Syllabus: Characterization of distributed systems-Introduction, examples of distributed systems, resource sharing and web, challenges. System Models: Introduction, Architectural models: S/w layers, system architecture and variants, Interface and Objects, Design requirements for distributed architectures, Fundamental Models: Interaction Model, Failure Model and Security Model

CHARACTERIZATION OF DISTRIBUTED SYSTEMS:

INTRODUCTION

Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks – all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading *distributed systems*.

Distributed system is the one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

Characteristics of Distributed Systems are,

***Concurrency*:** In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

***No global clock*:** When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network.

***Independent failures*:** All computer systems can fail, and it is the responsibility of system Designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a Computer, or the unexpected termination of a program somewhere in the system (a *crash*), is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running.

EXAMPLES OF DISTRIBUTED SYSTEMS

Typical examples of Distributed systems are,

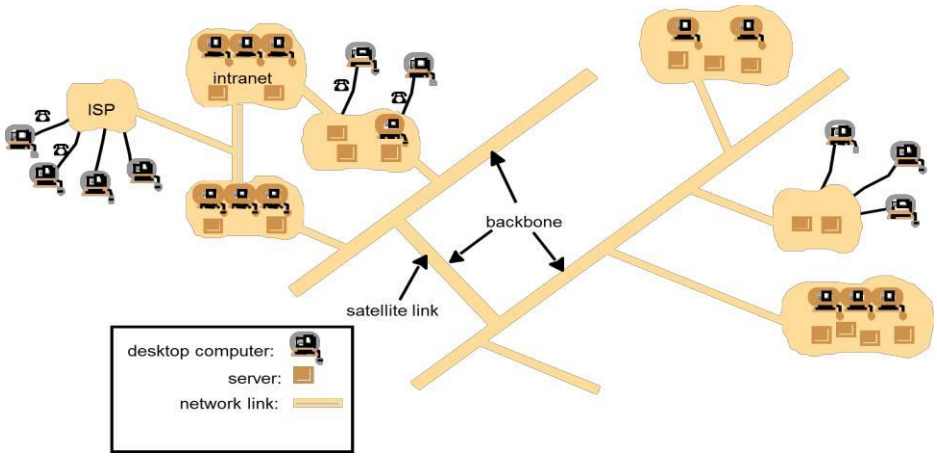
The Internet

Intranets

Mobile and Ubiquitous computing.

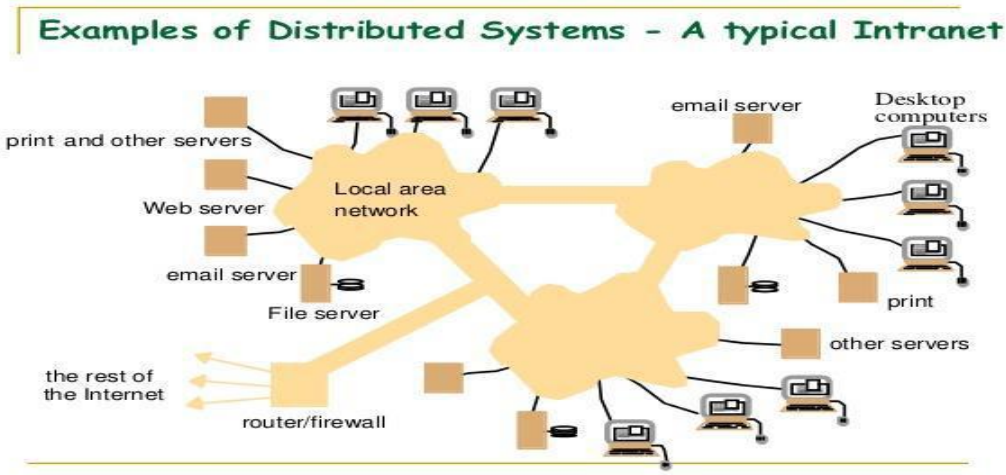
The Internet:

Internet is a very large distributed system. It enables users, wherever they are, to make use of services like www, email, file transfer. The set of services is open-ended. Refer figure below which shows a typical portion of internet. Internet connects millions of LANs and MANs to each other.



Intranet

- An intranet is a portion of the internet that is separately administered and has a boundary that can be configured to enforce local security policies.
- It may be composed of several LANs linked by backbone connections.
- The n/w configuration of a particular intranet is the responsibility of the organization that administers it.
- An intranet is connected to the Internet via router, which allows the users to use the services available in the Internet.



