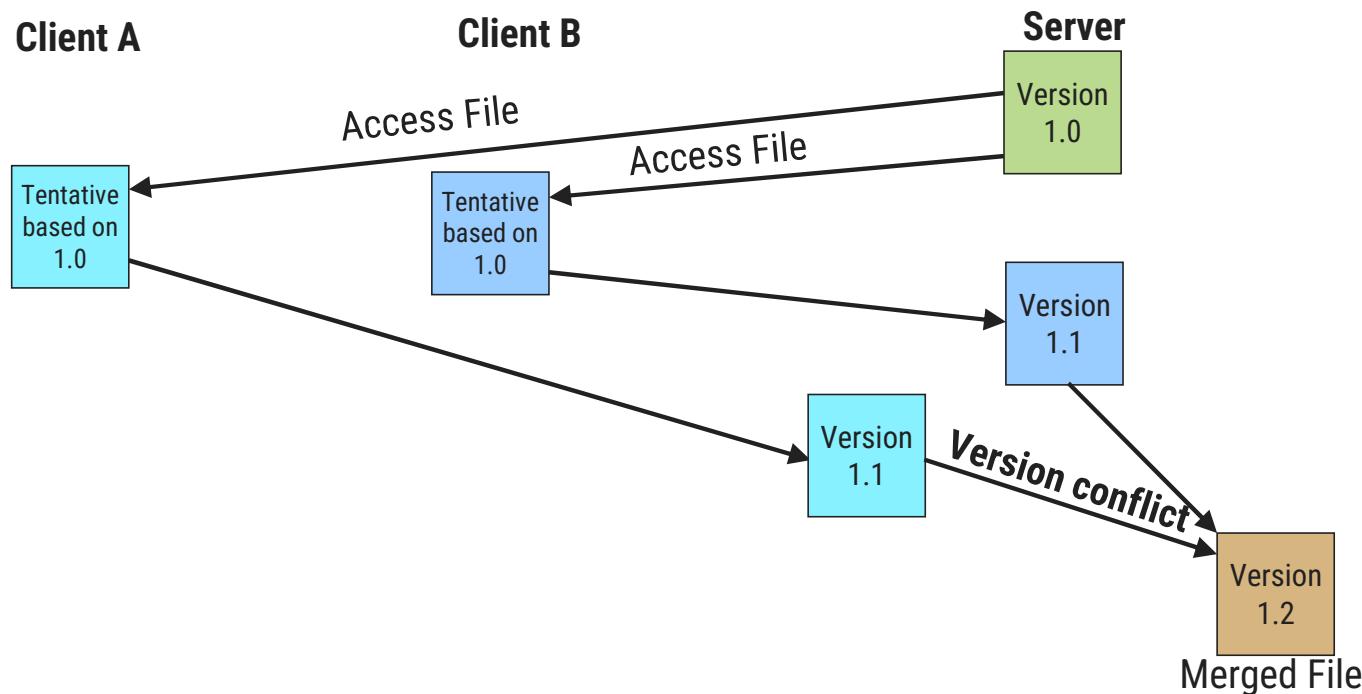
- ▶ In session semantics, all changes made to a file during a session are initially made visible only to the client process that opened the session.
- ▶ It is invisible to other remote processes who have the same file open simultaneously.
- ▶ Once the session is closed, the changes made to the file are made visible to remote processes only in later starting sessions.

File Sharing Semantics - Session Semantics



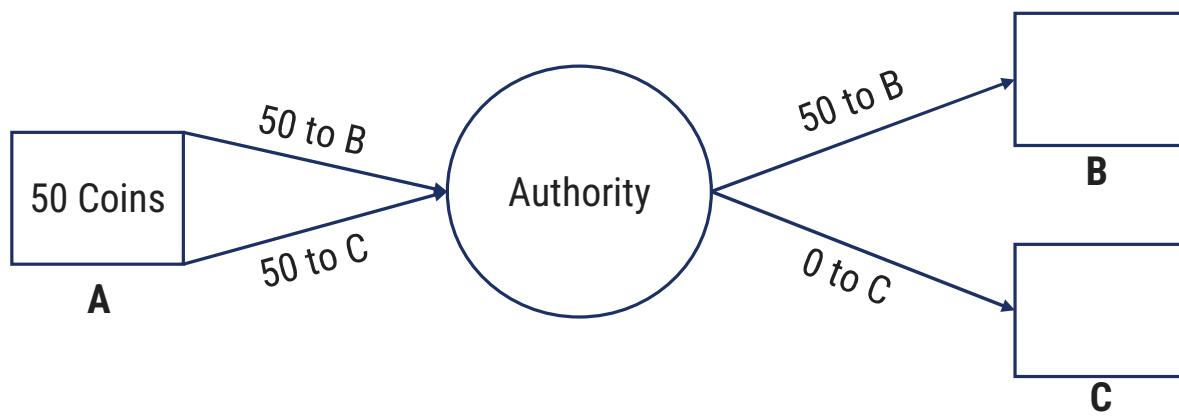
File Sharing Semantics - Immutable Shared-files Semantics

- ▶ Change to the file are handled by creating a new updated version of the file.
- ▶ Therefore, the semantics allows files to be shared only in the read-only mode.



File Sharing Semantics- Transaction like Semantics

- ▶ A transaction is a set of operations enclosed in-between a pair of begin_transaction and end_transaction like operations.
- ▶ All the transactions will be carried out in order, without any interference from other concurrent transactions.
- ▶ Partial modifications will not be visible to other concurrently executing transactions until the transaction ends.
- ▶ The example of this is a banking system.



Categories of Distributed System

Distributed Web-Based Systems

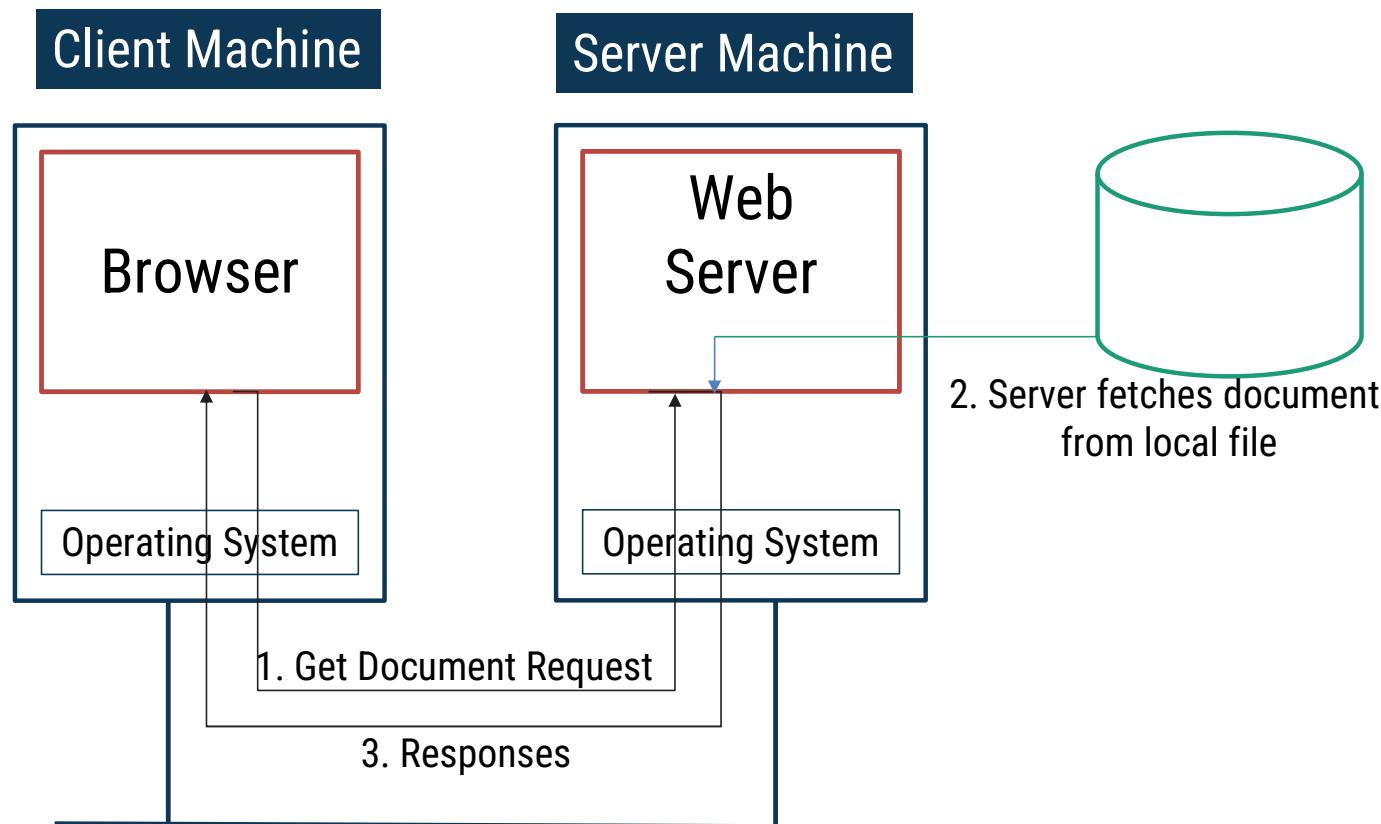
Distributed Web-Based Systems

- ▶ The World Wide Web (WWW) can be viewed as a huge distributed system consisting of millions of clients and servers for accessing linked documents.
- ▶ Servers maintain collections of documents, while clients provide users an easy-to-use interface for presenting and accessing those documents.

