Distributed Systems

Lecturer: Dr. Nadia Tarik Saleh

Email: nadia_tarik@uomosul.edu.iq

Mosul University

College of Computer Science and Mathematics

Dept. of Computer Science

2023-2024

Distributed Systems

Syllabus

- Introduction
  - Definition of a distributed system
  - Goals
  - Hardware concepts
  - The client server model
- Communication
  - Layered protocols
  - Remote procedure call
  - Remote object invocation
  - Message oriented communication
  - Steam oriented communication
- Processes
  - Threads
  - Clients
  - Servers
  - Code migration
  - Software agents
- Naming
- Synchronization
- Consistency & Replication

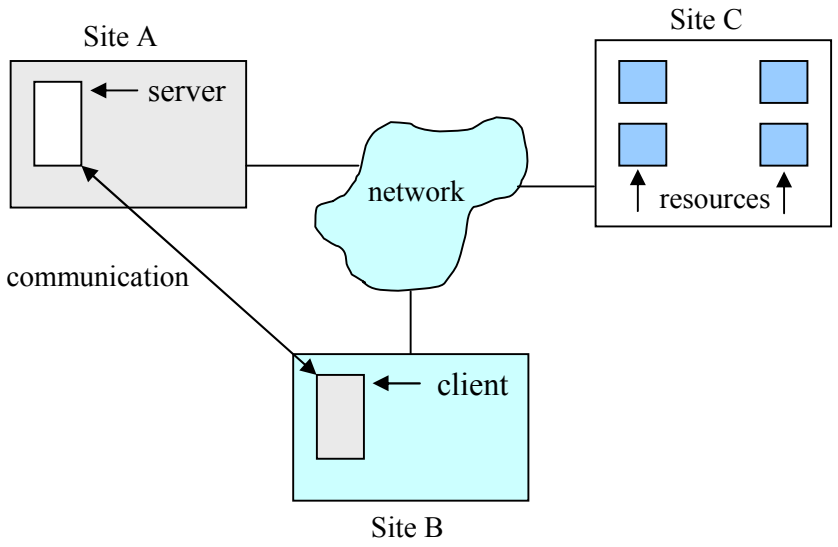
References

- Andrew S. Tanenbaum and Maarten Van Steen, *Distributed Systems: Principals and Paradigms*, 2nd ed., Upper Saddle River, New Jersey, USA: Prentice Hall, 2007.
- Marten Van Steen and Andrew S. Tanenbaum, *Distributed System*, 3rd Edition 2017.
- George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair, *Distributed Systems: Concepts and Design*, 5th ed., Addison-Wesley, 2012.
- Andrew S. Tanenbaum, *Modern Operating Systems*, 3rd Ed., USA: Prentice-Hall, Inc., 2008.
- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed., USA: John Wiley & Sons, Inc., 2013.

Distributed System (DS)

A distributed system is a collection of independent computers that appears to the users as a single coherent system.

The computers in a distributed system may vary in size and function. They may include microprocessors, personal computers, and large general-purpose computer systems. A general structure of a distributed system is shown in Figure below.



General Structure of a Distributed System

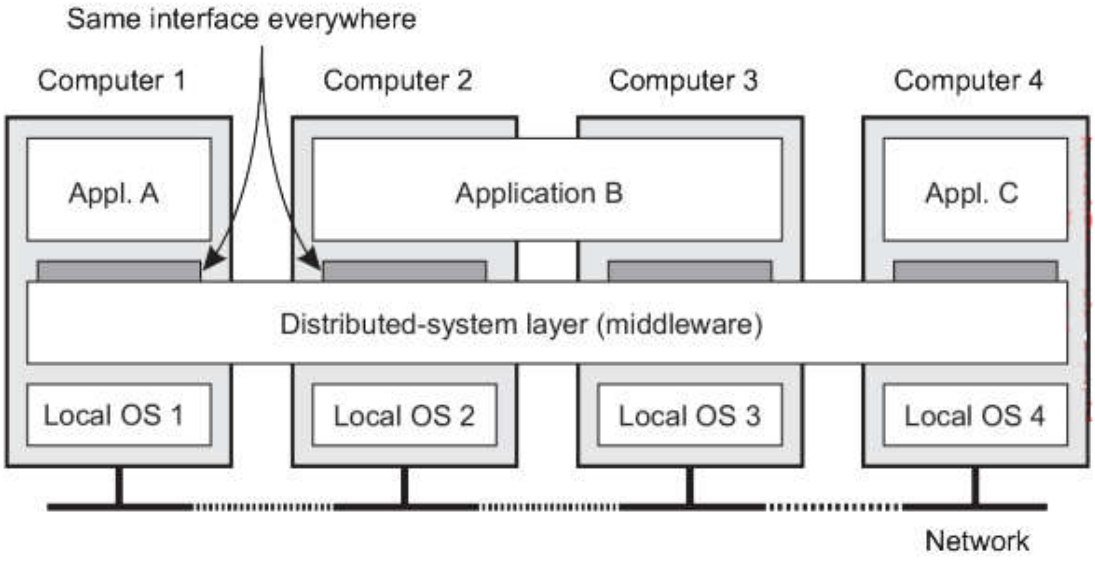
❖ Characteristics of Distributed Systems

- The distributed system consists of components (i.e., computers) that are independent.
- The users (either people or programs) are dealing with a single system.
- This components need to collaborate.
- Computers types could range from high-performance mainframe computers to small nodes in sensor networks.
- No assumptions are made on the way that computers are interconnected.
- The differences between the various computers and the ways in which they communicate are mostly hidden from users.

- Distributed systems should also be relatively easy to expand or scale.
- A distributed system will normally be continuously available, although perhaps some parts may be temporarily out of control.
- Users and applications should not notice that parts are being replaced, fixed, or that new parts are added.

❖ The Organization of Distributed System

- The distributed systems are often organized as *a layer of software -that is, logically placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems and basic communication networks.* Such distributed system is called ***Middleware***.
- The middleware layer extends over multiple machines.
- The distributed system provides the means for components of a single application to be distributed over number of computers and communicate with each other. Also, DS lets different applications to communicate.



The Organization of Distributed System

❖ Goals

1. **Making Resource Accessible**

The main goal of DS is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way.

Resources can be printers, computers, storage facilities, data, files, Web pages, and networks.

❖ Reasons to Share Resources

- **Economics:** it is cheaper to share costly resources such as printers, supercomputers, and others.
- **Collaboration:** connecting users and resources makes it easier to collaborate and exchange information, like when using the Internet.

2. **Distribution Transparency**

- Transparent system is a distributed system that is able to present itself to users and applications as if it was only a single computer system.
- This means hiding the fact that its processes and resources are physically distributed across multiple computers.

The concept of transparency can be applied to the following aspects of a distributed system:

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

Different forms of transparency in a distributed system

3. Openness

- An **open distributed system** is a system that offers components that can be easily be used by, or integrated into other systems. At the same time, an open DS itself will often consist of components that originate from elsewhere.
- To be open means that component should be constrained with standard rule that describe the syntax and semantics of those components. For example, in computer networks, standard rules control the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols.

An open distributed system should:

- **Portable:** Portability describes the ability of one distributed system application that can be executed without modification, on a different distributed system.
- **Flexible:** the system is organized as a collection of relatively small and easily replaceable or adaptable components. Such system is called **Extensible System**, where it is easy to add parts that run on a different operating system or even to replace an entire file system.

4. Scalability

Scalability of a system can be measured in three different dimensions:

1. Size: meaning that we can easily add more users and resources to the system.

2. Geographical: in which the users and resources may lie far apart.

3. Administrative: meaning that it can still be easy to manage the system even if it spans many independent administrative organizations.

❖ Scalability Limitations

1. **Centralized Services:** many services are centralized where they are implemented by means of only a single server running on a specific machine in the distributed system.
2. **Centralized Data:** having a single database would certainly saturate all the communication lines into and out of it.
3. **Centralized Algorithm:** a large number of messages have to be routed over many lines, and then run an algorithm to compute all the optimal routes.

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Examples of Scalability Limitations

- **Decentralized Algorithms:** only these types of algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists. It is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization.

❖ Scaling Techniques

- **Hiding communication latencies**: is important to achieve geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote service requests as much as possible.
- **Partitioning and Distribution**: involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system.
- **Replication** of the components across a distributed system. Replication increases availability and helps to balance the load between components leading to better performance.

❖ Pitfalls

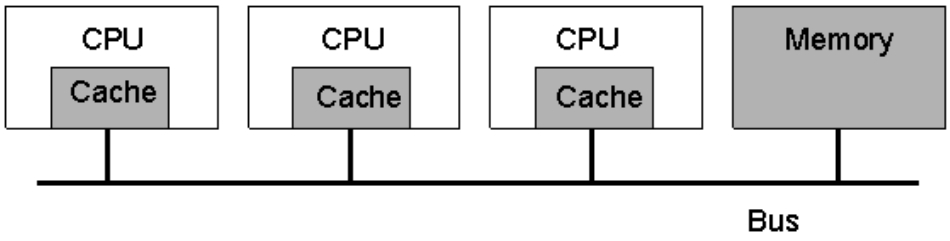
The following false assumptions are made by every developer when developing a distributed application for the first time:

- The network is reliable.
- The network is secure.
- The network is homogeneous.
- The topology does not change.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

❖ Hardware Concepts

Even though all distributed systems consist of multiple CPUs, there are several different ways of organizing hardware, in terms of how they are interconnected and how they communicate. Computers are divided into two groups:

1. **Multiprocessors (Shared-memory):** computers that are supplied with several processors that share one or more modules of memory (RAM). The processors may also have their own private memory.



Bus-based Multiprocessor (shared memory)

2. **Multicomputer: (Distributed-memory multiprocessors)**: the processors do not share memory but are connected by a very high speed network. These systems can have greater numbers of processors than shared-memory multiprocessors.

- Distributed systems can be further classified into two types:
  1. **Homogeneous Systems:** where connected computers are the same with identical processors. All computers have the same operating system, and are all connected through the same network.
  2. **Heterogeneous Systems:** may contain different independent computers, each has its own operating system and machine architecture and is connected through different networks.

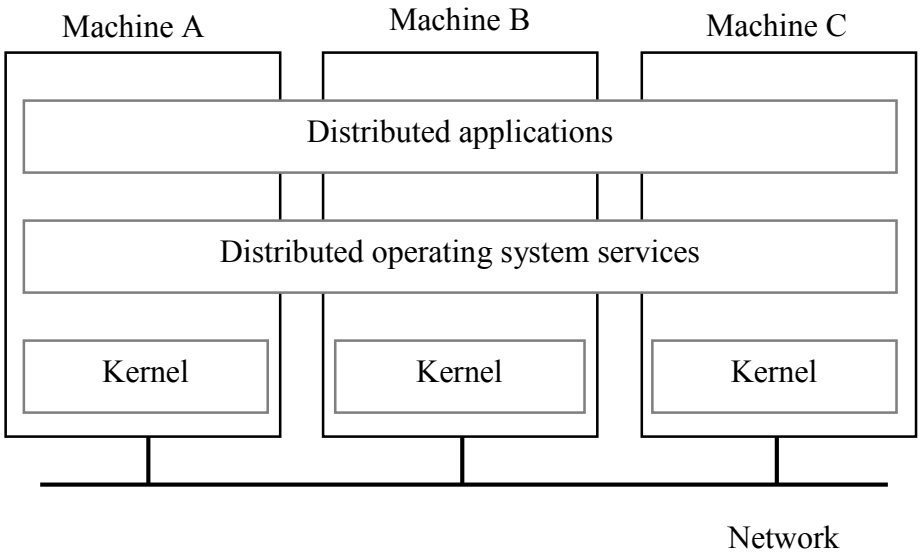
❖ Operating Systems for Distributed Systems

Operating systems for distributed computers is classified into:

1. **Distributed Operating Systems:** The functionality of distributed operating systems is essentially the same as that of traditional operating systems for uniprocessor systems except that they handle multiple CPUs.