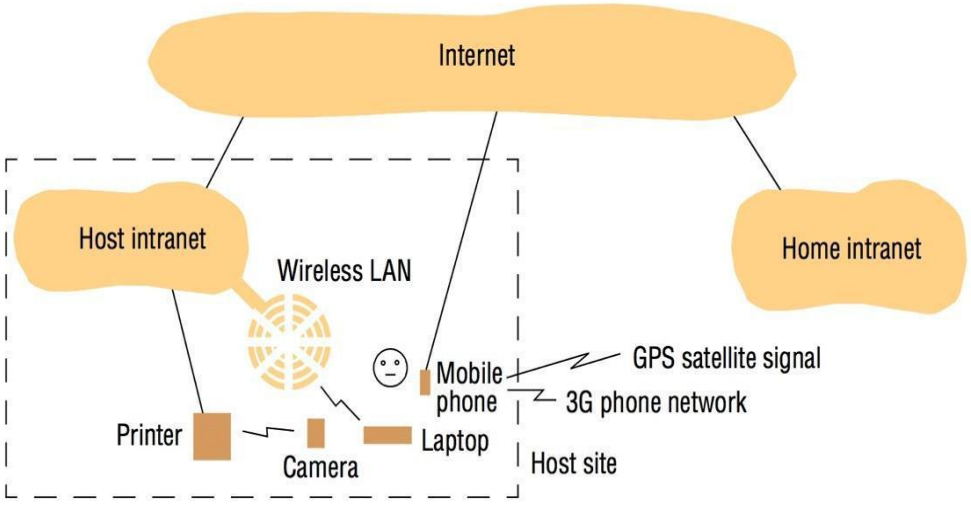
- Firewall is used to protect intranet by preventing unauthorized messages leaving or entering.
- Some organizations do not wish to connect their internal networks to the Internet at all. E.g. police and other security and law enforcement agencies are likely to have at least some internal networks that are isolated from outside world.
- These organizations can be connected to Internet to avail the services by dispensing with the firewall.
- The main issues arising in the design of components for use in intranets are, File services are needed to enable users to share data

  Firewalls should ensure legitimate access to services.

  Cost of installation and support should be minimum.

Mobile and Ubiquitous computing:

- ✓ Integration of portable computing devices like Laptops, smartphones, handheld devices, pagers, digital cameras, smart watches, devices embedded in appliances like refrigerators, washing machines, cars etc. with the distributed systems became possible because of the technological advances in device miniaturization and wireless networking.
- ✓ These devices can be connected to each other conveniently in different places, makes mobile computing possible.
- ✓ Figure below shows how a user from home intranet can access the resources at Host intranet using mobile devices.



- ✓ In mobile computing, users who are away from home intranet, are still allowed to access resources via the devices they carry.
- ✓ Ubiquitous computing is the harnessing of many small, cheap computational devices that are present in user's physical environments, including home, office and others.
- ✓ The term ubiquitous is intended to suggest that small computing devices will eventually become so pervasive in everyday objects that they are scarcely noticed.
- ✓ The presence of computers everywhere is useful only when they can communicate with one another.
- ✓ E.g. it would be convenient for users to control their washing machine and hi-fi system using "Universal remote control" device at home.
- ✓ The mobile user can get benefit from computers that are everywhere.

- ✓ Ubiquitous computing could benefit users while they remain in a single environment such as the home, office or hospital.
- ✓ Figure below shows a user who is visiting a host organization. The users home intranet and the host intranet at the site that the user is visiting. Both intranets are connected to the rest of the Internet.

Resource Sharing and Web

We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.

Looked at from the point of view of hardware provision, we share equipment such as printers and disks to reduce costs.

But of far greater significance to users is the sharing of the higher-level resources that play a part in their applications and in their everyday work and social activities. For example, users are concerned with sharing data in the form of a shared database or a set of web pages – not the disks and processors on which they are implemented.

Similarly, users think in terms of shared resources such as a search engine or a currency converter, without regard for the server or servers that provide these.

In practice, patterns of resource sharing vary widely in their scope and in how closely users work together. At one extreme, a search engine on the Web provides a facility to users throughout the world, users who need never come into contact with one another directly. At the other extreme, in *computer-supported cooperative working* (CSCW), a group of users who cooperate directly share resources such as documents in a small, closed group. The pattern of sharing and the geographic distribution of particular users determines what mechanisms the system must supply to coordinate users' actions.

We use the term *service* for a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications. For example, we access shared files through a file service; we send documents to printers through a printing service; we buy goods through an electronic payment service. The only access we have to the service is via the set of operations that it exports. For example, a file service provides *read*, *write* and *delete* operations on files.

The fact that services restrict resource access to a well-defined set of operations is in part standard software engineering practice. But it also reflects the physical organization of distributed systems. Resources in a distributed system are physically encapsulated within computers and can only be accessed from other computers by means of communication. For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.