Architecture

- ▶ Many Web-based systems are organized as simple client-server architecture.
- ▶ The simplest way to refer to a document is by means of a reference called a Uniform Resource Locator (URL).
- ▶ It specifies where a document is located, often by embedding the Domain name server (DNS) of its associated server.
- ▶ A URL specifies the application-level protocol for transferring the document across the network.
- ▶ A client interacts with Web servers through a special application known as a browser.
- ▶ A browser is responsible for properly displaying a document.

Traditional web based system



Web Documents

- ▶ Fundamental to the Web is that all information comes in the form of a document.
- ▶ Most documents can be divided into two parts:
 1. A main part that acts as a template.
 2. The second part consists of many different bits and pieces that jointly constitute the document that is displayed in a browser.
- ▶ The main part is generally written in a markup language (HTML, XML).
- ▶ Each embedded document has an associated MIME (Multipurpose Internet Mail Exchange) type.
- ▶ It was developed to provide information on the content of a message body that was sent as part of electronic mail.

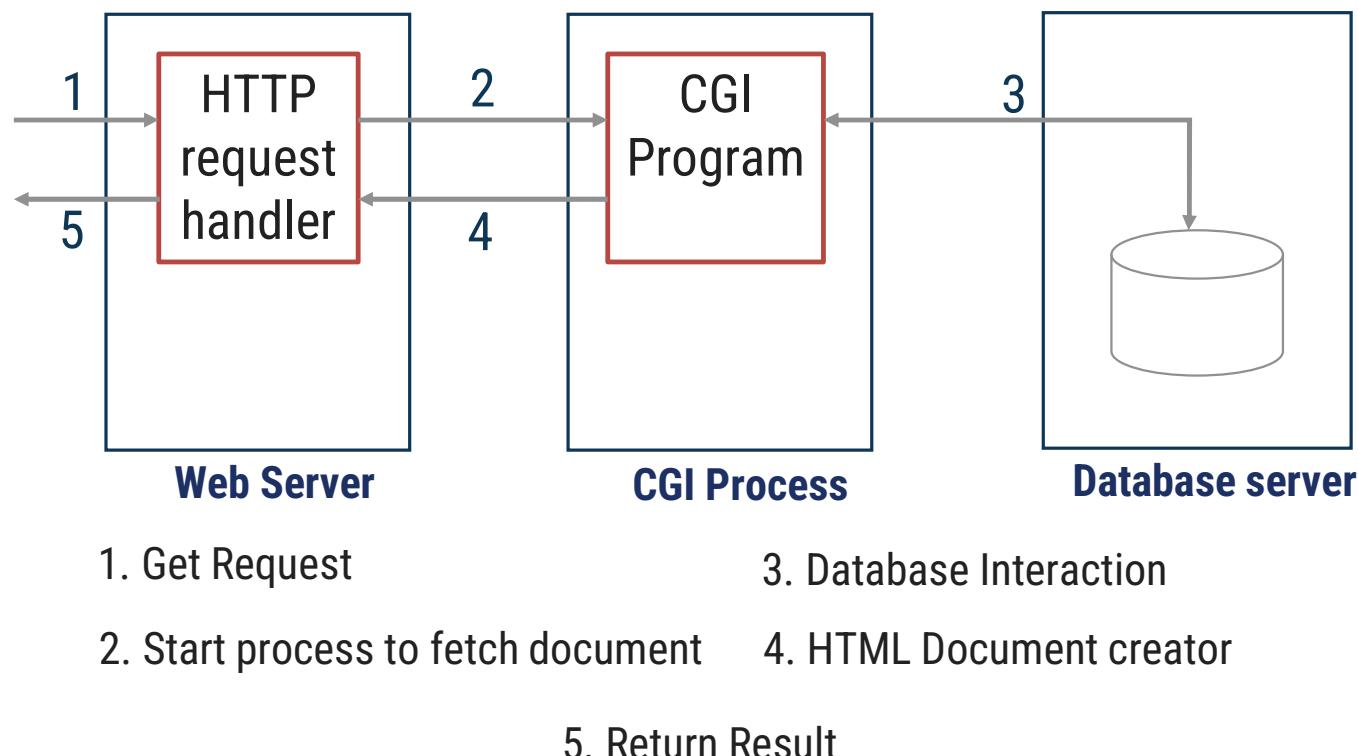
MIME Types

Type	Subtype	Description
Text	Plain	Unformatted text
	HTML	Text including HTML markup commands
	XML	Text including XML markup commands
Image	GIF	Image in GIF format
	JPEG	Image in JPEG format
Audio	Basic	Audio, 8-bit PCM sampled at 8000 Hz
	Tone	A specific audible tone
Video	MPEG	Movie in MPEG format
	Pointer	Representation of a pointer device for presentation
Multipart	Mixed	Independent parts in the specified order
	Parallel	Parts must be viewed simultaneously
Application	Octet-stream	A uninterpreted byte sequence
	Postscript	A printable document in postscript
	PDF	A printable document in PDF

Multitier Architecture

- ▶ Common Gateway Interface (CGI) defines a standard way by which a Web server can execute a program taking user data as input.
- ▶ User data come from an HTML forms.
- ▶ It specifies the program that is to be executed at the server side, along with parameter values that are filled in by the user.
- ▶ Once the form has been completed, the program's name and collected parameter values are sent to the server.
- ▶ When the server get a request it starts the program named in the request and passes its parameter values.
- ▶ Program simply does its work and returns the results in the form of a document that is sent back to the user's browser to be displayed.