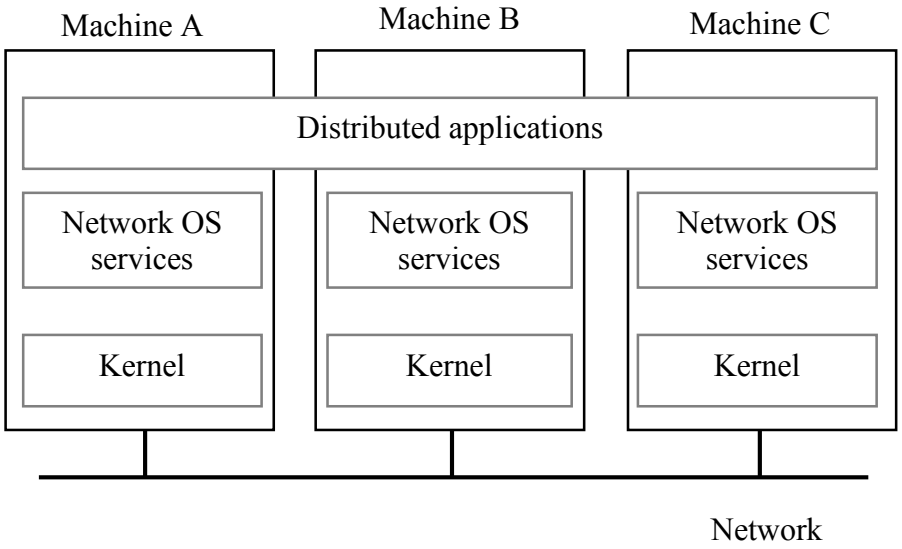
   ➢ There are two types of distributed operating systems:

- *A multiprocessor operating system:* manages the resources of a multiprocessor. It supports multiple processors and could access shared memory.
- *A multicomputer operating system:* is developed for homogeneous multicomputer. Communication is performed through message passing between nodes.



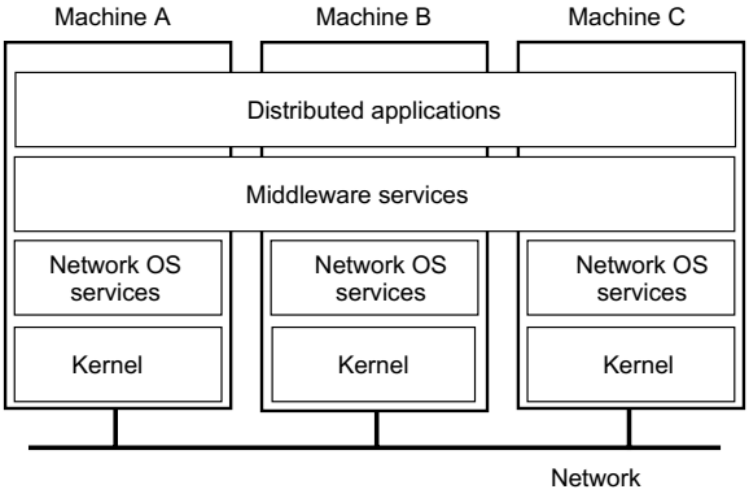
Multicomputer Operating System

2. **Network Operating Systems:** network operating systems do not assume that the underlying hardware is homogeneous. The machines and their operating systems may be different, but they are all connected to each other in a computer network.



Network Operating System

3. **Middleware:** an additional layer of software that is used in network operating systems to hide the heterogeneity of the collection of underlying platforms and to improve distribution transparency.



General Structure of a Distributed System as Middleware

❖ A Comparison between Systems

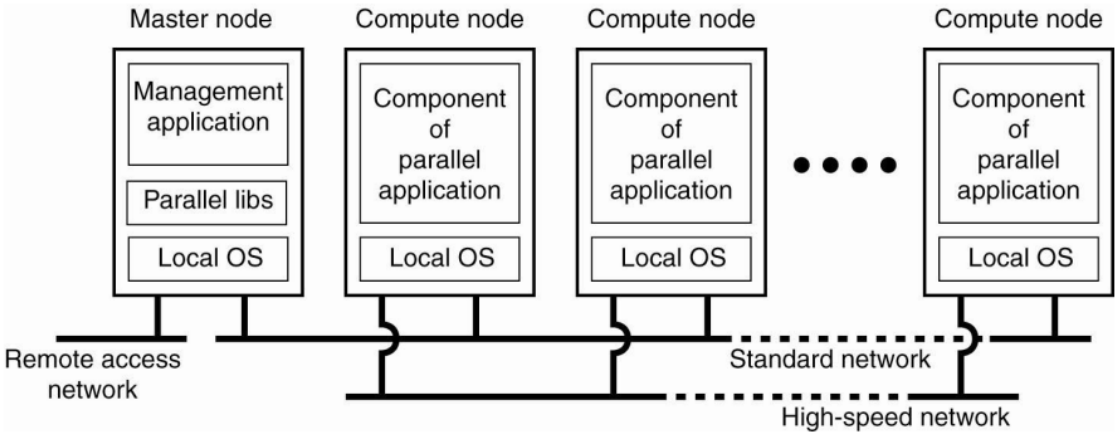
A brief comparison between distributed operating systems, network operating systems, and (middleware-based) distributed systems is given in table below:

Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

❖ Common Types of Distributed Systems

1. ***Clustered Computing System:*** is a collection of relatively simple computers (Homogeneous resources) connected in a high-speed network. This type of DS has the following features:

- The underlying hardware consists of a collection of similar workstations or PCs;
- Closely connected by means of a same high-speed local-area network;
- Each node runs the same operating system.
- Cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines.



Cluster Computing System

The cluster consists of a collection of ***compute*** *nodes* that are controlled and accessed by means of a single ***master*** *node*.

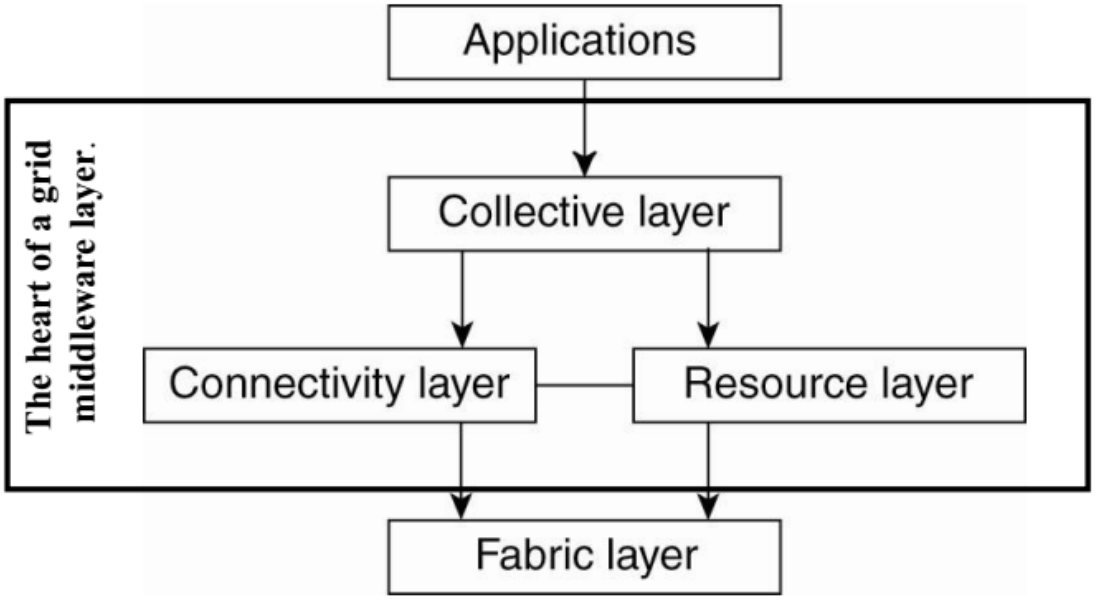
- The *master* node manages the cluster and performs:

- the allocation of nodes to a particular parallel program;
- maintains a queue of submitted jobs;
- provides an interface for the users of the system;
- runs the middleware needed for the execution of programs.

- The *compute* nodes often need a standard operating system and execute a particular parallel program.

2. ***Grid Computing System:*** a system in which the computers from different organizations (Heterogeneous resources) are brought together to allow the collaboration of a group of people or institutions.

- In this system, there is no assumption about the type of hardware, operating systems and networks.
- The resources consist of *Compute servers*, *storage facilities*, *databases* and *special networked devices*.
- The software for Grid computing providing access to resources from different administrative domains, and to only those users belongs to a specific organization.



The Architecture of Grid Computing

The architecture of the Grid is often described in terms of "**layers**", each providing a *specific function*. In general, the higher layers are focused on the user, whereas the lower layers are more focused on computers and networks (hardware).

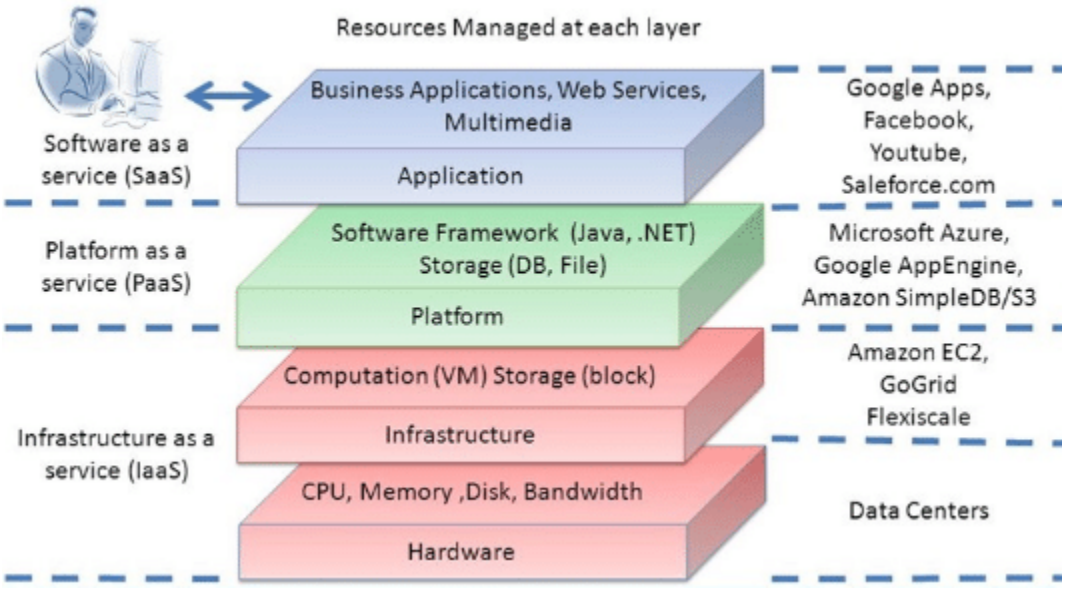
- *The architecture consists of four layers:*

1. The lowest ***fabric layer*** provides interfaces to local resources (all the physical infrastructure of the Grid, including computers, storage media, and the communication network).
2. The ***connectivity layer*** consists of communication protocols for supporting grid transactions between different computers and other resources on the Grid. For example, protocols are needed to transfer data between resources, or to access a resource from a remote location.
3. The ***resource layer*** is responsible for managing a single resource. It offers a set of protocols and APIs (Application Programming Interface) for efficient coordination and sharing of individual resource.

4. The ***collective layer*** deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources.
5. The ***application layer*** consists of all different user applications (science, engineering, business, financial) and make use of the grid computing environment. This is the layer that users of the grid will "see".

3. Cloud Computing: is the delivery of different computing services through the internet. These resources include *data storage*, *servers*, *databases*, *networking*, and *software*.

- The companies that offer the computing services are called ***cloud providers*** and the charges for cloud computing services are based on usage.
- Cloud computing enables companies and users to consume a compute resource, such as a *virtual machine (VM)*, *storage* or *an application*, as a utility —just like electricity— rather than having to build and maintain computing infrastructures in house.