The term *server* is probably familiar to most readers. It refers to a running program (a *process*) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately. The requesting processes are referred to as *clients,* and the overall approach is known as *client-server computing*. In this approach, requests are sent in messages from clients to a server and replies are sent in messages from the server to the clients. When the client sends a request for an operation to be carried out, we say that the client *invokes an operation* upon the server. A complete interaction between a client and a server, from the point when the client sends its request to when it receives the server's response, is called a *remote invocation*.

World Wide Web

key feature of the Web is that it provides a *hypertext* structure among the documents that it stores, reflecting the users' requirement to organize their knowledge. This means that documents contain *links* (or *hyperlinks*) – references to other documents and resources that are also stored in the Web.

The Web is an *open* system: it can be extended and implemented in new ways without disturbing its existing functionality. First, its operation is based on communication standards and document or content standards that are freely published and widely implemented. For example, there are many types of browser, each in many cases implemented on several platforms; and there are many implementations of web servers. Any conformant browser can retrieve resources from any conformant server. So users have access to browsers on the majority of the devices that they use, from mobile phones to desktop computers.

Second, the Web is open with respect to the types of resource that can be published and shared on it. At its simplest, a resource on the Web is a web page or some other type of *content* that can be presented to the user, such as media files and documents in Portable Document Format. If somebody invents, say, a new image-storage format, then images in this format can

immediately be published on the Web. Users require a means of viewing images in this new format, but browsers are designed to accommodate new content-presentation functionality in the form of 'helper' applications and 'plug-ins'.

The Web has moved beyond these simple data resources to encompass services, such as electronic purchasing of goods. It has evolved without changing its basic architecture. The Web is based on three main standard technological components:

- The Hypertext Markup Language (HTML), a language for specifying the contents and layout of pages as they are displayed by web browsers.
- Uniform Resource Locators (URLs), also known as Uniform Resource Identifiers (URIs), which identify documents and other resources stored as part of the Web.
- A client-server system architecture, with standard rules for interaction (the Hypertext Transfer Protocol – HTTP) by which browsers and other clients fetch documents and other resources from web servers.

HTML:

- It is used to specify text and images that make up the contents of a web page and to say how they are laid out and formatted for presentation to the user.
- Different tags are used in html.
- Either we can produce html by hand or using any HTML-aware editor.
- The html text is stored in a file that a web server can access.

URL:

- The purpose of URL is to identify a resource. Browsers looks up the corresponding URL when user clicks on a link or selects one of their bookmarks.
- Every URL has two top level components
- The first component "scheme" declares which type of URL this is.
  E.g. mailto: xyz@abc.com indicates a user's email address, or
  ftp://ftp.dowloadIt.com/software/aProg.exe identifies a file to be retrieved using FTP.
- The second component specifies the path of the resource.

HTTP:

- Hypertext Transfer protocol defines the ways in which browsers and other types of client interact with web servers.
- Main features are,

- Request-reply interactions:
- Content types
- One resource per request
- Simple access control.

CHALLENGES IN DISTRIBUTED SYSTEMS

HETEROGENEITY:

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- **Networks;**
- **Computer hardware;**
- **Operating systems;**
- **programming languages;**
- **Implementations by different developers.**

Although the Internet consists of many different sorts of network their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network.

Data types such as integers may be represented in different ways on different sorts of hardware – for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware.

Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another. Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data

structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

OPENNESS:

Openness cannot be achieved unless the specification and documentation of the Key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures, which are usually cumbersome and slow-moving.

However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people.

Systems that are designed to support resource sharing in this way are termed *open distributed systems* to emphasize the fact that they are extensible. They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services and the reimplementation of old ones, enabling application programs to share resources. A further benefit that is often cited for open systems is their independence from individual vendors.

SECURITY:

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality (protection against disclosure to unauthorized individuals), integrity (protection against alteration or corruption), and availability (protection against interference with the means to access the resources).

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example:

1. A doctor might request access to hospital patient data or send additions to that data.

2. In electronic commerce and banking, users send their credit card numbers across the Internet.

In both examples, the challenge is to send sensitive information in a message over a network in a secure manner. But security is not just a matter of concealing the contents of messages – it also involves knowing for sure the identity of the user or other agent on whose behalf a message was sent. In the first example, the server needs to know that the

User is really a doctor, and in the second example, the user needs to be sure of the identity of the shop or bank with which they are dealing. The second challenge here is to identify a remote user or other agent correctly. Both of these challenges can be met by the use of encryption techniques developed for this purpose.