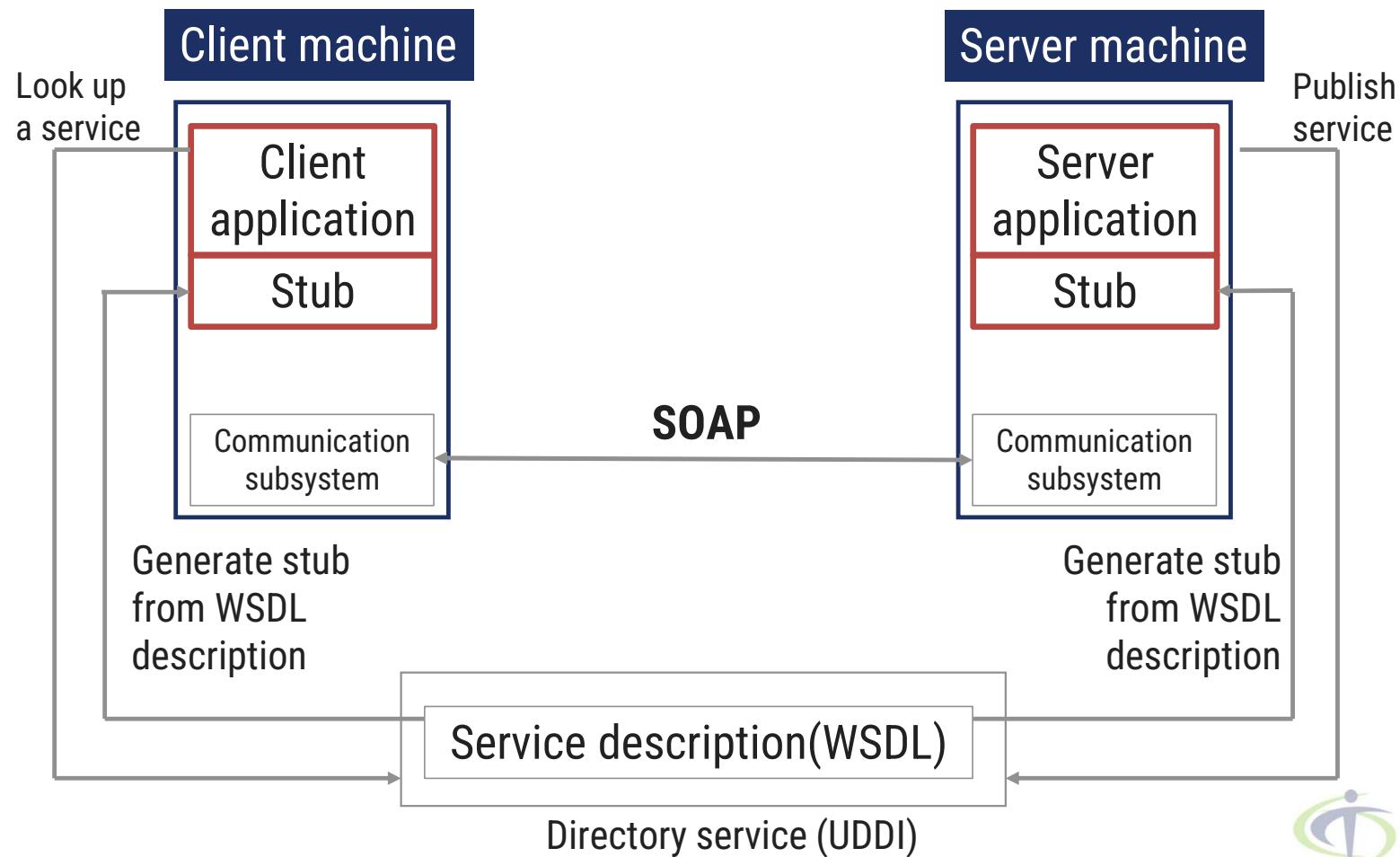
Multitier Architecture



Web Services

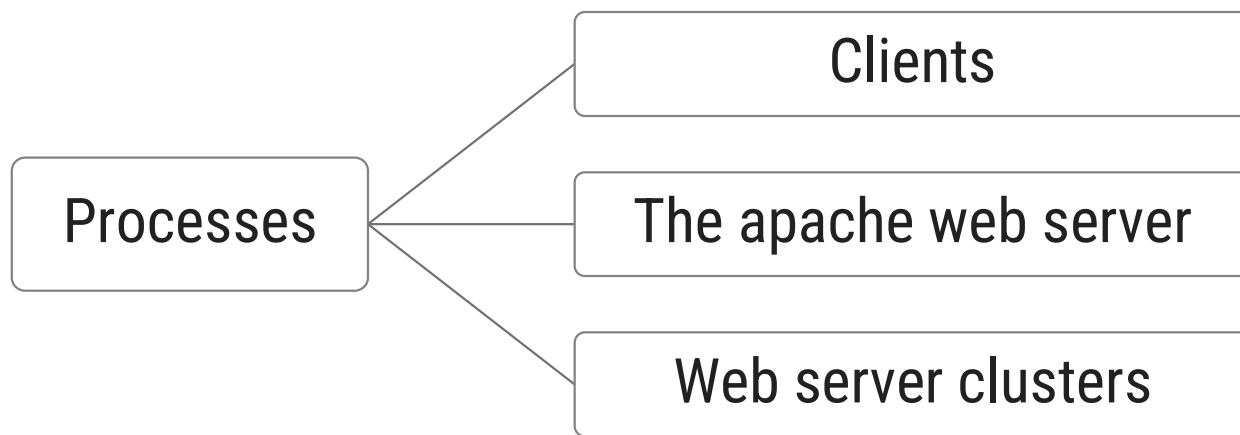
- ▶ Services available over the web.
- ▶ The basic idea is that some client application can call upon the services as provided by a server application.
- ▶ **UDDI** prescribes the layout of a database containing service descriptions that will allow Web service clients to browse for relevant services.
- ▶ UDDI stands for Universal Description, Discovery and Integration standard.
- ▶ Services are described by means of the Web Services Definition Language (**WSDL**).
- ▶ WSDL description contains the precise definitions of the interfaces provided by a procedure specification, data types and location.
- ▶ Core element of a Web service is the specification of how communication takes place.
- ▶ The Simple Object Access Protocol (**SOAP**) is used, which is essentially a framework in which much of the communication between two processes can be standardized.

Web Services



Processes

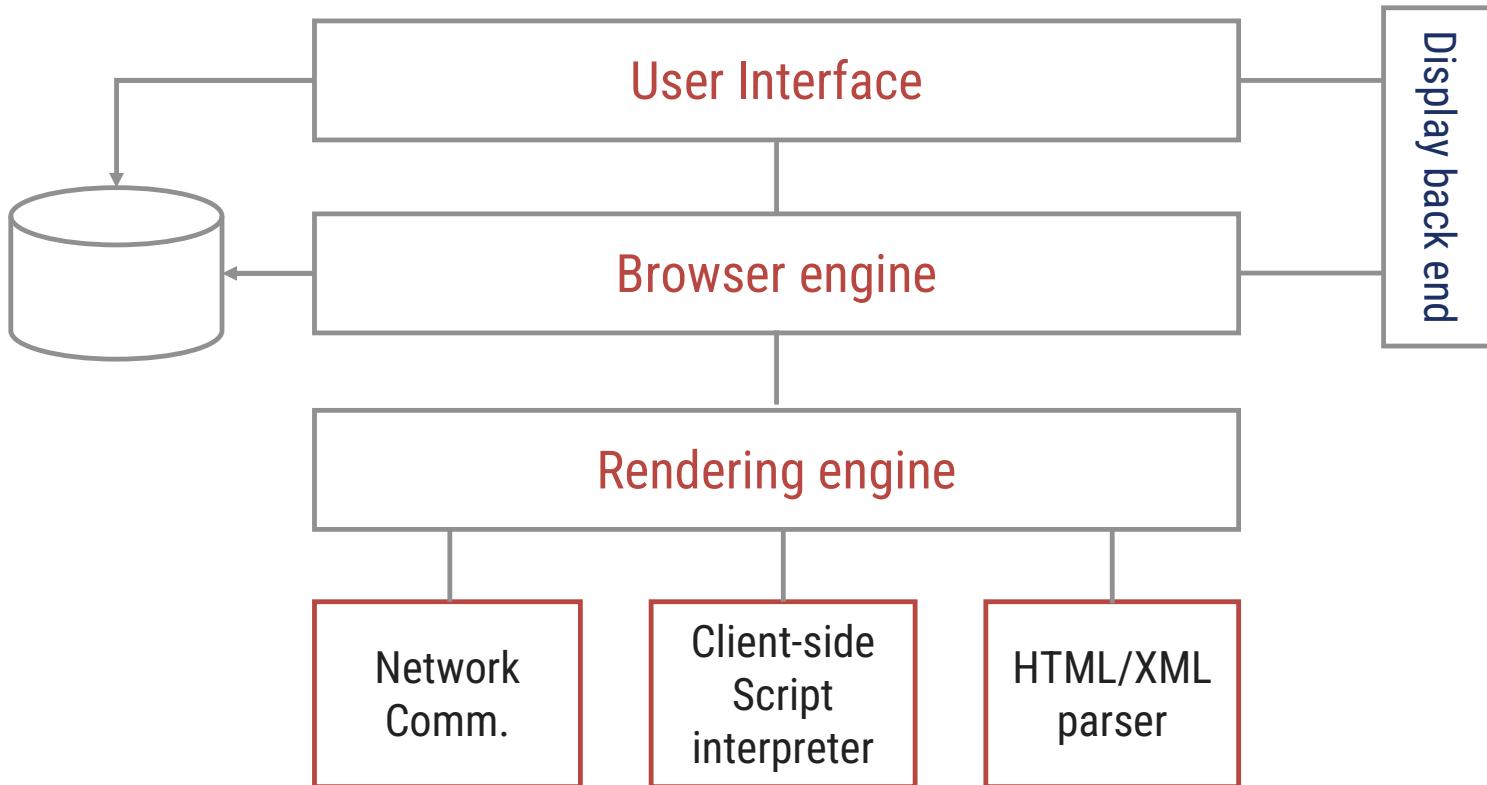
- ▶ The most important processes used in Web-based systems and their internal organization.



Clients

- ▶ The most important Web client is a piece of software called a Web browser.
- ▶ It enables a user to navigate through Web pages by fetching those pages from servers and subsequently displaying them on the users screen.
- ▶ The core of a browser is formed by the browser engine and the rendering engine.
- ▶ The rendering engine contains all the code for properly displaying documents.
- ▶ This rendering require parsing HTML or XML, but may also require script interpreter(for JavaScript).