Organization of Cloud Computing

- *Clouds are organized into four layers:*

1- **Hardware:** the lowest layer manages the necessary hardware: processors, routers, power and cooling system. It contains the resources that customers cannot see actually.
2- **Infrastructure:** It deploys virtualization techniques to provide customers an infrastructure consisting of virtual storage and computing resources.
3- **Platform:** provides the services and means to easily develop and deploy applications that needed to run in the cloud. These platforms are installed in the cloud.
4- **Application:** actual applications are offered to customers for use and customization. These applications are executed in the cloud, such as office suits (text processors, presentation applications).

- Cloud computing providers offer these layers to the users through various interfaces (command line, programming interfaces (API) and web interfaces), which results in three types of services:

1. ***Infrastructure-as-a-Service (IaaS)***: covering the hardware and infrastructure layers.
2. ***Platform-as-a-Service (PaaS)***: covering the platform layer.
3. ***Software-as-a-Service (SaaS)***: covering the application layer.

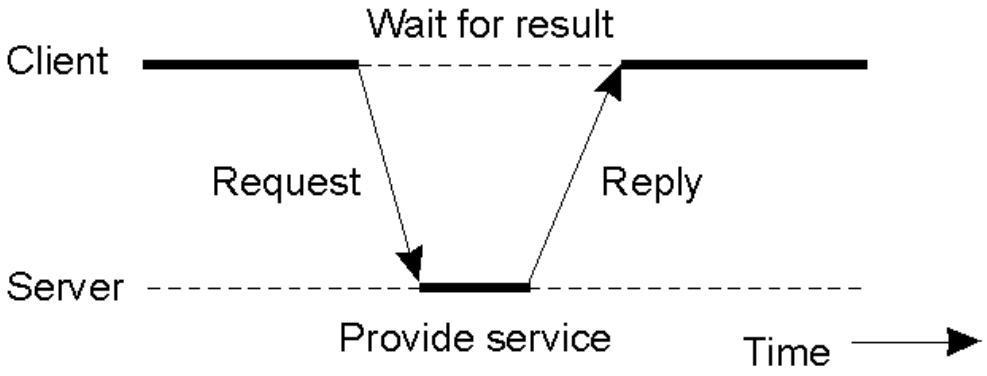
❖ Client-Server Model

The most commonly applied model for distributed system architecture.

a) Simple Client-Server Architecture

Processes in a distributed system are divided into two groups:

1. ***Server***: is a process implementing a specific service and containing the data, e.g.: a file system service or database service.
2. ***Client***: is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply.



The communication between the client and server is implemented as follows:

1. The client requests a service by simply packaging a message for the server, identifying the service it wants with the input data.
2. The message is sent to the server, which always is waiting for the incoming request.
3. Starting the process of the received message by the server.
4. Packaging the results in a reply message to be sent to the client.

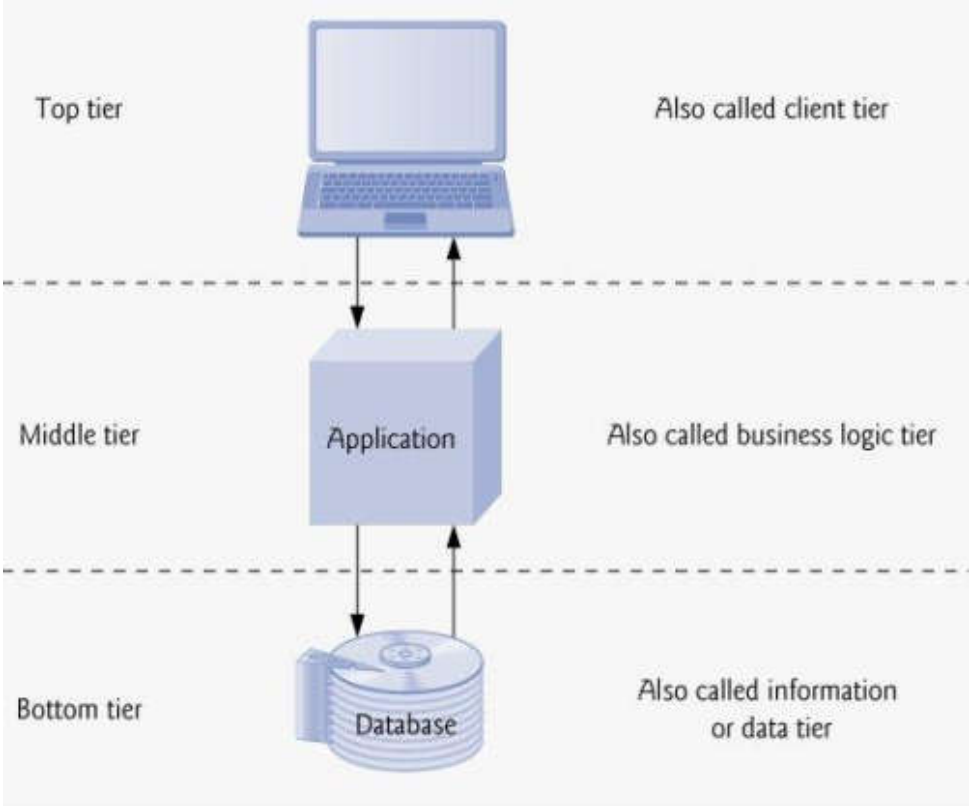
b) Multi-tiered Architecture

In this architecture the client-server system is distributed across several machines and the application is distributed into three layers:

(1) User interface layer.

(2) Processing layer.

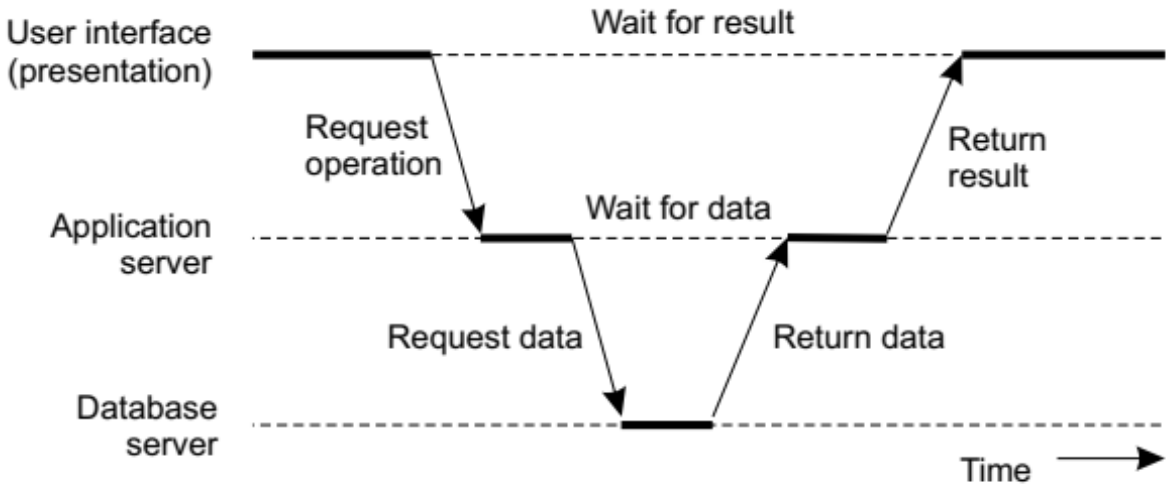
(3) Data layer.



Client-Server Organization in Multi-tier Architecture

c) A Server Acts as a Client

In this architecture, a program that forms a part of the processing layer is executed in a separate machine (Database server) and the essential of the processing (Application server) resides in another machine.



Architecture

Distributed systems are often complex pieces of software of which the components are scattered across multiple machines. There are different ways on how to view the organization of a distributed system:

- The logical organization of the collection of software components (Software Architecture)
- The actual physical realization (underlying Platform).

The organization of distributed systems is mostly about the software components that forms the system. These ***software architectures*** tell us **how the various software components are to be organized and how they should interact.**

❖ *Architecture Styles*

The architectural style is formulated in terms of:

1. **Components**,
2. The way that components are **connected** to each other,
3. The **data exchanged** between components,
4. How these elements are jointly **configured** into a system.

- A ***component*** is a modular unit with an interface specifying the services it provides and requests. ***Components*** can be small (one object) or large (a library or complete application). Examples: Client, Server, Database,.. etc.
- A ***connector*** is a mechanism that communicates, coordinates or cooperates with components. A ***connector*** is a building block that enables interaction among components. It allows for the flow of control and data between components. ***Components*** may interact with each other using any of the following ***connectors:*** Client/Server Middleware, Message Passing, Streaming Data, Shared Variables, Remote Procedure Calls, TCP/IP Ports, HTTP,.. etc.

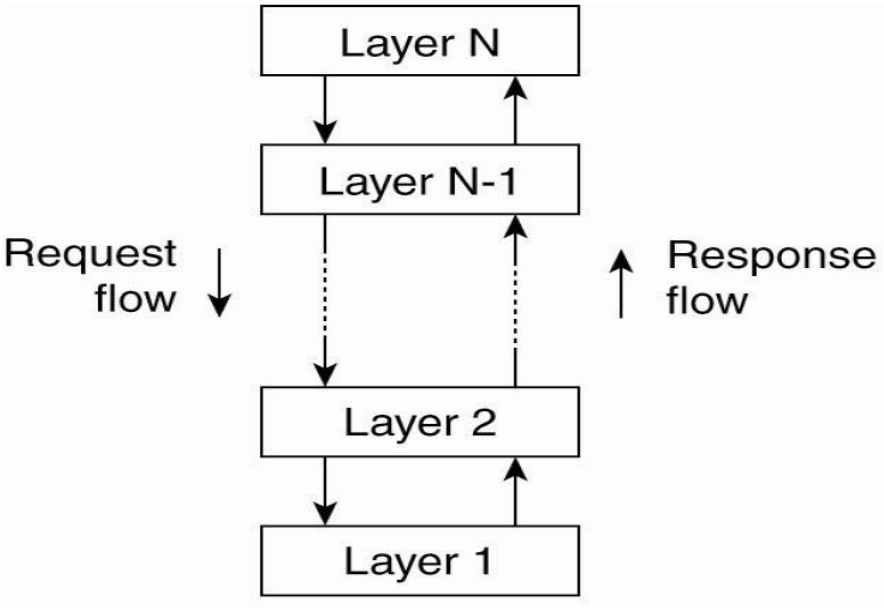
The most important architectural styles for distributed systems are:

- Layered architectures
- Object-based architectures
- Resource-centered architectures (Resource-based architectures)
- Publish-Subscribe Architectures (Event-based Architectures)

1. Layered Style

The basic idea for the layered style is simple: **components are organized in a layered fashion where a component at layer L; is allowed to call components at the underlying layer Li, but not the other way around**.

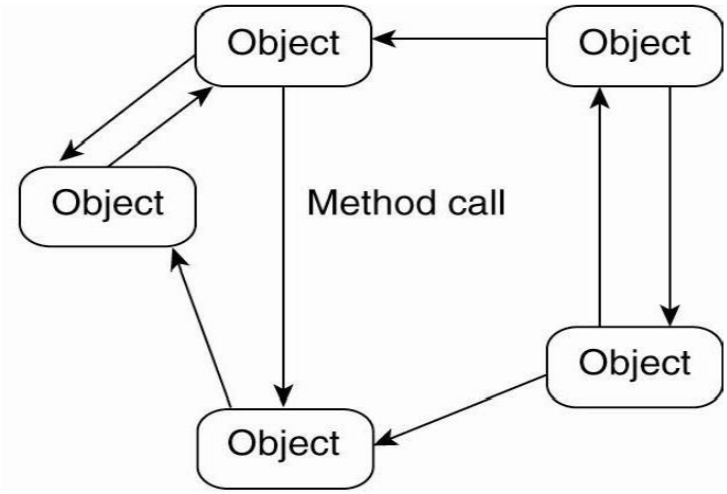
This model has been widely adopted by the **networking community**. A key observation is that control generally flows from layer to layer: requests go down the hierarchy whereas the results flow upward.