However, the following two security challenges have not yet been fully met: *Denial of service attacks*: *Security of mobile code*:

SCALABILITY:

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers and servers in the Internet has increased dramatically. Figure below shows the increasing number of computers and web servers during the 12-year history of the Web up to 2005.

Date	*Computers*	*Web servers*	*Percentage*
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July	~200,000,000	42,298,371	21
2005, July	353,284,187	67,571,581	19

It is interesting to note the significant growth in both computers and web servers in this period, but also that the relative percentage is flattening out – a trend that is explained by the growth of fixed and mobile personal computing. One web server may also increasingly be hosted on multiple computers.

The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources:

Controlling the performance loss:

Preventing software resources running out:

Avoiding performance bottlenecks:

FAILURE HANDLING

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. Failures in a distributed system are partial – that is, some components fail while others continue to function. Therefore the handling of failures is particularly difficult. The following are techniques for dealing with failures.

***Detecting failures*:** Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file. Chapter 2 explains that it is difficult or even impossible to detect some other failures, such as a remote crashed server in the Internet. The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

***Masking failures*:** Some failures that have been detected can be hidden or made less severe. Two examples of hiding failures:

1. Messages can be retransmitted when they fail to arrive.
2. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Just dropping a message that is corrupted is an example of making a fault less severe – it could be retransmitted. The reader will probably realize that the techniques described for hiding failures are not guaranteed to work in the worst cases; for example, the data on the second disk may be corrupted too, or the message may not get through in a reasonable time however often it is retransmitted.

***Tolerating failures*:** Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components. Their clients can be designed to tolerate failures, which generally involves the users tolerating them as well. For example, when a web browser cannot contact a web server, it does not make the user wait forever while it keeps on trying – it informs the user about the problem, leaving them free to try again later. Services that tolerate failures are discussed in the paragraph on redundancy below.

***Recovery from failures*:** Recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed. In general, the computations performed by some programs will be incomplete when a fault occurs, and the permanent data that they update (files and other material stored in permanent storage) may not be in a consistent state.

***Redundancy*:** Services can be made to tolerate failures by the use of redundant components.

CONCURRENCY

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time.

The process that manages a shared resource could take one client request at a time. But that approach limits throughput. Therefore services and applications generally allow multiple client requests to be processed concurrently. To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads. In this case it is possible that several threads may be executing concurrently within an object, in which case their operations on the object may conflict with one another and produce inconsistent results.

For example, if two concurrent bids at an auction are 'Smith: $122' and 'Jones: $111', and the corresponding operations are interleaved without any control, then they might get stored as 'Smith: $111' and 'Jones: $122'.

The moral of this story is that any object that represents a shared resource in a distributed system must be Responsible for ensuring that it operates correctly in a concurrent environment. This applies not only to servers but also to objects in applications. Therefore any programmer

who takes an implementation of an object that was not intended for use in a distributed system must do whatever is necessary to make it safe in a concurrent environment.

TRANSPARENCY

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

Access transparency enables local and remote resources to be accessed using identical operations.

Location transparency enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.

Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

The two most important transparencies are access and location transparency; their presence or absence most strongly affects the utilization of distributed resources. They are sometimes referred to together as *network transparency*.

QUALITY OF SERVICE

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance*.

Adaptability to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

Reliability and security issues are critical in the design of most computer systems. The performance aspect of quality of service was originally defined in terms of responsiveness and

computational throughput, but it has been redefined in terms of ability to meet timeliness guarantees, as discussed in the following paragraphs.

Some applications, including multimedia applications, handle *time-critical data* – streams of data that are required to be processed or transferred from one process to another at a fixed rate. For example, a movie service might consist of a client program that is retrieving a film from a video server and presenting it on the user's screen. For a satisfactory result the successive frames of video need to be displayed to the user within some specified time limits.

In fact, the abbreviation QoS has effectively been commandeered to refer to the ability of systems to meet such deadlines. Its achievement depends upon the availability of the necessary computing and network resources at the appropriate times. This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time (for example, the task of displaying a frame of video).

The networks commonly used today have high performance – for example, BBC iPlayer generally performs acceptably – but when networks are heavily loaded their performance can deteriorate, and no guarantees are provided. QoS applies to operating systems as well as networks. Each critical resource must be reserved by the applications that require QoS, and there must be resource managers that provide guarantees. Reservation requests that cannot be met are rejected.

SYSTEM MODELS:

INTRODUCTION:

Systems that are intended for use in real-world environments should be designed to function correctly in the widest possible range of circumstances and in the face of many possible difficulties and threats.

Different system models are,

Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements; the computational elements being individual computers or aggregates of them supported by appropriate network interconnections.