Clients (Web Browser)

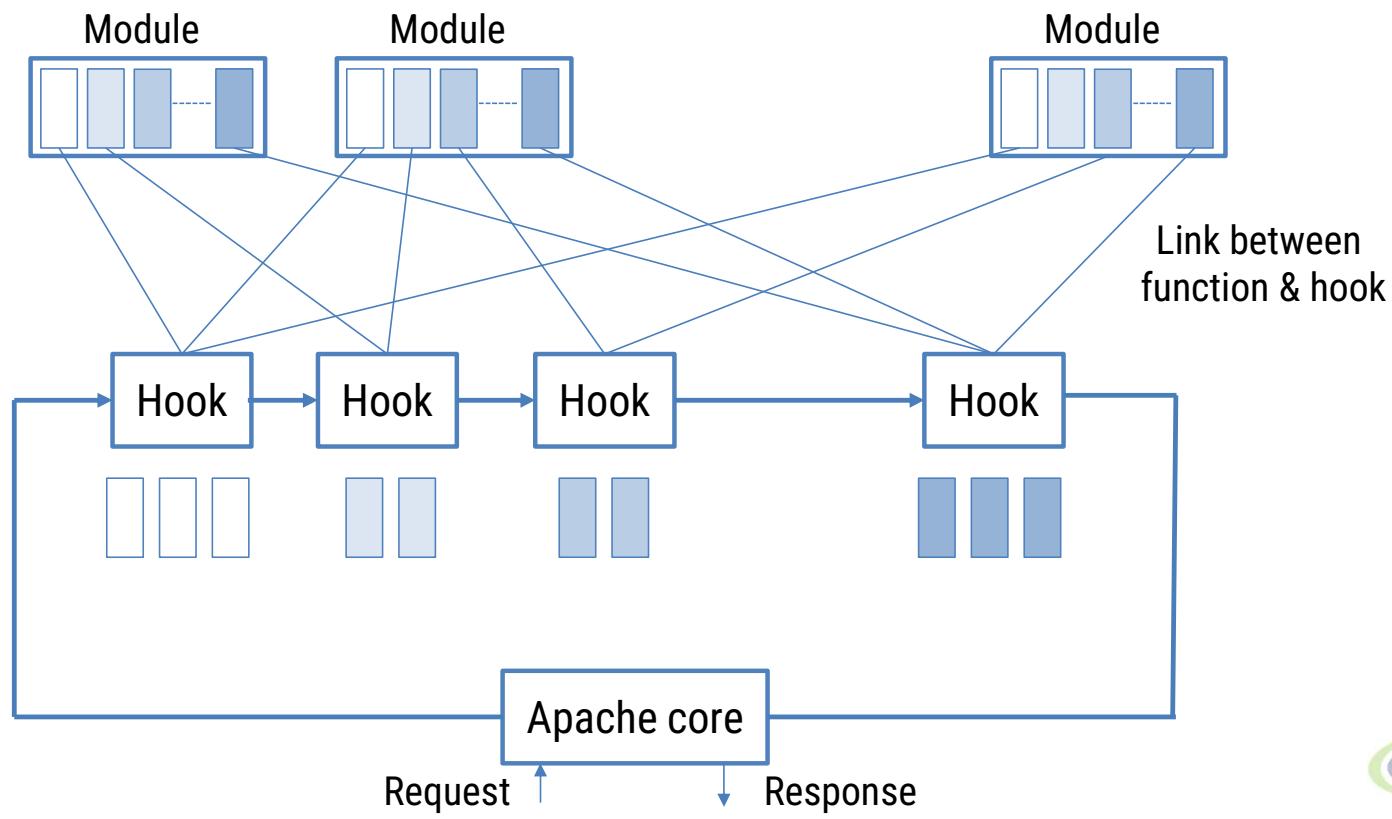


Apache Web Server

- ▶ Apache Portable Runtime (APR), is a library that provides a platform-independent interface for file handling, networking, locking and threading.
- ▶ Apache core assumes that requests are processed in a number of phases.
- ▶ Each phase consisting of a few hooks.
- ▶ Each hook represents a group of similar actions that need to be executed as part of processing a request.
- ▶ For example, there is a hook to translate a URL to a local file name.

Apache Web Server

► General organization of the Apache Web server



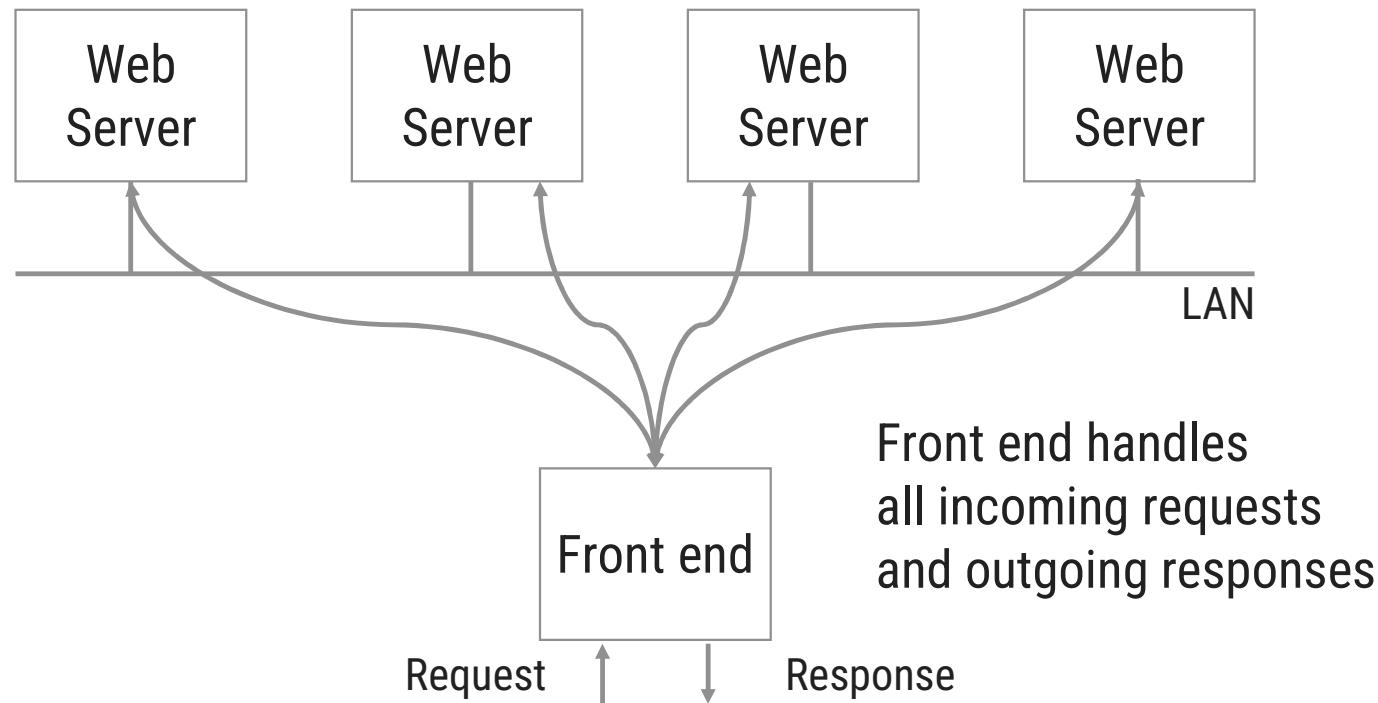
Web Server Clusters

- ▶ An important problem related to the client-server nature of the Web is that a Web server can easily become overloaded.
- ▶ A practical solution employed in many designs is to simply replicate a server on a cluster of servers.
- ▶ This principle is an example of horizontal distribution.
- ▶ The design of the front end becomes a serious performance bottleneck.
- ▶ Whenever a client issues an HTTP request, it sets up a TCP connection to the server.

Web Server Clusters

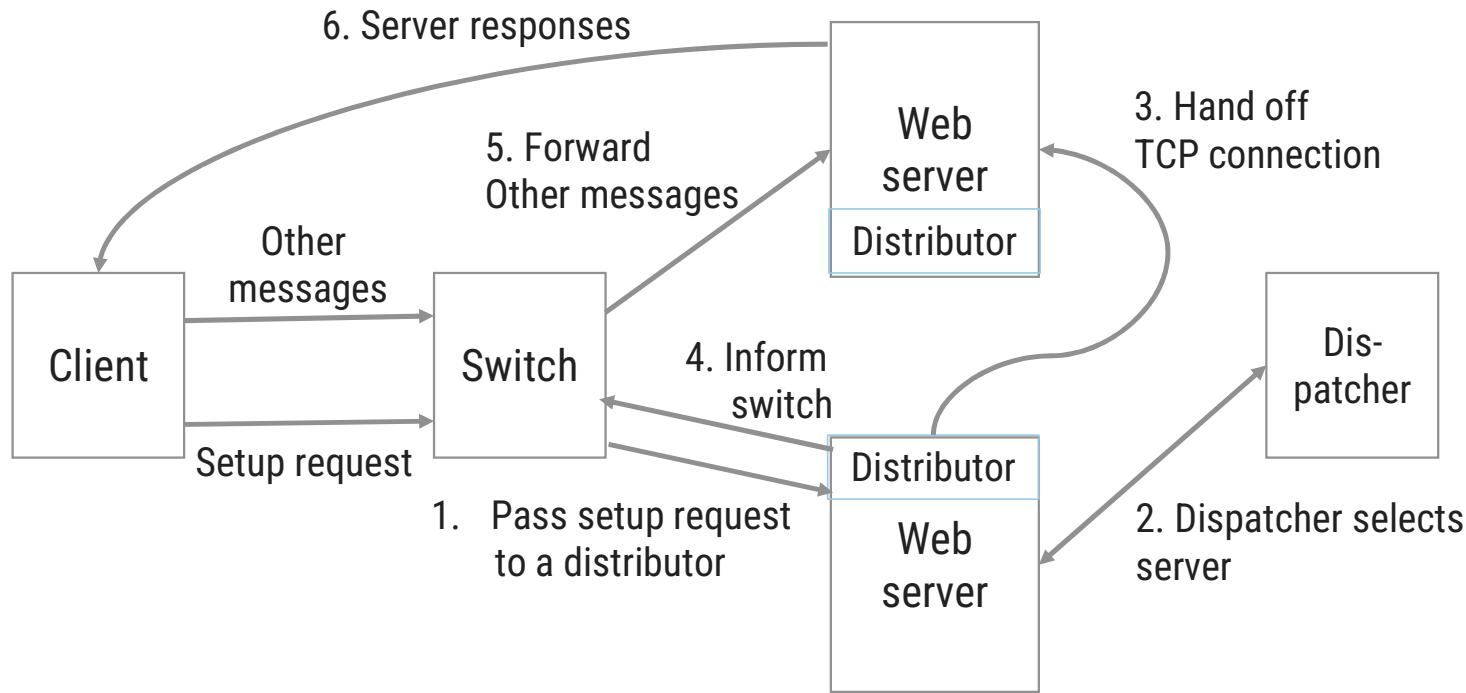
- ▶ A better approach is to deploy content-aware request distribution by which,
 - The front end first inspects an incoming HTTP request
 - Then decides which server it should forward that request
- ▶ In combination with TCP handoff, the front end has two tasks.
 1. When a request initially comes in, it must decide which server will handle the rest of the communication with the client.
 2. The front end should forward the client's TCP messages associated with the handed off TCP connection.
- ▶ The dispatcher is responsible for deciding to which server a TCP connection should be handed off.
- ▶ The switch is used to forward TCP messages to a distributor.