Layered Architectural Style

2. Object-based Architectures

In essence, **each object corresponds to what we have defined as a component, and these components are connected through a (remote) procedure call mechanism.** This software architecture matches the **client-server system architecture.**



An Object-based Architectural Style

3. Resource-centred Architectures

One can also view the distributed systems as a ***huge collection of resources that are individually managed by components.*** Resources may be added to be removed by (remote) applications, and can be retrieved or modified. This approach has been widely adopted for **the web and is known as Representational State Transfer (REST)**. There are four keys characteristics of what are known **RESTful** architectures:

1. Resources are identified through a single naming scheme (ID).
2. All services offer the same interface, consists of the most four operation shown in the table below.
3. Messages sent to or from a service are fully self-described.
4. After executing an operation at a service, that component forgets everything about the caller. This property is also referred to as **stateless execution.**

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

The Four Operations Available in RESTful Architecture

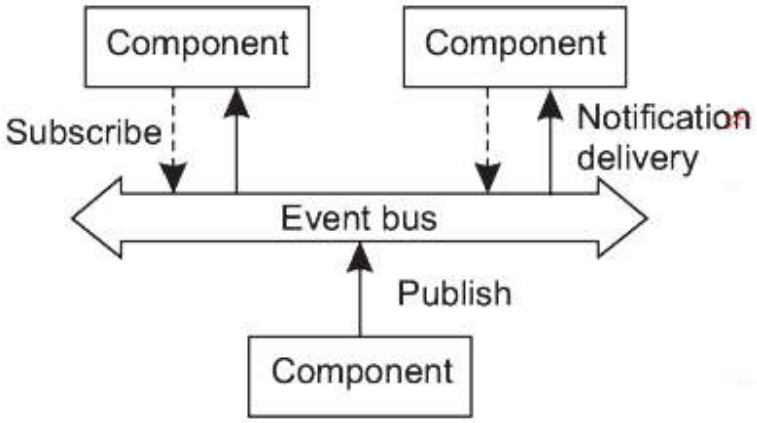
4. Publish-Subscribe Architectures

In this model, **coordination** covers the communication and cooperation between **processes**. It binds the activities performed by **processes**.

There are two types of coordination:

1. **Direct coordination**: a process can communicate only if it knows the name of the identifier of the other process it wants to exchange information with and both processes have to be up and running, e.g., talking over cell phones.
2. **Mailbox coordination**: there is no need for two communicating processes to be executing at the same time in order for communication to take place. Instead, communications takes place by putting messages in a mail box.

The combination of these two types forms the group of model called **event-based coordination**. In this model the process can **publish** a **notification** describing the occurrence of an event. Processes may **subscribe** to a specific kind of notification.



The event-based architectural style

❖ **System Architectures (Hardware Architectures)**

Important styles of hardware architecture for distributed systems:

1. Centralized architectures:

a) Basic client-server.

b) Application layering.

c) Multi-tiered architectures.

2. Decentralized architectures:

a) Structured Peer to Peer (P2P).

b) Unstructured P2P.

c) Hierarchical P2P.

3. Hybrid architectures.

1. Centralized architectures:

b) Application Layering

Three levels can be distinguished between a client and a server:

1) The user-interface level,

2) The processing level,

3) The data level.

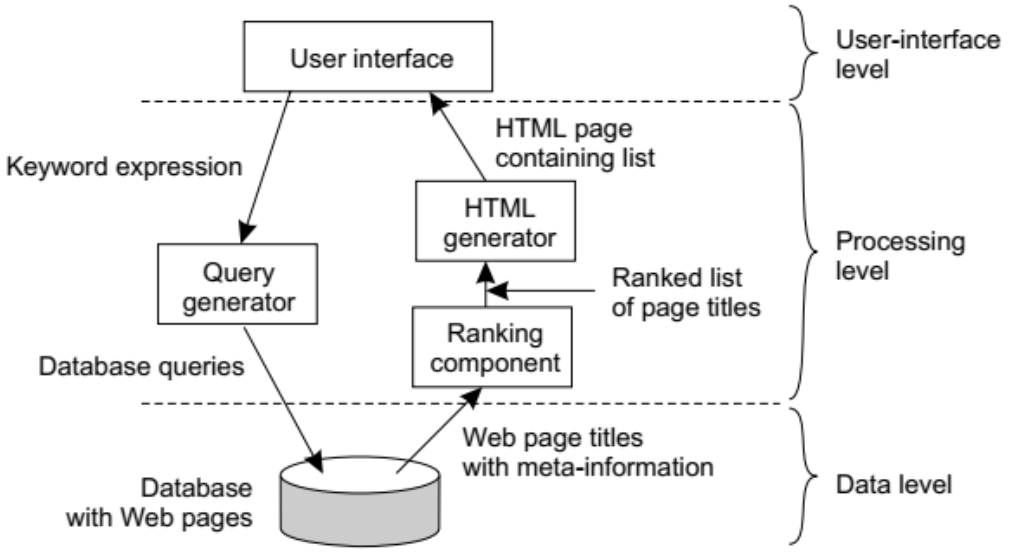
Many client server applications can be constructed from three different pieces: a part that handles interaction with a user, a part that operates on a database or file system, and a middle part that generally contains the core functionality of an application. This middle part is logically placed at the processing level.

1- **The user-interface level** contains all that is necessary to directly interface with the user, such as display management. Clients typically implement the user-interface level. This level consists of the programs that allow end users to interact with applications.

2- **The processing level** typically contains the applications.

3- **The data level** manages the actual data that is being acted on.

E.g.: consider an Internet search engine. The user interface of a search engine is very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been pre-fetched and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list, and transforms that list into a series of HTML pages. Within the client-server model, this information retrieval part is typically placed at the processing level.



General Organization of an Internet Search Engine into Three Different Layers

2. Decentralized Architectures:

a) Structured Peer-to-Peer Architectures

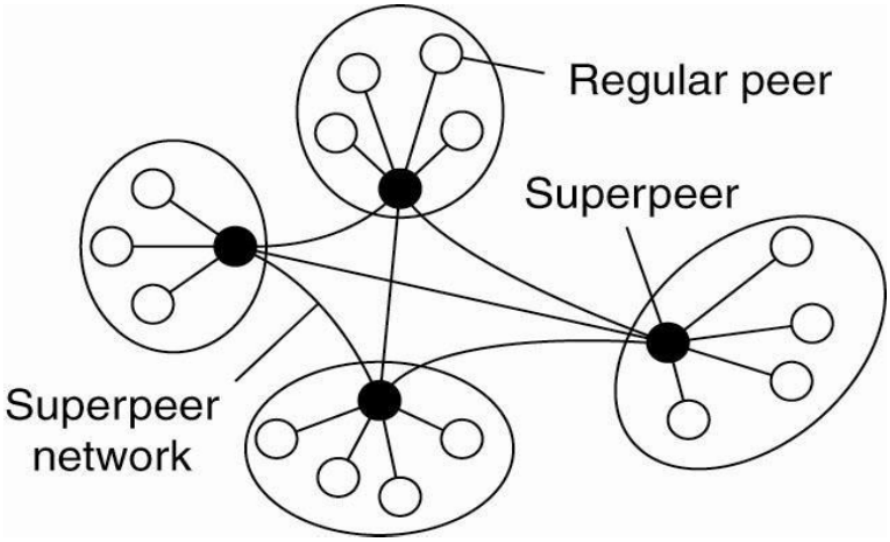
The nodes (i.e. processes) that constitute a peer-to-peer system are all equal. Each process will act as a client and a server. The nodes are organized as a specific topology: a ring, a binary tree, a grid, etc.

b) Unstructured Peer-to- Peer Architectures

Each node maintains a **list of neighbours**; a node generally changes its local list continuously. For example, a node may discover that a neighbour is no longer responsive and that it needs to be replaced.

c) Hierarchical Organized peer-to-peer networks (Super-peers)

In unstructured p2p systems, locating relevant data items can become problematic as the network grows. The reason for this problem is: **there is no deterministic way of routing a lookup request to a specific data item**. An alternative many peer-to-peer systems have proposed to make use of ***special nodes that maintain an index of data items***. Nodes such as those are generally referred to as ***super-peers***. ***Super-peers*** are often also organized in a peer-to-peer network, leading to a ***hierarchical organization***.

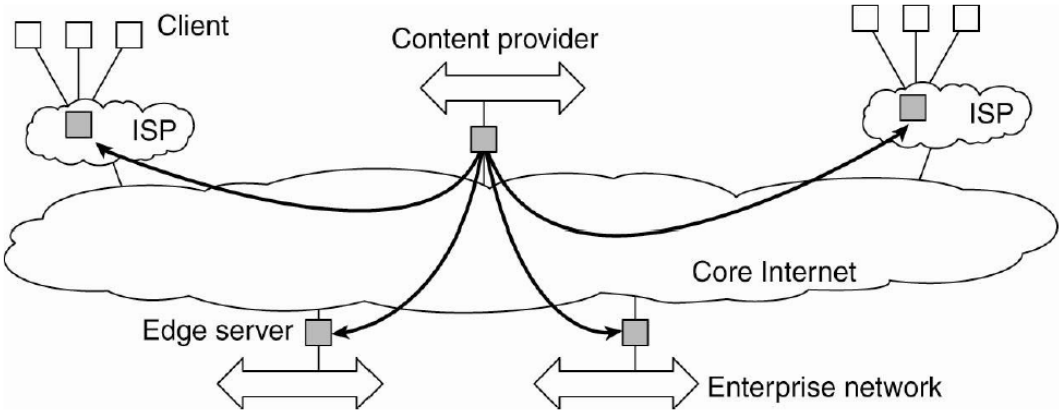


A Hierarchical Organization of Nodes into a Superpeer Network.

3. Hybrid Architectures

A) Edge-Server Systems

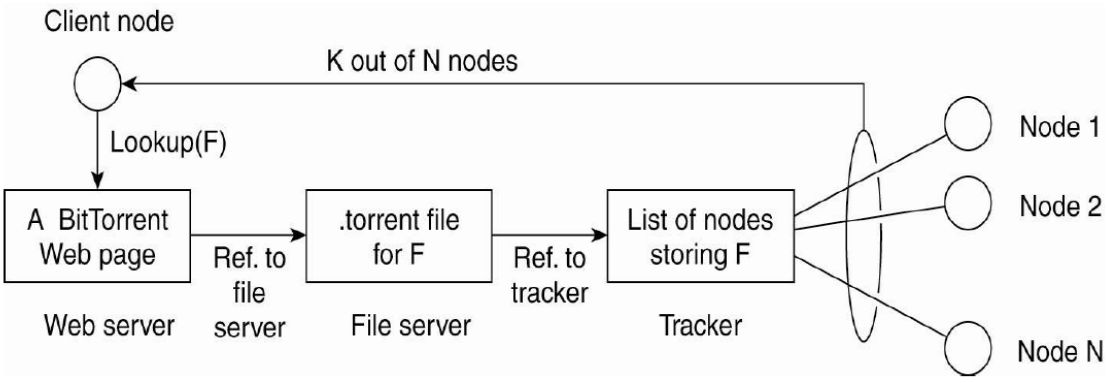
Edge-server systems are deployed on the Internet where servers are placed "at the edge" of the network, e.g., an **Internet Service Provider (ISP)**, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of Internet.



Viewing the Internet as Consisting of a Collection of Edge Servers

B) Collaborative Distributed Systems

In this architecture, when one node joins the system, it can use a fully decentralized scheme for collaboration. Let us consider the BitTorrent file-sharing system. BitTorrent is a peer-to-peer file downloading system. The basic idea is that when an end-user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file.