Fundamental models take an abstract perspective in order to examine individual aspects of a distributed system. In this chapter we introduce fundamental models that examine three important aspects of distributed systems: *interaction models*, which consider the structure and sequencing of the communication between the elements of the system;
failure models, which consider the ways in which a system may fail to operate correctly and; *security models*, which consider how the system is protected against attempts to interfere with its correct operation or to steal its data.

ARCHITECTURAL MODEL

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. The architectural design of a building has similar aspects – it determines not only its appearance but also its general structure and architectural style (gothic, neo-classical, modern) and provides a consistent frame of reference for the design.

Software Layers

- Software architecture referred to:
  - The structure of software as layers or modules in a single computer.
  - The services offered and requested between processes located in the same or different computers.
- Software architecture is breaking up the complexity of systems by designing them through layers and services.
  - Layer: a group of related functional components.
  - Service: functionality provided to the next layer.
- **Platform**

The lowest-level hardware and software layers are often referred to as a platform for distributed systems and applications.

→ These low-level layers provide services to the layers above them, which are implemented independently in each computer.

→ These low-level layers bring the system's programming interface up to a level that facilitates communication and coordination between processes.

- Common examples of platform are: Intel x86/Windows, Intel x86/Linux Intel x86/Solaris , SPARC/SunOS, PowePC/MacOS

Middleware

It was a layer of software whose purpose is

→ To mask heterogeneity presented in distributed systems and provides interoperability between lower layer and upper layer.

→ To provide a convenient programming model to application developers.

→ Major Examples of middleware are:

- Sun RPC (Remote Procedure Calls)
- OMG CORBA (Common Request Broker Architecture)
- Microsoft D-COM (Distributed Component Object Model)
- Sun Java RMI

System Architectures

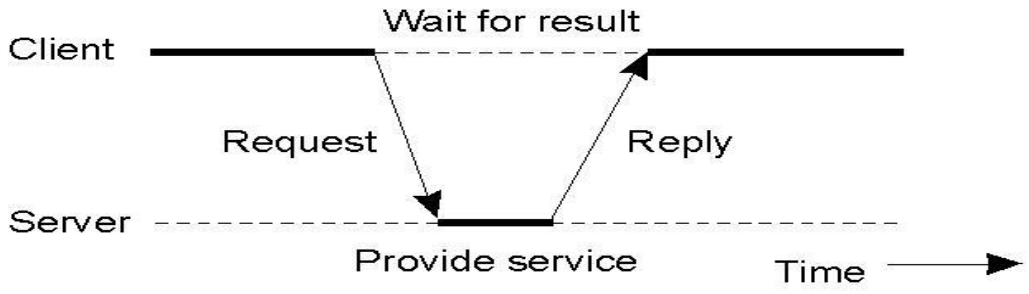
- The most evident aspect of distributed system design is the division of responsibilities between system components (applications, servers, and other processes) and the placement of the components on computers in the network.
- It has major implication for:
  - Performance
  - Reliability
  - Security

In a distributed system, processes with well-defined responsibilities interact with each other to perform a useful activity. The two major types of architectural models are described below.

Client-server and peer-to-peer.

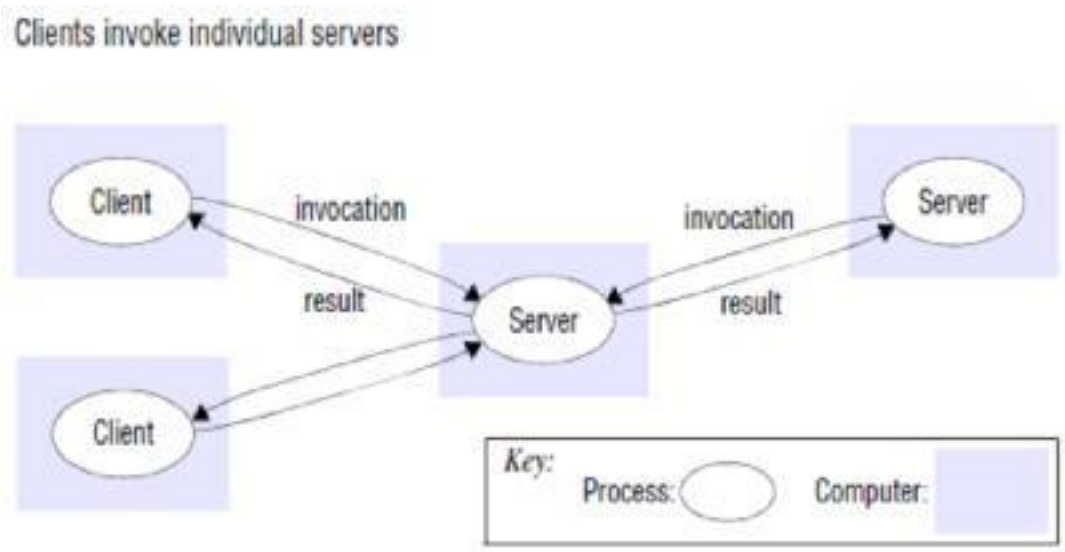
Client Server:

This is the architecture that is most often cited when distributed systems are discussed. It is historically the most important and remains the most widely employed.