Web Server Clusters



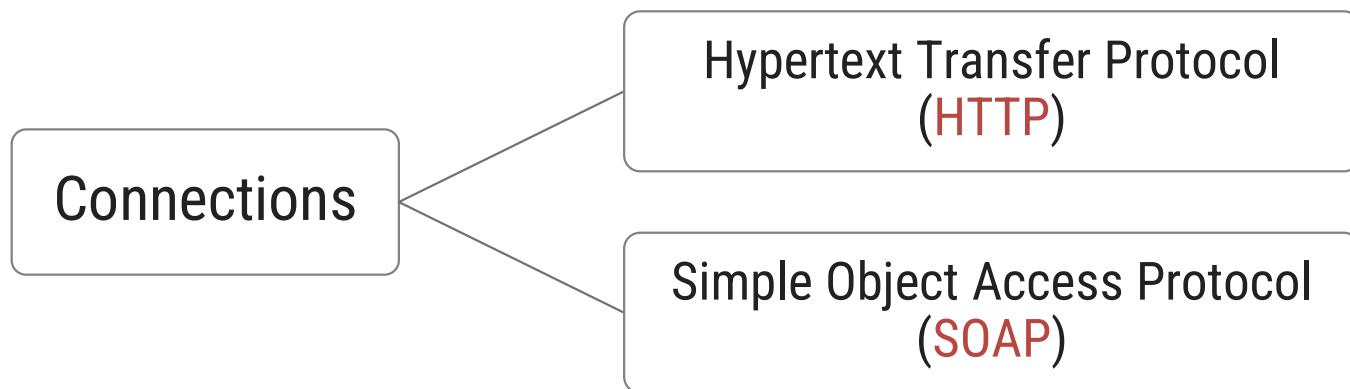
The principle of using a server cluster in combination
with a front end to implement a Web service.

Web Server Clusters



A scalable content-aware cluster of web

Connections



Hypertext Transfer Protocol (HTTP)

- ▶ HTTP is based on TCP.
- ▶ Whenever a client issues a request to a server,
 - It first sets up a TCP connection to the server
 - Then sends its request message on that connection.
- ▶ The same connection is used for receiving the response.
- ▶ By using TCP as its underlying protocol, HTTP need not be concerned about lost request and responses.

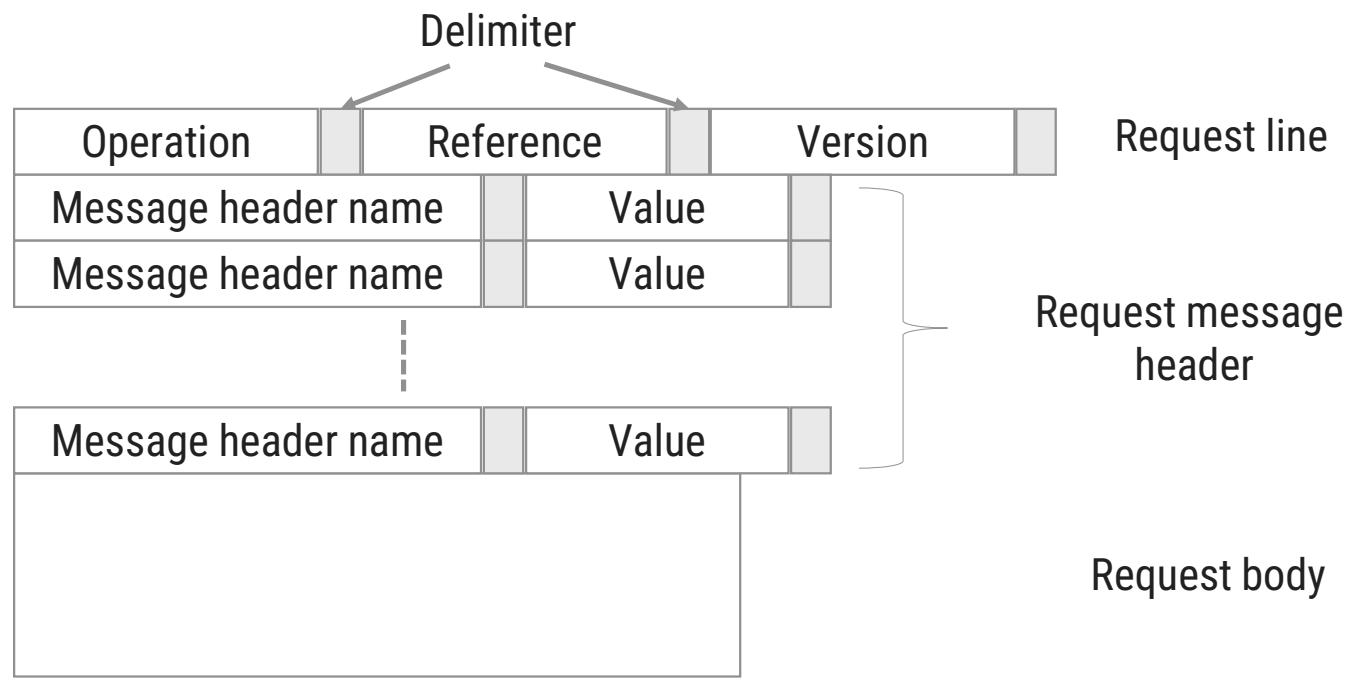
HTTP Methods

- ▶ A client can request each of these operations to be carried out at the server by sending a request message containing the operation desired to the server.
- ▶ Most commonly used request methods are as follows:

OPERATION	DESCRIPTION
Head	Request to return the header of a document
Get	Request to return a document to the client
Put	Request to store a document
Post	Provide data that are to be added to a document
Delete	Request to delete document

HTTP Messages

- ▶ All communication between a client and server takes place through messages.
- ▶ HTTP recognizes only request and response messages.



Simple Object Access Protocol (SOAP)

- ▶ The Simple object access protocol (SOAP) forms the standard for communication with Web services.
- ▶ A SOAP message generally consists of two parts, which are jointly put inside what is called a SOAP envelope.
- ▶ The body contains the actual message.
- ▶ Header is optional, containing information relevant for nodes along the path from sender to receiver.
- ▶ Everything in the envelope is expressed in XML.

Naming

- ▶ Web uses a single naming system to refer a documents.
- ▶ The names used are called Uniform resource identifiers (URIs).
- ▶ Uniform resource locator (URL) is a URI that identifies a document by including information on how and where to access the document.
- ▶ A URL is used as a globally unique, location-independent and persistent reference to a document.
- ▶ How to access a document is generally reflected by the name of the scheme that is part of the URL such as http, ftp or telnet.

Naming

- ▶ URL also contains the name of the document to be looked up by that server.
- ▶ General structure of URLs are as follows.

Scheme	Host name	Pathname
http ://	www.darshan.ac.in	/home/comp/faculty
1. Using only DNS name		
Scheme	Host name	Port
http ://	www.darshan.ac.in : 80	/home/comp/faculty
2. Combining DNS name with port number		
Scheme	Host name	Port
http ://	130.37.24.11 : 80	/home/comp/faculty
3. Combining an IP address with port number		

Synchronization

- ▶ Synchronization has not been much of an issue for most traditional Web- based systems for two reasons.
 1. The servers never exchange information with other servers means that there is nothing much to synchronize.
 2. The web can be considered as being a read-mostly system.
- ▶ Distributed authoring of Web documents is handled through a separate protocol called WebDAV (Web Distributed Authoring and Versioning)
- ▶ To synchronize concurrent access to a shared document, WebDAV supports a simple locking mechanism.