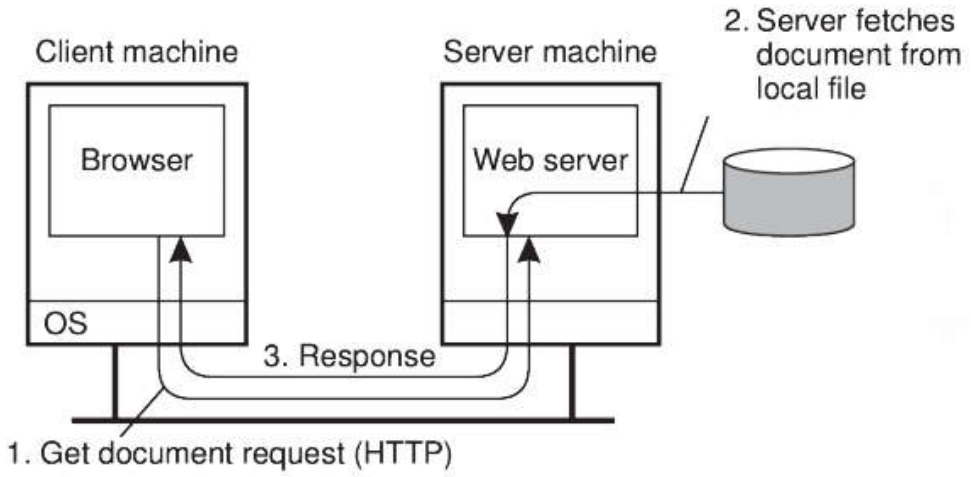


The Principal Working of BitTorrent

- ***Example Architecture: Simple Web-based systems***

Many Web-based systems still organized as relatively simple client-server architecture. The core of a Web site is formed by a process that has access to a local file system storing documents. 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 by embedding the **Domain Name System** (**DNS)** name of its associated server along with a file name by which the server can look up the document in its local file system. Furthermore, a URL specifies the application-level protocol for transferring the document across the network.

A client interacts with Web server through a browser, which is responsible properly displaying a document. Also, browser accept input from a user mostly by letting the user select a reference to another document, which it then subsequently fetches and displays. The communication between the browser and Web server is standardized: they both adhere to **Hyper Text Transfer Protocol (HTTP)**.



The Organization of a Traditional Web Site

Processes

❖ *Processes*

A process is often defined as a program in execution.

- To execute a program, an operating system creates a number of virtual processors, each one for running a different program.
- To keep track of these virtual processors, the operating system has a **process table**, containing entries to store:
  - CPU register values,
  - Memory maps (process address space),
  - Open files,
  - Accounting information: this involves the CPU usage, real time used, process numbers, and so on.
- Multiple processes may be concurrently sharing the same CPU and other hardware resources.
- A process consists of at least one **thread**.

❖ *Thread*

A **process** is executed on one of the Operating System virtual processor.

- To keep track of the processes, the OS creates entry for each process in the **Process Table** which called **Process Context** to store process related execution information.
- A **thread** is a basic unit of CPU utilization and a process may contain number of threads.
- In particular, a **Thread Context** often consists of nothing more than the CPU context, along with some other information for thread management.

❖ Thread Usage in Nondistributed Systems

The advantages of using multithreading:

1. The ***performance*** of a multithreaded application is better than that of its single-threaded.
2. Provide a convenient means of allowing ***blocking system calls*** without blocking the entire process in which the thread is running.
3. The possibility to exploit ***parallelism*** when executing the program on a multiprocessor computer system.
4. Multithreading is also useful in the context of ***large*** **applications**. Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process.

❖ Thread Implementation

Threads are often provided in the form of a **thread package** which contains:

(1) **Operations to create and destroy threads.**

(2) **Operations on synchronization variables**.

- There are basically two approaches to implement a thread package:
  - **User-level thread approach:** constructs a thread library that is executed entirely in user mode.
  - **Kernel-level thread approach:** is to have the kernel be aware of threads and schedule them.

❖ Threads in Distributed Systems

1- Multithreaded Clients:

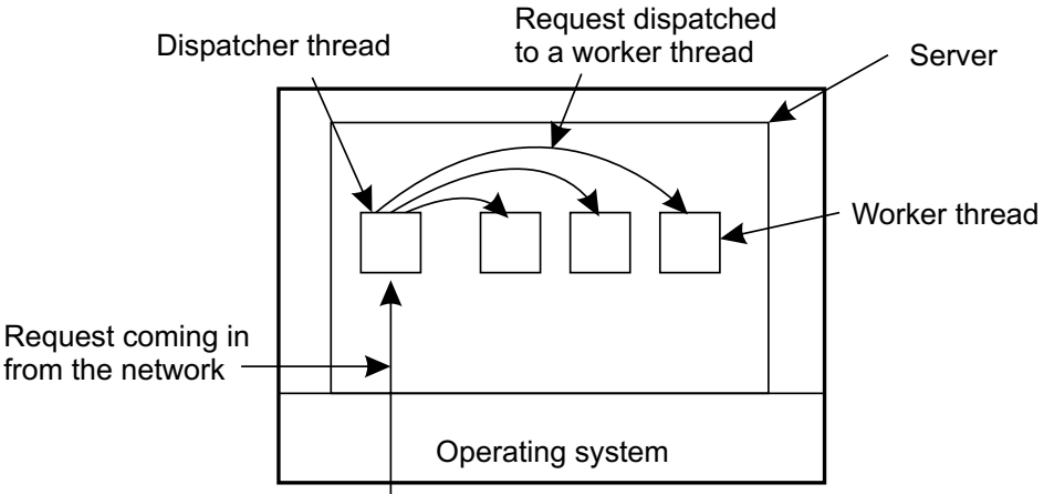
- An important property of threads is that they can provide a convenient means of **allowing blocking system calls without blocking the entire process in which the thread is running**.

- This property makes threads suitable to be used in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.
- Developing client application as a multithreaded **simplifies matters considerably**. e.g.: The Web browser is doing a number of tasks simultaneously.

2. Multithreading Server:

The main use of multithreading in distributed systems for server side:

- practice shows that multithreading ***simplifies server code*** considerably,
- makes it much easier to develop servers that exploit ***parallelism to get high performance***. e.g.: Multithreading File server in a **dispatcher/worker model**.



A Multithreaded Server Organized in a Dispatcher/Worker Model

Here one thread, the **dispatcher**, reads incoming requests for a file operation. The requests are sent by clients to this server. After examining the request, the server chooses an idle (i.e., blocked) **worker thread** and hands

it the request. The worker proceeds by performing a blocking **read** on the local file system, which may cause the **thread to be suspended** until the data are fetched from disk. If the thread is suspended, another thread is selected to be executed. For example, the **dispatcher** may be selected to acquire more work. Alternatively, another **worker thread** can be selected that is now ready to run.

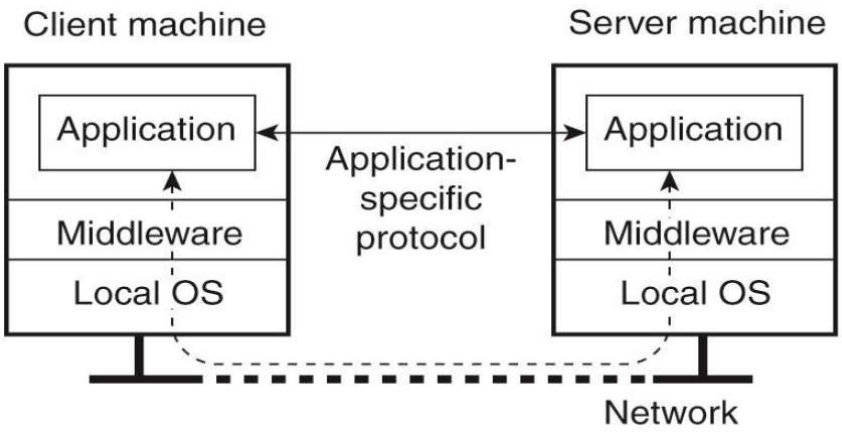
❖ *Client*

Networked user Interface: **A major task of client machines is to provide the means for users to interact with remote servers**. There are two ways to support this interaction:

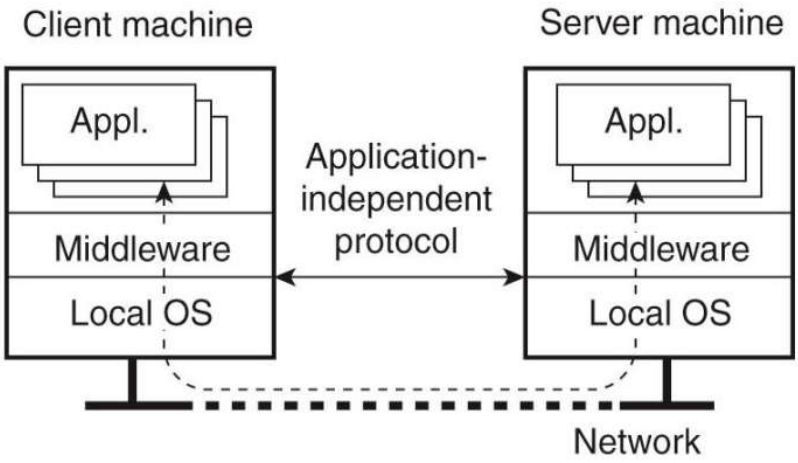
a) For each remote service the client machine will have a separate counterpart that can contact the service over the network. In this case, an **application-level protocol will handle the synchronization**.

b) Providing a **direct access to remote services** by only offering a convenient user interface.

- This means that the client machine is used **only as a terminal with no need for local storage**.
- In the case of networked user interfaces, everything is processed and stored at the server. This thin-client approach is receiving more attention as Internet connectivity increases.



(a) A networked application with its own protocol



(b) A general solution to allow access to remote applications

❖ *Servers*

- A server is a process implementing a specific service on behalf of a collection of clients.
- It waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.
- There are number of issues with server design:

a) How to organize the server,
b) Where clients contact a server,
c) How a server can be interrupted,
d) Server is stateless,

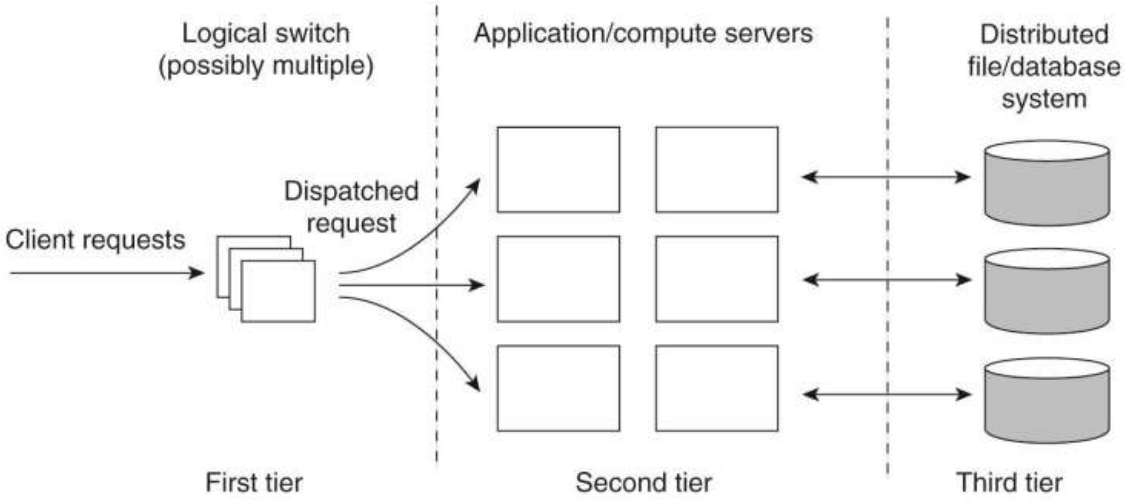
a) There are several ways to organize servers:

1. **Iterative server:** the server **itself handles** the request and, if necessary, **returns** a response to the requesting client.
2. **A concurrent server:** does not handle the request itself. It **passes it to a separate thread or another process**, after which it immediately waits for the next incoming request.
3. **A multithreaded server** is an example of a concurrent server which **forks a new process** for each new incoming request. The thread or process that handles the request is **responsible for returning a response** to the requesting client.

❖ Server Clusters

- Server cluster is a collection of machines connected through a network, a **local-area network**, often offering **high bandwidth and low latency**, where each machine **runs one or more servers**.
- A server cluster is logically organized into three tiers:

1. The first tier consists of a (logical) switch through which client requests are routed.
2. Application processing servers: these are typically servers running on high performance hardware dedicated to delivering compute power.
3. The third tier, which consists of data-processing servers, notably files and database servers.