In particular, client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage. Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet domain names to network addresses. Another web-related example concerns *search engines*, which enable users to look up summaries of information available on web pages at sites throughout the Internet. These summaries are made by programs called *web crawlers*, which run in the background at a search

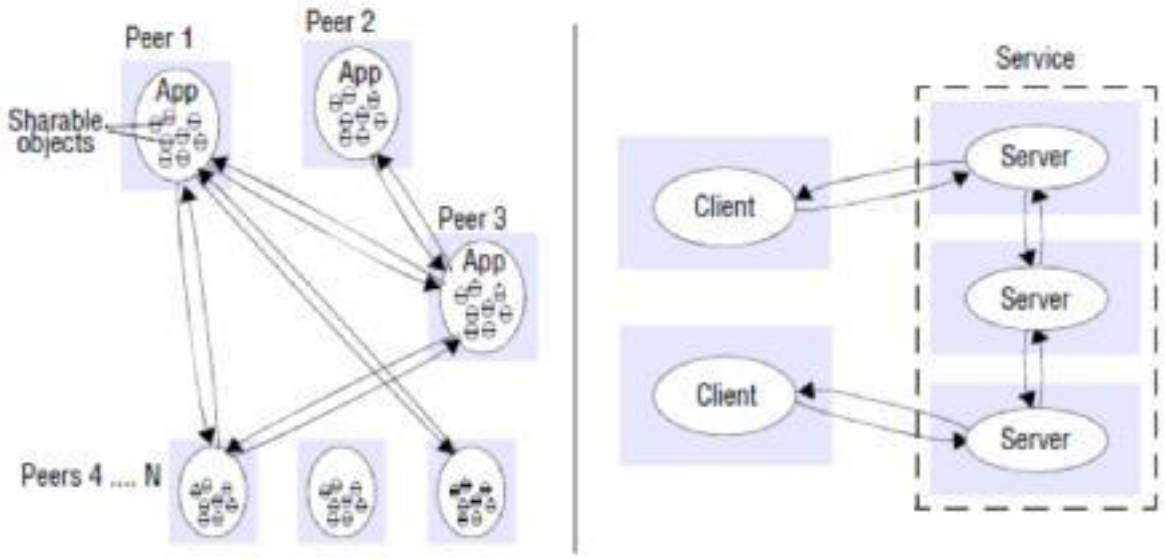
engine site using HTTP requests to access web servers throughout the Internet. Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers. In this example, the server tasks (responding to user queries) and the crawler tasks (making requests to other web servers) are entirely independent; there is little need to synchronize them and they may run concurrently. In fact, a typical search engine would normally include many concurrent threads of execution, some serving its clients and others running web crawlers.



Peer to Peer :

In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as *peers* without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other.

While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly. The centralization of service provision and management implied by placing a Service at a single address does not scale well beyond the capacity of the computer that hosts the service and the bandwidth of its network connections.



Above figure illustrates the form of a peer-to-peer application. Applications are composed of large numbers of peer processes running on separate computers and the pattern of communication between them depends entirely on application requirements. A large number of data objects are shared, an individual computer holds only a small part of the application database, and the storage, processing and communication loads for access to objects are distributed across many computers and network links. Each object is replicated in several computers to further distribute the load and to provide resilience in the event of disconnection of individual computers. The need to place individual objects and retrieve them and to maintain replicas amongst many computers renders this architecture substantially more complex than the client-server architecture.

Variants of Client Server Model

The problem of client-server model is placing a service in a server at a single address that does not scale well beyond the capacity of computer host and bandwidth of network connections. To address this problem, several variations of client-server model have been proposed. Some of these variations are discussed below.

Services provided by multiple servers

- Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes.
- E.g. cluster that can be used for search engines.