Synchronization

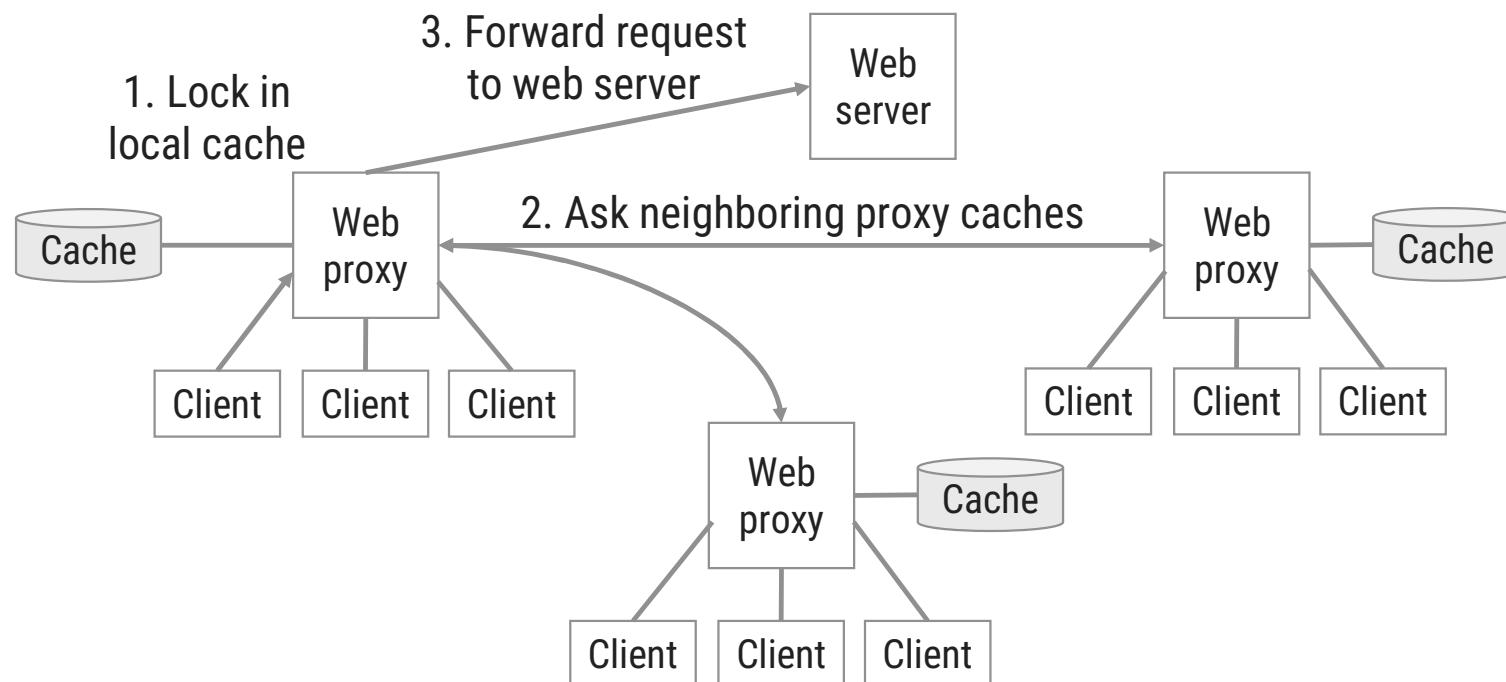
- ▶ There are two types of write locks.
 1. An exclusive write lock can be assigned to a single client, and will prevent any other client from modifying the shared document while it is locked.
 2. A shared write lock, which allows multiple clients to simultaneously update the document.
- ▶ Assigning a lock is done by passing a lock token to the requesting client.
- ▶ The server registers which client currently has the lock token.
- ▶ Whenever the client wants to modify the document, it sends an HTTP post request to the server along with the lock token.

Web Proxy Caching

- ▶ Web proxy accepts requests from local clients and passes these to Web servers.
- ▶ When a response comes in, the result is passed to the client.
- ▶ The advantage of this approach is that the proxy can cache the result and return that result to another client, if necessary.
- ▶ Web proxy can implement a shared cache.
- ▶ In addition to caching at browsers and proxies, it is also possible to place caches that cover a region, or even a country, thus leading to hierarchical caches.
- ▶ Such schemes are mainly used to reduce network traffic.

Web Proxy Caching

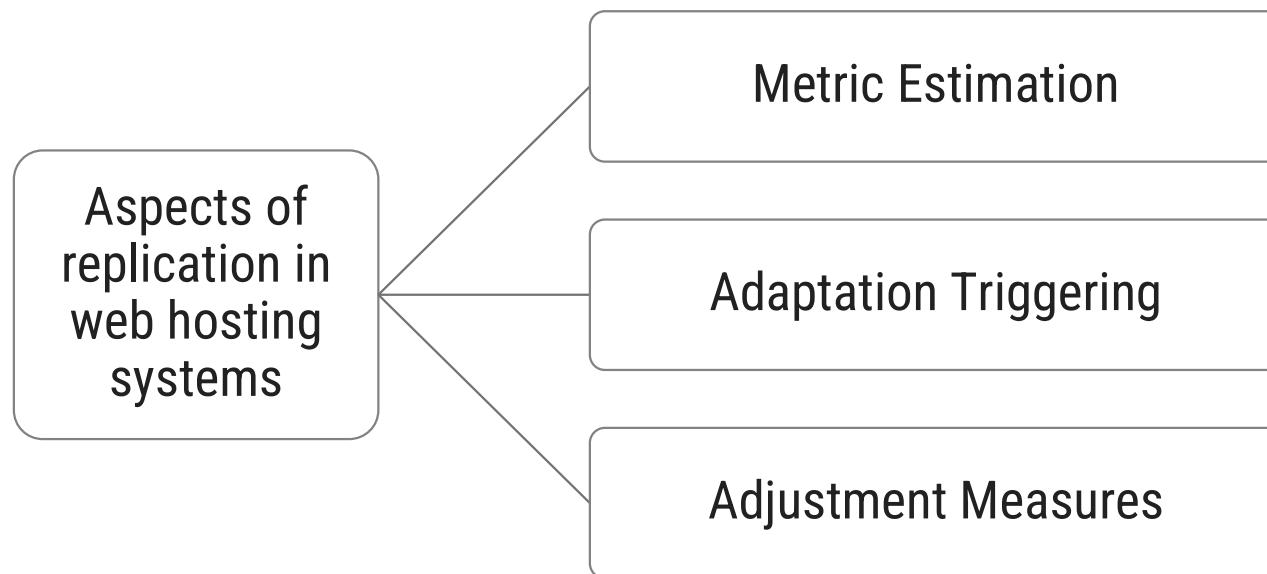
- As an alternative to building hierarchical caches, one can also organize caches for cooperative deployment.



The Principle of cooperative caching

Replication for Web Hosting Systems

- ▶ There are essentially three different kinds of aspects related to replication in web hosting systems.



Metric Estimation

- ▶ **Latency metrics:** By which the time is measured for an action to take place for example: Fetching a document.
- ▶ **Spatial metrics:** It mainly consists of measuring the distance between nodes in terms of the number of network-level routing hops or hops between autonomous systems.
- ▶ **Network usage metrics:** It computes consumed bandwidth in terms of the number of bytes to transfer.
- ▶ **Consistency metrics:** It tell us to what extent a replica is deviating from its master copy.
- ▶ **Financial metrics:** It is closely related to the actual infrastructure of the Internet.

Adaptation Triggering

- ▶ Important question that needs to be addressed is when and how adaptations are to be triggered.
- ▶ A simple model is to periodically estimate metrics and subsequently take measures as needed.
- ▶ Special processes located at the servers collect information and periodically check for changes.