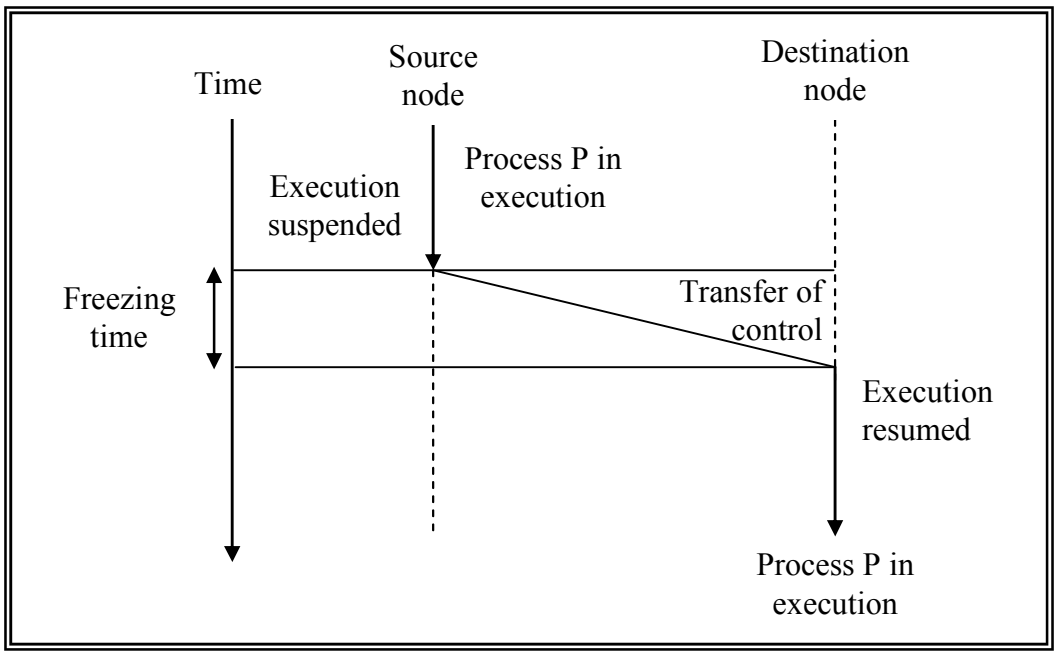
The General Organization of a Three-tiered Server Cluster

❖ Drawbacks

- In these clusters, there is often a **separate administration machine** that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.
- Most server clusters offer **a single access point**. When that point fails, the cluster becomes unavailable.

❖ *Code Migration*

Code Migration in distributed systems took place in the form of **process migration in which an entire process was moved from one machine to another**, for simplifying the design of a distributed system.



Flow Execution in Migration Process

❖ Reasons for Code Migration

1. ***Load Balancing***: overall system ***performance*** can be improved if processes are moved from heavily-loaded to lightly-loaded machines.
2. ***Minimize communication***: a computation may be migrated to a node where specific resources, data, or services are available to minimize the amount of communication.
3. ***Computation Speedup*** and ***Parallelism***: the total process turnaround time can be reduced if a single process can be divided into a number of sub-processes that can run concurrently on different sites.
4. ***Flexibility***: dynamically configuring the distributed system by deciding in advance where each part of an application should be executed.
5. ***Fault Tolerance***: this is done by migrating processes from nodes that may have a partial failure.

❖ **Models for Code Migration**

1. ***Weak Mobility***, in this model, it is possible to transfer only the process code segment, along with perhaps some initialization data. A characteristic feature of weak mobility involves the transfer of a process from its initial state, i.e., the process has not yet begun execution and hence does not require the transfer of the process state.

2. ***Strong Mobility***, both code and execution segments of a process can be transferred. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off.

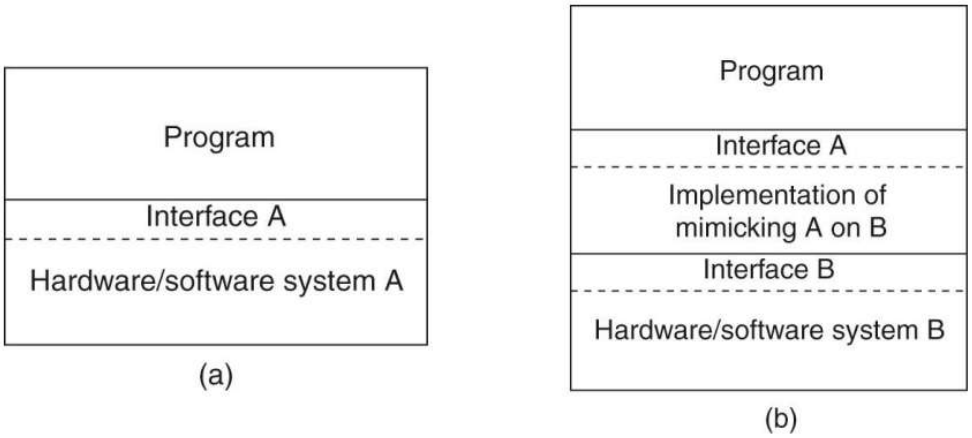
❖ *Virtualization*

Virtualization means to ***create a virtual version of a device or resource***, such as a server, storage device, network or even an operating system where the framework divides the resource into one or more execution environments.

- This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as ***resource virtualization***.

❖ **The Role of Virtualization in Distributed Systems**

Every (distributed) computer system offers a programming interface to higher level software. There are many different types of interfaces, ranging from the basic instruction set (**as (a) in the next figure**) as offered by a CPU to the huge collection of application programming interfaces that are shipped with many current middleware systems. Another important issue is that ***Virtualization*** deals with extending or replacing an existing interface, so as to mimic the behavior of another system (**as (b) in the next figure**).

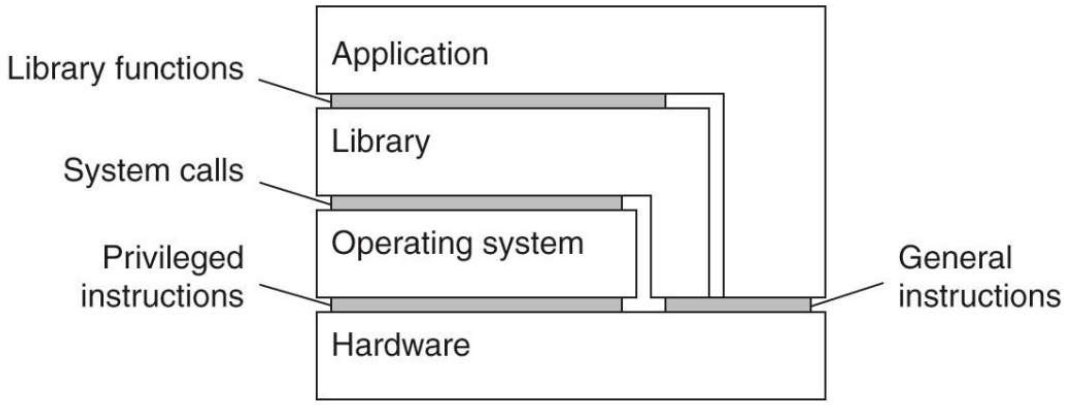


(a) General organization between a program, interface, and system.
(b) General organization of virtualizing system A on top of system B.

❖ Architectures of Virtual Machines

Computer systems generally offer *four different types* of interfaces, at *four different levels*:

1. An interface between the hardware and software, consisting of machine instructions that can be invoked by any program.
2. An interface between the hardware and software, consisting of machine instructions that can be invoked only by privileged programs, such as an operating system.
3. An interface consisting of system calls as offered by an operating system.
4. An interface consisting of library calls, generally forming what is known as an application programming interface (API). In many cases, the aforementioned system calls are hidden by an API.



Various Interfaces Offered by Computer Systems

Communication

❖ Network protocols

Communication in distributed systems is always based on ***low-level message passing*** as offered by the underlying network. While for non-distributed systems, communication is based on ***shared memory***.

1. Layered protocol:

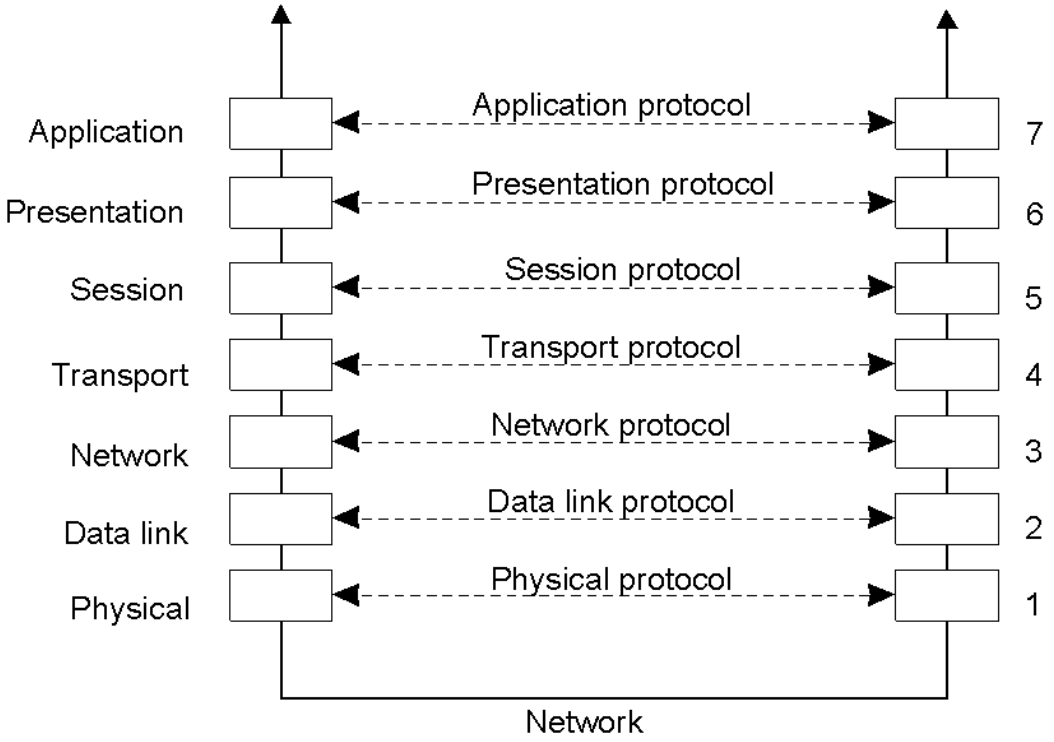
Many different agreements are needed to ensure correct communication between **A** and **B**. To make it easier to deal with different levels involved in communication, the **International Standards Organization (ISO)** developed a reference model that clearly *identifies the various levels*, *gives them standard names*, and *points out which level should do which job*. This model is called the **Open Systems Interconnection (OSI)** Reference Model.

The **OSI** model is designed to ***allow open systems to communicate***. An **open system** *is one that is prepared to communicate with any other open system by using standard rules that control the format, contents, and meaning of the messages sent and received.*

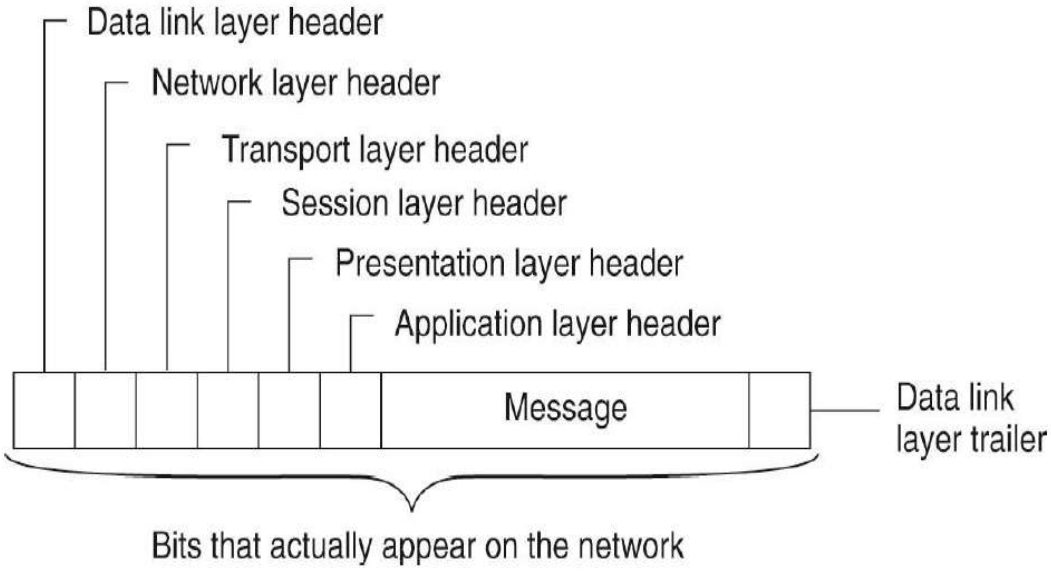
The **OSI** model, communication is divided up into **seven levels** or layers. Each layer deals with one specific aspect of the communication.