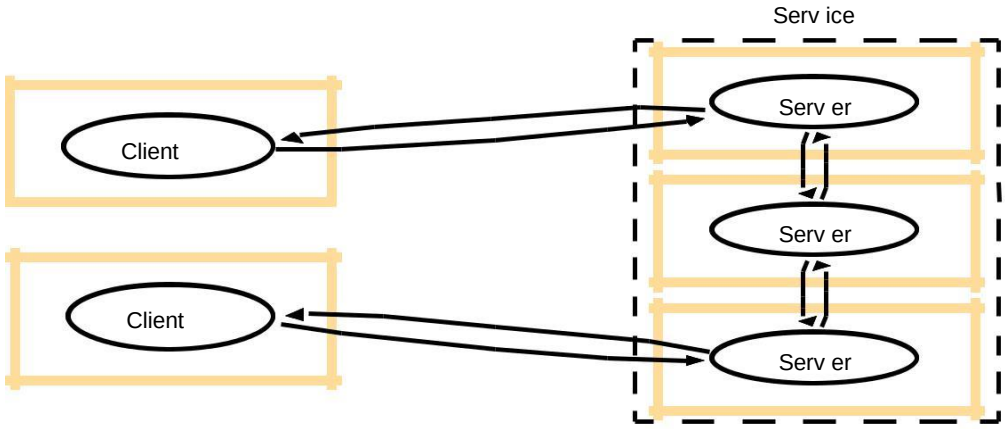
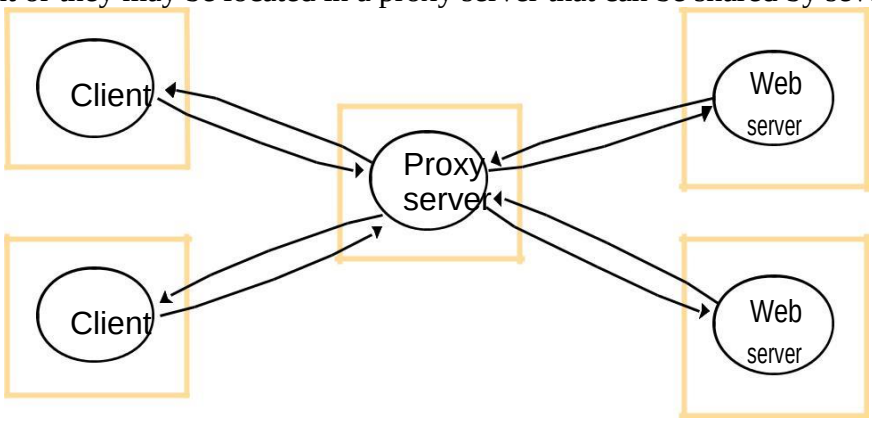


Figure. A service provided by multiple servers

Proxy servers and caches

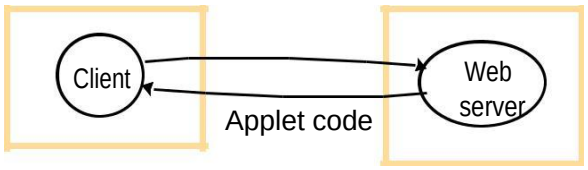
A cache is a store of recently used data objects. When a new object is received at a computer it is added to the cache store, replacing some existing objects if necessary. When an object is needed by a client process the caching service first checks the cache and supplies the object from there if an up-to-date copy is available. If not, an up-to-data copy is fetched. Caches may be collected with each client or they may be located in a proxy server that can be shared by several clients.



Mobile code

- Applets are a well-known and widely used example of mobile code.
- Applets downloaded to clients give good interactive response
- Mobile codes such as Applets are a potential security threat to the local resources in the destination computer.
- Browsers give applets limited access to local resources. For example, by providing no access to local user file system.
  - E.g. a stockbroker might provide a customized service to notify customers of changes in the prices of shares; to use the service, each customer would have to download a special applet that receives updates from the broker's server, display them to the user and perhaps performs automatic to buy and sell operations triggered by conditions set up by the customer and stored locally in the customer's computer.

a) client request results in the downloading of applet code



b) client interacts with the applet

Figure. Web applets

Mobile agents

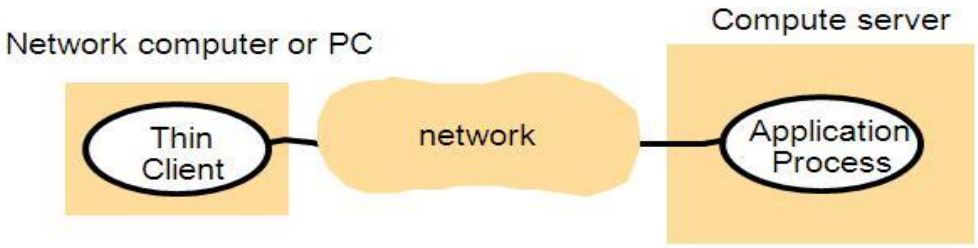
- A running program (code and data) that travels from one computer to another in a network carrying out of a task, usually on behalf of some other process.
- Examples of the tasks that can be done by mobile agents are:
  - To collecting information.
  - To install and maintain software maintain on the computers within an organization.