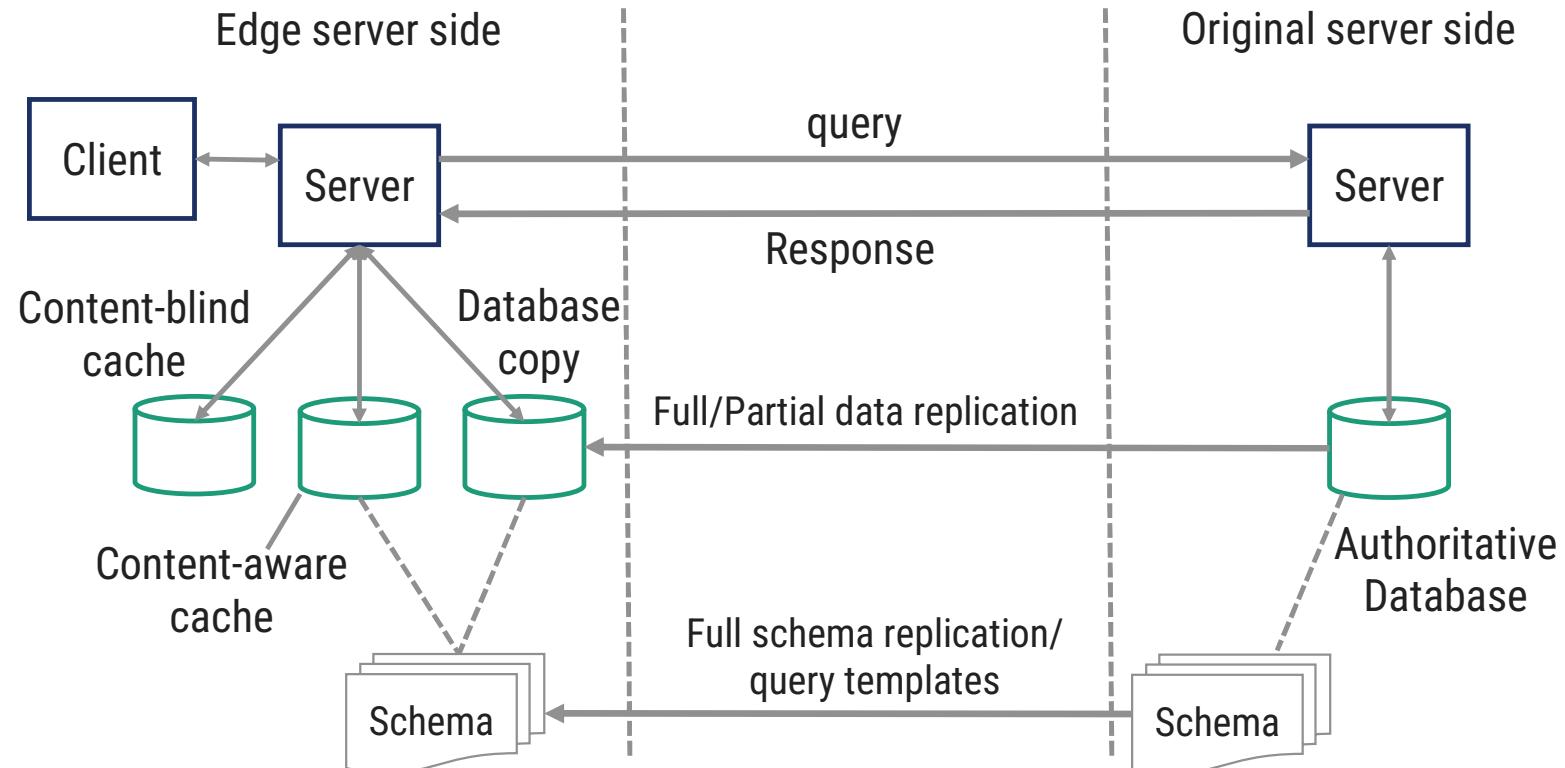
Adjustment Measures

- ▶ There are essentially only three measures that can be taken to change the behavior of a Web hosting service:
 1. Changing the placement of replicas
 2. Changing consistency enforcement
 3. Deciding on how and when to redirect client requests

Replication of Web Applications

- ▶ It is complicated to improve performance of Web applications through caching and replication.
- ▶ To improve performance, we can apply full replication of the data stored at the origin server.
- ▶ This scheme works well whenever the update ratio is low and when queries require an extensive database search.
- ▶ Replicating for performance will fail when update ratio is high.
- ▶ Alternative is partial replication in which only a subset of the data is stored at the edge server.
- ▶ The problem with partial replication is that it may be very difficult to manually decide which data is needed at the edge server.

Replication of Web Applications



Alternatives for caching and replication with Web applications.

Categories of Distributed System

Distributed Coordination-Based Systems

Distributed Coordination-Based Systems

- ▶ Instead of concentrating on the transparent distribution of components, emphasis lies on the coordination of activities between those components.
- ▶ Key to the approach followed in coordination-based systems is the clean separation between computation and coordination.
- ▶ Essence: We are trying to separate computation from coordination; coordination deals with all aspects of communication between processes, as well as their cooperation.
- ▶ Make a distinction between:
 1. **Temporal coupling:** Are cooperating/communicating processes alive at the same time?
 2. **Referential coupling:** Do cooperating/communicating processes know each other explicitly?

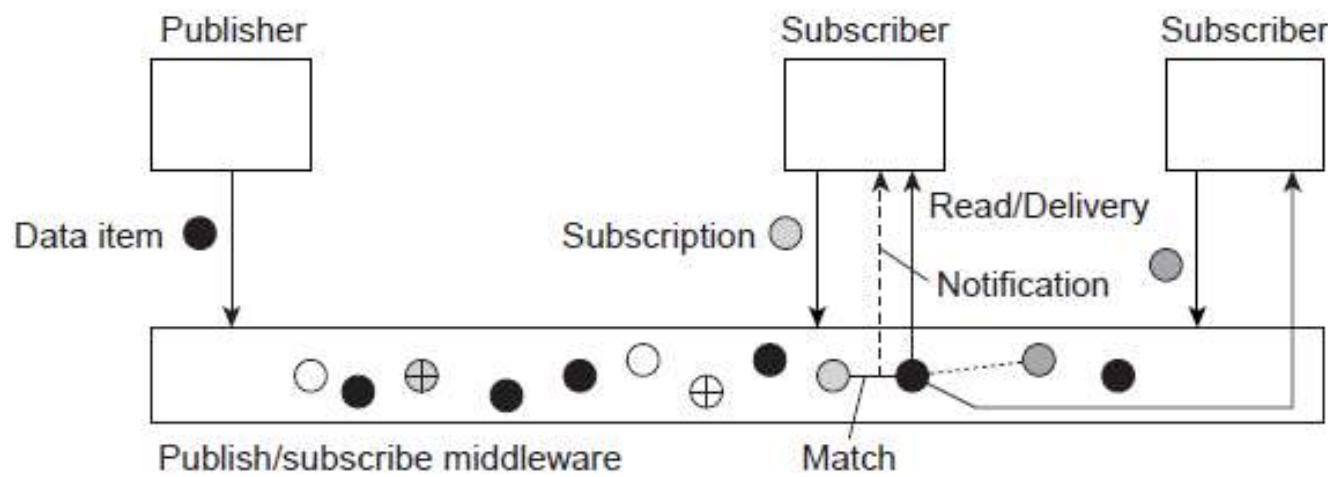
Distributed Coordination-Based Systems

► Coordination models

		Temporal	
		Coupled	Decoupled
Referential	Coupled	Direct	Mailbox
	Decoupled	Meeting oriented	Generative communication

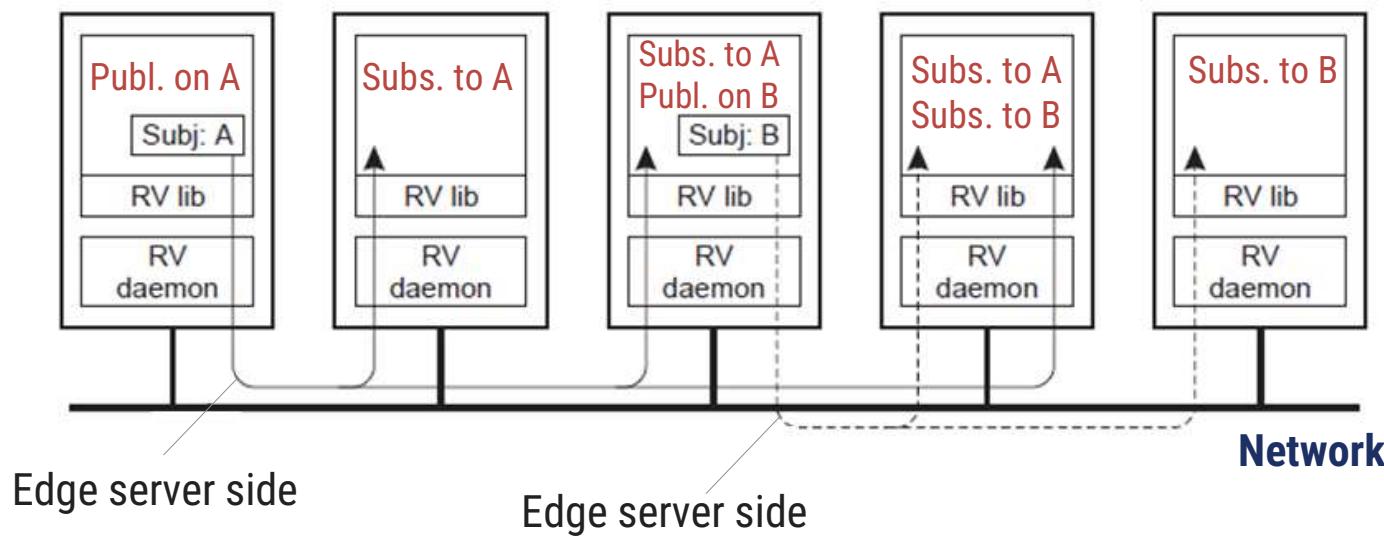
Architectures: Overview

- ▶ A data item is described by means of attributes.
- ▶ When made available, it is said to be published.
- ▶ A process interested in reading an item, must provide a subscription: a description of the items it wants.
- ▶ Middleware must match published items and subscriptions.