When process ***A*** on machine1 wants to communicate with process ***B*** on machine2:

- Process **A** builds a message in its own address space and executes a system call that causes the operating system to send the message over the network to **B**. Process **A** passes the message to the application layer on its machine.
- The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer.
- The presentation layer adds its own header and passes the result down to the session layer, and so on.
- Some layers add a **header** to the front with a **trailer** to the end.
- When it hits the bottom, the physical layer transmits the message, (which might look as shown in the second figure below) by putting it onto the physical transmission medium.
- When the message arrives at machine2, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver; process **B**, which may reply to it using the reverse path.



Layers, interfaces, and protocols in the OSI model.



A typical message as it appears on the network

A) Lower-Level Protocols

- ❖ The **physical layer** deals with **standardizing the electrical, mechanical, and signaling interfaces**. That is, how two computers are connected and how 0s and 1s are represented.
- ❖ The **data link layer** provides the means to detect and correct transmission errors.
- ❖ The **network layer** contains the protocols for routing a message through a computer network. The **network protocol IP** (Internet Protocol) is the most widely used Internet protocol. An **IP** packet (the technical term for a message in the network layer) can be sent without any setup (**connectionless**).

B) Transport Protocols

- Packets can be lost on the way from the sender to the receiver. The job of the transport layer is to recover this kind of error (**reliable connection** or **connection oriented**).
- Upon receiving a message from the application layer, the transport layer breaks it into pieces small enough for transmission, assigns each one a sequence number, and then sends them all.

C) Higher- Level Protocols

Above the transport layer, OSI distinguished three additional layers. In practice, only the application layer is ever used.

- The **session layer** provides support for sessions between applications.
- Most messages do not consist of random bit strings, but more structured information such as people's names, addresses, amounts of money, and so on. In the **presentation layer,** it is possible to define records containing fields like these.
- The **application layer** is used by end-user software such as web browsers and email clients. It provides protocols that allow software to send and receive information and present meaningful data to users. A typical application-specific protocol is the **HyperText Transfer Protocol** (HTTP), which is designed to remotely manage the transfer of Web pages. Some other examples are **File Transfer Protocol** (FTP) and **Domain Name System** (DNS).

2. Middleware protocol:

Middleware is an application that logically lives in the application layer, it contains many general-purpose protocols to support a variety of middleware services.

Middleware communication protocols support *high-level communication services*. For example, protocols that allow **a process to call a procedure or an object on a remote machine**. Likewise, there are high-level communication services for **setting and synchronizing streams for transferring real-time data**, such as those needed for multimedia applications.

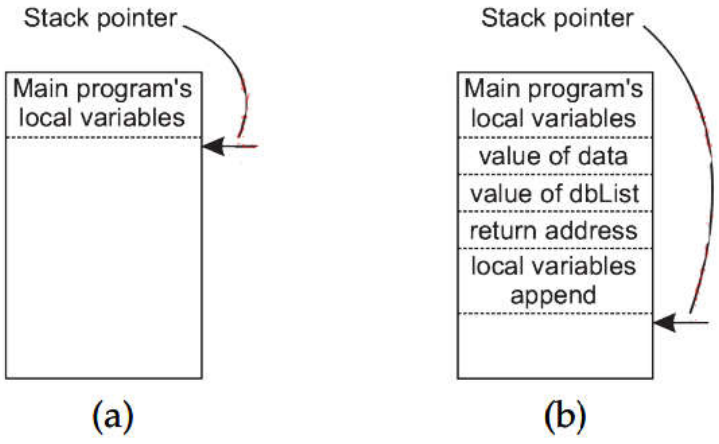
❖ **Remote Procedure Call**

When a process on machine **A** calls a procedure on machine **B**, the calling process on **A** is suspended, and execution of the called procedure takes place on **B**. Information is transported from the caller to the callee in the parameters and comes back in the procedure result. This method is known as **Remote Procedure Call**, or often just **RPC**.

- ***Basic RPC Operation***

a) Conventional Procedure Call

A conventional (i.e., single machine) procedure call: Consider a call in C like: **count = read (fd, buf, nbytes)**; where ***fd***: an integer indicating a file, ***buf***: an array of characters into which data are read, and ***nbytes***: an integer telling how many bytes to read. If the call is made from the main program, the stack will be as shown in **(a)** below. The caller pushes the parameters onto the stack in order, as shown in **(b)**. After the read procedure has finished running it removes the return address and transfers control back to the caller. The caller removes the parameters from the stack, returning the stack to the original state it had before the call.

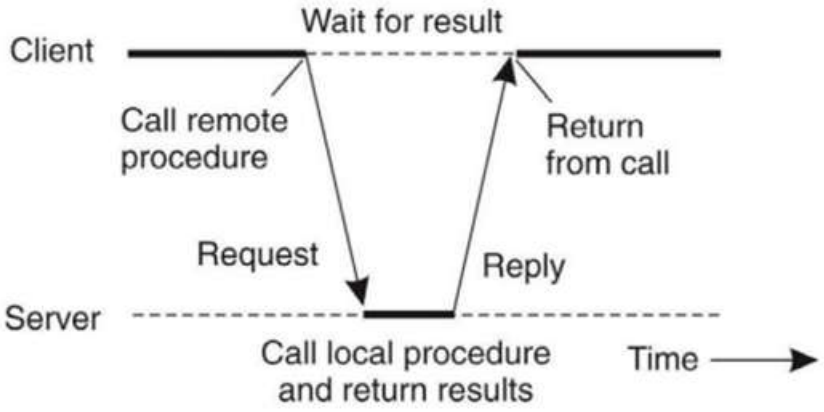


(a) Parameter passing in a local procedure call: the stack before the call to read.
(b) The stack while the called procedure is active.

b) Client and Server Stubs

When ***read*** is a remote procedure (e.g., one that will run on the file server's machine), a different version of ***read***, called a ***client stub*** is used. It is called using the calling sequence of the previous figure (b) which does a call to the local operating system. Then, it packs the parameters into a message and requests that message to be sent to the server. Following the call to ***send***, the client stub blocks itself until the reply comes back. When the message arrives at the server, the server's operating system passes it up to a ***server stub***.

A server stub is the server-side equivalent of a client stub: ***it is a piece of code that transforms requests coming in over the network into local procedure calls.*** The server stub will have called ***receive*** and be blocked waiting for incoming messages. The ***server stub unpacks the parameters from the message and then calls the server procedure*** in the usual way (i.e., as in the previous figure). The server performs its work and then returns the result to the caller in the usual way.



Principle of RPC between a client and server program.

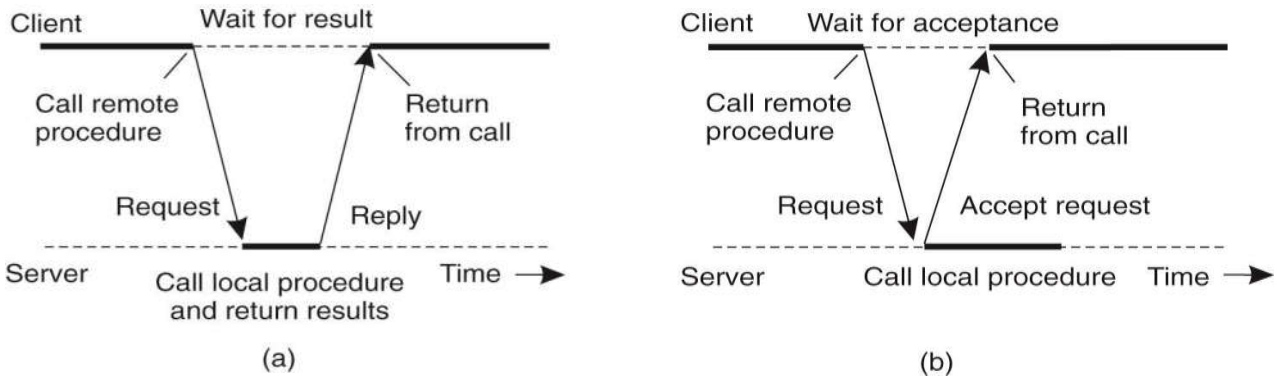
The steps of a remote procedure call can be summarized:

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

- ***Asynchronous RPC***

When a client calls a remote procedure, the client will **block** until a reply is returned. This behaviour is unnecessary when there is no result to return, and only leads to blocking the client while it could have proceeded and done useful work just after requesting the remote procedure to be called.

With **asynchronous RPCs**, the server immediately sends a reply back to the client the moment the RPC request is received, after which it calls the requested procedure. The reply acts as an acknowledgement to the client that the server is going to process the RPC. The client will continue without further blocking as soon as it has received the server's acknowledgement.



(a) The interaction between client and server in a traditional RPC.
(b) The interaction using asynchronous RPC.

2- MESSAGE-ORIENTED COMMUNICATION

❖ *Message-Oriented Transient Communication*

Many distributed systems and applications are built directly on top of the simple message-oriented model offered by the transport layer as part of middleware solutions.

❖ *Berkeley Sockets*

A socket is a 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.

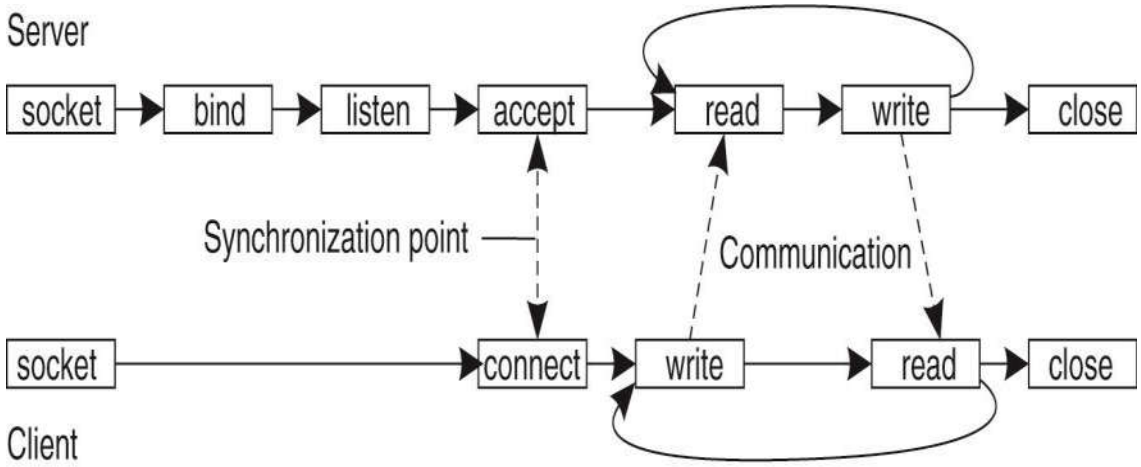
A socket forms an abstraction over the actual communication end point that is used by the local operating system for a specific transport protocol.

The following table shows the socket primitives for TCP/IP

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

Servers generally execute the first four primitives in the order given. The bind primitive associates a local address with the newly-created socket. For example, a server should bind the IP address of its machine together with a port number to a socket. Binding tells the operating system that the server wants to receive messages only on the specified address and port.

A client and server follow a general pattern for connection-oriented communication using sockets.



Connection-oriented communication pattern using sockets

- ***The Message-Passing Interface (MPI)***

The independence of hardware and platform led to the definition of a standard for message passing, called the Message-Passing Interface or MPI which assumes communication takes place within a known group of processes.

Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A *(group/D, process/D)* pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address.