TIB/Rendezvous: Overview

- ▶ Coordination model: makes use of subject-based addressing, leading to what is known as a publish-subscribe architecture
- ▶ Receiving a message on subject X is possible only if the receiver had subscribed to X
- ▶ Publishing a message on subject X, means that the message is sent to all (currently running) subscribers to X.

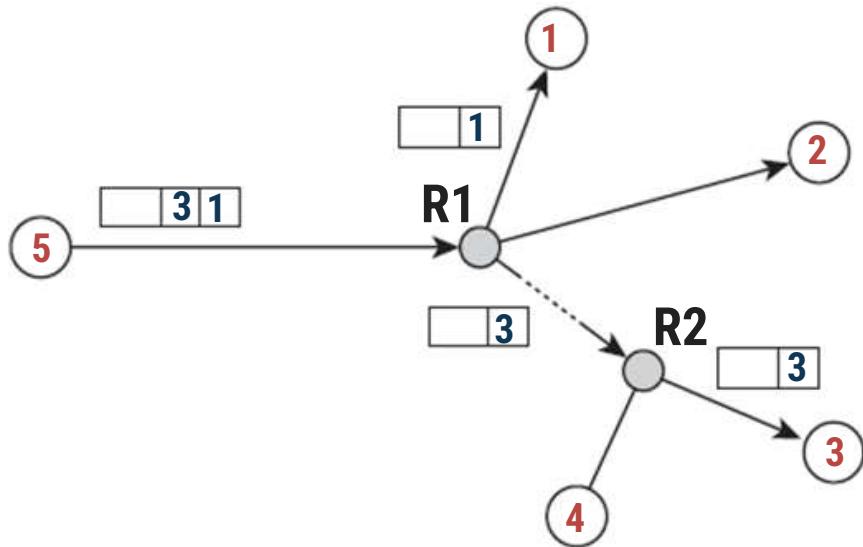


Communication

- ▶ Content-based routing
- ▶ **Observation:** When a coordination-based system is built across a wide-area network, we need an efficient routing mechanism (centralized solutions won't do).
- ▶ Solution
 - **Naive:** Broadcast subscriptions to all nodes in the system and let servers prepend destination address when data item is published
 - **Refinement:** Forward subscriptions to all routers and let them compute and install filters.

Content-based routing: naive solution

- ▶ The other extreme solution is to let every server broadcast its subscriptions to all other servers. As a result, every server will be able to compile a list of (subject, destination) pairs.
- ▶ Then, whenever an application submits a message on subjects, its associated server prepends the destination servers to that message.
- ▶ When the message reaches a router, the latter can use the list to decide on the paths that the message should follow, as shown in Fig.



Interface	Filter
To node 3	$a \in [0,3]$
To node 4	$a \in [2,5]$
Toward router R1	(unspecified)

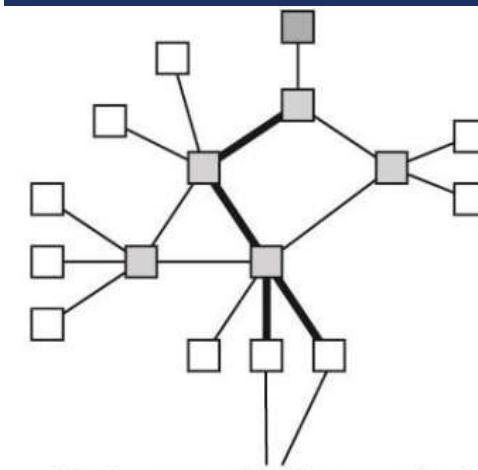
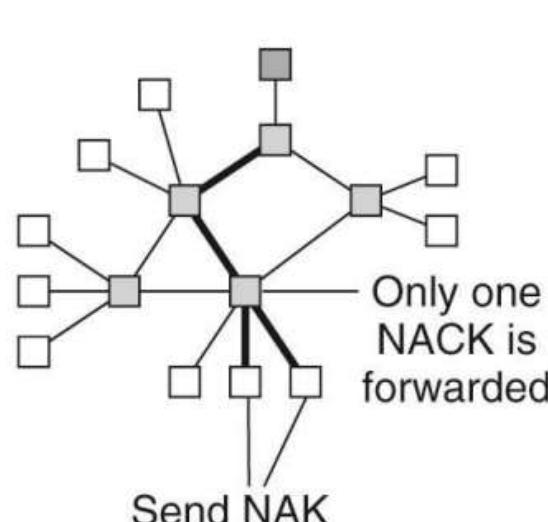
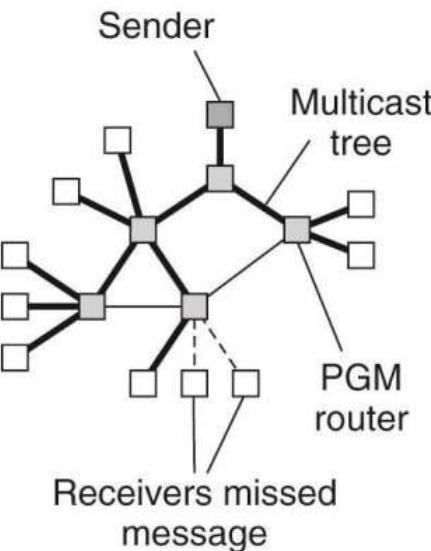
Fault Tolerance in TIB/Rendezvous

- ▶ Problem: TIB/RV relies on multicasting for publishing messages to all subscribers. This mechanism needs to be extended to wide-area networks and requires reliable multicasting.
- ▶ Solution: Pragmatic General Multicast (PGM): a NACK-based scheme in which receivers tell the sender that they are missing something (no hard guarantees).

A message is sent along a multicast tree

A router will pass only a single NAK for each message.

A message is retransmitted only to receivers that have asked for it



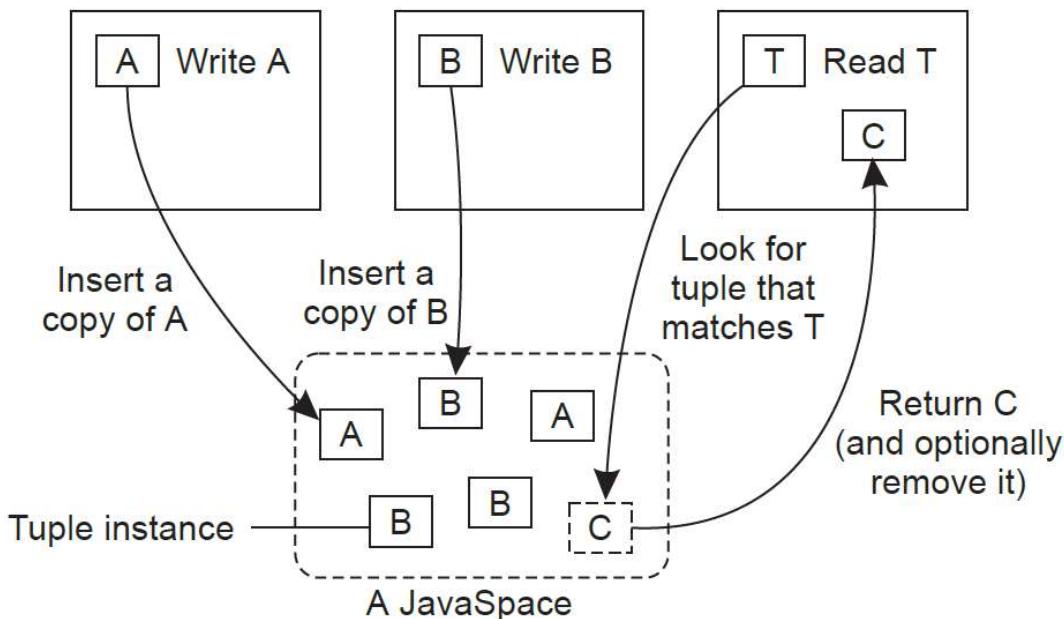
Jini: Overview

- ▶ Coordination-based system from Sun Microsystems
- ▶ Written in Java: one language everywhere
- ▶ Uses RMI and Java Object Serialization to enable Java objects to move around the network
- ▶ Offers network plug and play of services (Java objects)
- ▶ Services may come and go without administration and reconfiguration
- ▶ Federation, not central control
- ▶ Programming interfaces designed for robustness

Jini: Main Components

- ▶ **Service**: an entity that another program, service or user can use. It can be a piece of computation, a hardware device or software.
- ▶ **Client**: a Jini device or component that becomes a member of the federation in order to use a Jini service.
- ▶ **Lookup Service**: keeps track of the services offered in the federation.
 - Repository of available services.
 - Stores each service as Java objects.
 - Clients download services on demand.

Jini: Javaspaces



- ▶ **Write:** A copy of a tuple (tuple instance) is stored in a JavaSpace
- ▶ **Read:** A template is compared to tuple instances; the first match returns a tuple instance
- ▶ **Take:** A template is compared to tuple instances; the first match returns a tuple instance and removes the matching instance from the JavaSpace