- ***Message-Oriented Persistent Communication***

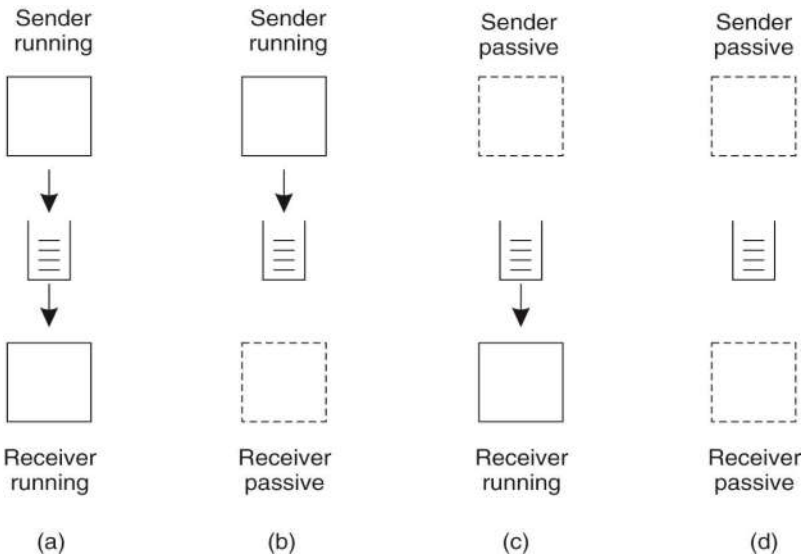
It is an important class of message-oriented middleware services, generally known as **message-queuing systems**, or just **Message-Oriented Middleware (MOM)** which provide extensive support for persistent asynchronous communication.

The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission.

❖ *Message-Queuing Model*

The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues. These messages are forwarded over a series of communication servers and are eventually delivered to the destination, even if it was down when the message was sent. An important aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient's queue.

- Once a message has been deposited in a queue, it will remain there until it is removed, irrespective of whether its sender or receiver is executing. This gives us the following four combinations with respect to the execution mode of the sender and receiver:



a) Both the sender and receiver execute during the entire transmission of a message.

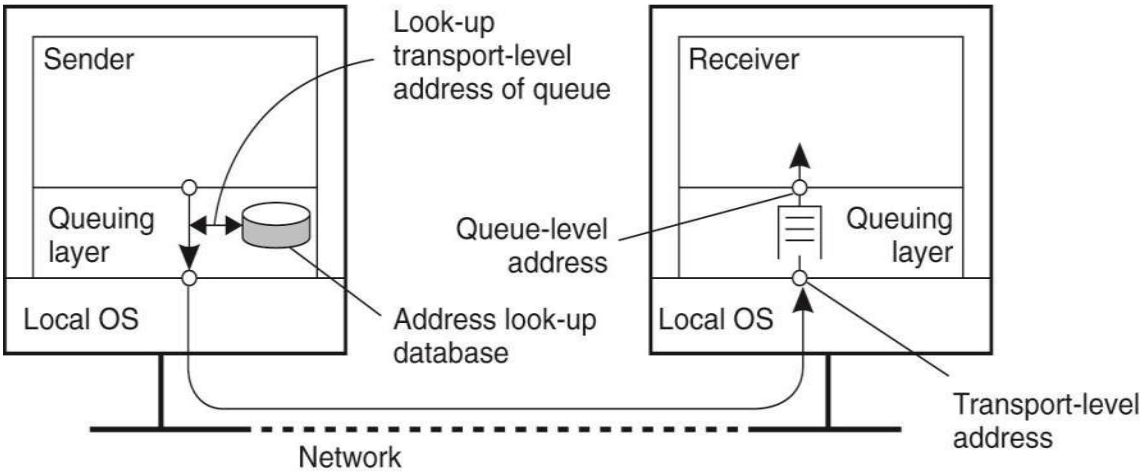
b) Only the sender is executing, while the receiver is passive.

c) The receiver can read messages that were sent to it, but respective senders may not execute.

d) The system is storing messages even while sender and receiver are passive.

❖ ***General Architecture of a Message-Queuing System***

- The first restrictions that is messages can be put only' into queues that are local to the sender and messages can be read only from local queues.
- However, a message put into a queue will contain the specification of a destination queue. It is important to realize that the collection of queues is distributed across multiple machines.
- Consequently, for a message-queuing system to transfer messages, it should maintain a mapping of queues to network locations which means maintain a database of queue names to network locations.



The relationship between queue-level addressing and network-level addressing.

Queues are managed by **queue managers**. Normally, a queue manager interacts directly with the application that is sending or receiving a message.

❖ ***Message Brokers***

In message-queuing systems, conversions are handled by special nodes in a queuing network, known as message brokers. A message broker acts as an application-level gateway in a message-queuing system. Its main purpose is to convert incoming messages so that they can be understood by the destination application.

This kind of queuing system is used if the collection of applications that make up a distributed information system is highly diverse.

❖ ***Channels***

An important component of MQ is formed by the message channels. Each message channel has exactly one associated send queue from which it fetches the messages it should transfer to the other end. Each of message channels is managed by a message channel agent (MCA).

Each MCA (Message Channel Agent) has a set of associated attributes that determine the overall behaviour of a channel.

- Message TransferTo transfer a message from one queue manager to another (possibly remote) queue manager it is necessary that each message carries its destination address.
- An address in MQ consists of two parts: the name of the queue manager to which the message is and the name of the destination queue resorting.

It is also necessary to specify the route that a message should follow. In most cases, routes are explicitly stored inside a queue manager in a routing table. To manage routing, a programming interface is needed that is relatively simple, called the Message Queue Interface (MQI) and the most important primitives of MQI are summarized in below.

Primitive	Description
MQopen	Open a (possibly remote) queue
MQclose	Close a queue
MQput	Put a message into an opened queue
MQget	Get a message from a (local) queue

3- Stream-Oriented Communication

There are forms of communication in which timing plays a crucial role which offer to exchange time-dependent information such as audio and video streams.

- ***Support for Continuous Media***
  - ***Data Stream***

To capture the exchange of time-dependent information, distributed systems generally provide support for data streams. A data stream is a sequence of data units. Data streams can be applied to discrete as well as continuous media. Timing is crucial to continuous data streams.

To capture timing aspects, a distinction is often made between different transmission modes:

- **Asynchronous transmission mode:** the data items in a stream are transmitted one after the other, but there are no further timing constraints on when transmission of items should take place.

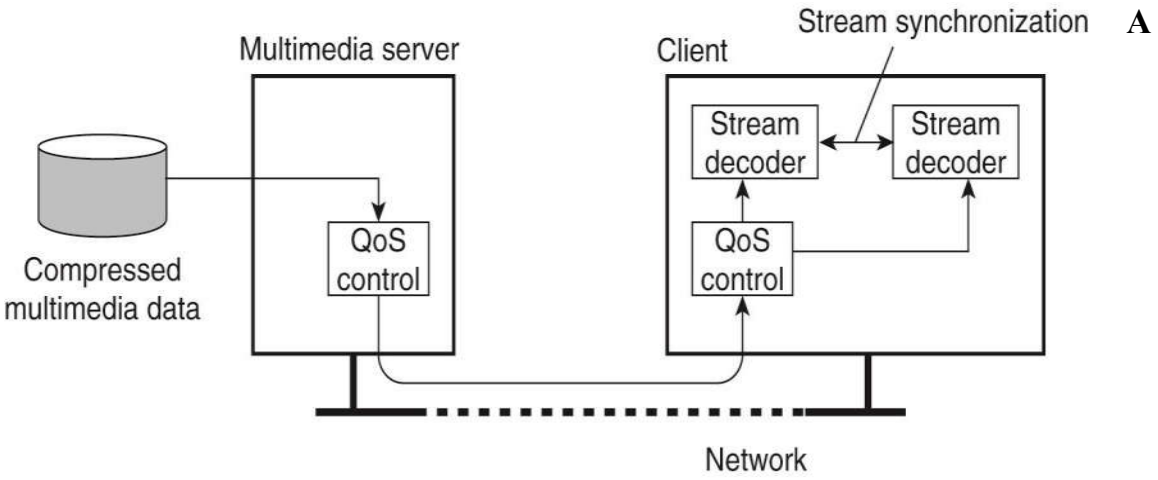
- **Synchronous transmission mode**, there is a maximum end-to-end delay defined for each unit in a data stream.

- **Isochronous transmission mode**, it is necessary that data units are transferred on time. This means that data transfer is subject to a maximum and minimum end-to-end delay.

Streams can be 1) A **simple stream** consists of only a single sequence of data, or 2) a **complex stream** consists of several related simple streams, called substreams.

This general architecture reveals a number of important issues:

- The multimedia data, notably video and to a lesser extent audio, will need to be compressed substantially in order to reduce the required storage and especially the network capacity.
- Controlling the quality of the transmission and synchronization issues.



General architecture for streaming stored multimedia data over a network.

➢ ***Streams and Quality of Service:***

Timing (and other nonfunctional) requirements are generally expressed as Quality of Service (QoS) requirements. These requirements describe what is needed from the underlying distributed system and network to be ensured. QoS for continuous data streams mainly concerns timeliness, volume, and reliability.

To specify required QoS, the following properties are specified:

- 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.

❖ ***Stream Synchronization***

Synchronization of streams deals with maintaining temporal relations between streams.

- ***Synchronization Mechanisms***

Two issues need to be distinguished to perform synchronization: (1) the basic mechanisms for synchronizing two streams, and (2) the distribution of those mechanisms in a networked environment.

Synchronization mechanisms can be viewed at several different levels:

- At the lowest level, synchronization is done explicitly by operating on the data units of simple streams.
- A better approach is to offer an application interface that allows it to more easily control streams and devices.

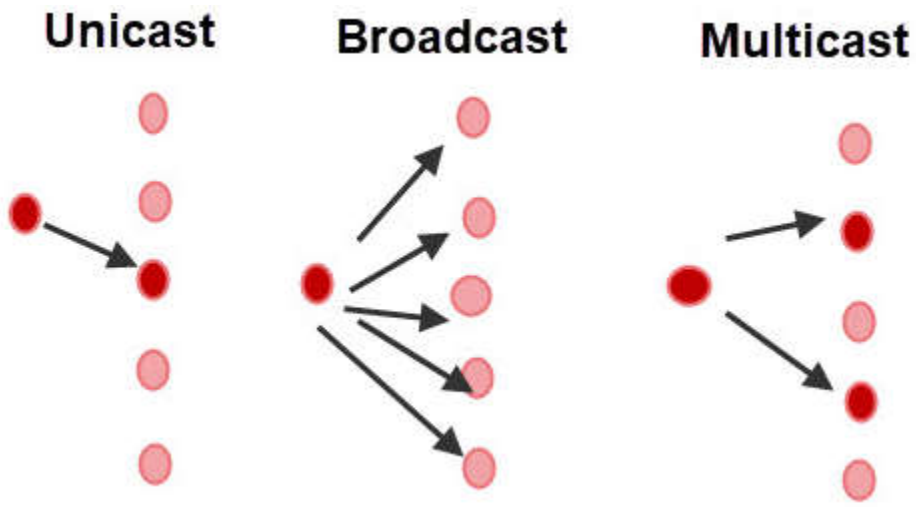
4- Multicast Communication

An important topic in communication of distributed systems is the support for sending data to multiple receivers, also known as multicast communication.

Multicast is group communication where data transmission is addressed to a group of destination computers simultaneously. Multicast can be one-to-many or many-to-many distribution.

Data is transported over a network by three simple methods i.e. Unicast, Broadcast, and Multicast.

- Unicast
  - One-to-one
  - Destination – unique receiver host address
- Broadcast
  - One-to-all
  - Destination – address of network
- Multicast
  - One-to-many
  - Multicast group must be identified
  - Destination – address of group



- ***Multicast application examples***

- Financial services

  Delivery of news, stock quotes, financial indices, etc.

- Remote conferencing/e-learning

  Streaming audio and video to many participants (clients, students)

  Interactive communication between participants

- Data distribution

  e.g., distribute experimental data from Large Hadron Collider (LHC) at CERN lab to interested physicists around the world