Network computers

- It downloads its operating system and any application software needed by the user from a remote file server.
- Applications are run locally but the file are managed by a remote file server.
- Network applications such as a Web browser can also be run.

Thin clients

- It is a software layer that supports a window-based user interface on a computer that is local to the user while executing application programs on a remote computer.
- This architecture has the same low management and hardware costs as the network computer scheme.
- Instead of downloading the code of applications into the user's computer, it runs them on a compute server.
- Compute server is a powerful computer that has the capacity to run large numbers of application simultaneously.



Mobile devices and spontaneous interoperation

- Mobile devices are hardware computing components that move between physical locations and thus networks, carrying software component with them.
- Many of these devices are capable of wireless networking ranges of hundreds of meters such as WiFi (IEEE 802.11), or about 10 meters such as Bluetooth.
- Mobile devices include:
  - Laptops
  - Personal digital assistants (PDAs)
  - Mobile phones
  - Digital cameras
  - Wearable computers such as smart watches

Design Requirements for distributed architectures

Performance Issues

→ Performance issues arising from the limited processing and communication capacities of computers and networks are considered under the following subheading:

- Responsiveness
  - E.g. a web browser can access the cached pages faster than the non-cached pages.
- Throughput
- Load balancing
  - E.g. using applets on clients, remove the load on the server.

Quality of service

→ The ability of systems to meet deadlines.

→It depends on availability of the necessary computing and network resources at the appropriate time.

→This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time.

- E.g. the task of displaying a frame of video

→ The main properties of the quality of the service are:

- Reliability
- Security
- Performance
- Adaptability

Use of caching and replication

→ Distributed systems overcome the performance issues by the use of data replication and caching.

Dependability issues

→ Dependability of computer systems is defined as:

- Correctness
- Security

→ Security is locating sensitive data and other resources only in computers that can be secured effectively against attack.

E.g. a hospital database

Fault tolerance

→ Dependable applications should continue to function in the presence of faults in hardware, software, and networks.

→ Reliability is achieved by redundancy.

FUNDAMENTAL MODELS

***Interaction*:** Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes. In the analysis and design of distributed systems we are concerned especially with these interactions. The interaction model must reflect the facts that communication takes place with delays that are often of considerable duration, and that the accuracy with which independent processes can be coordinated is limited by these delays and by the difficulty of maintaining the same notion of time across all the computers in a distributed system.

***Failure*:** The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs (including software faults) or in the network that connects them. Our model defines and classifies the faults. This provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

***Security*:** The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents. Our security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

INTERACTION MODEL

The discussion of system architectures in indicates that fundamentally distributed systems are composed of many processes, interacting in complex ways. For example:

• Multiple server processes may cooperate with one another to provide a service; the examples mentioned above were the Domain Name System, which partitions and replicates its data at servers throughout the Internet, and Sun's Network Information Service, which keeps replicated copies of password files at several servers in a local area network.

Two significant factors affecting interacting processes in a distributed system:

• Communication performance is often a limiting characteristic.

• It is impossible to maintain a single global notion of time.

Performance of communication channels

→ The communication channels in our model are realized in a variety of ways in distributed systems, for example

- By an implementation of streams
- By simple message passing over a computer network

→ Communication over a computer network has the performance characteristics such as:

Latency

The delay between the start of a message's transmission from one process to the beginning of its receipt by another.

Bandwidth

The total amount of information that can be transmitted over a computer network in a given time. Communication channels using the same network, have to share the available bandwidth.

Jitter

The variation in the time taken to deliver a series of messages. It is relevant to multimedia data. For example, if consecutive samples of audio data are played with differing time intervals then the sound will be badly distorted.

Interaction Model- Computer clocks and timing events

Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Two processes running on different computers can associate timestamp with their events. Even if two processes read their clock at the same time, their local clocks may supply different time. This is because computer clock drift from perfect time and their drift rates differ from one another. Clock drift rate refers to the relative amount that a computer clock differs from a perfect reference clock. Even if the clocks on all the computers in a distributed system are set to the same time initially, their clocks would eventually vary quite significantly unless corrections are applied. There are several techniques to correcting time on computer clocks. For example, computers may use radio signal receivers to get readings from GPS (Global Positioning System) with an accuracy about 1 microsecond.

Interaction Model-Variations:

Two variants of the interaction model are

Synchronous distributed systems

- It has a strong assumption of time
- The time to execute each step of a process has known lower and upper bounds.
- Each message transmitted over a channel is received within a known bounded time.
- Each process has a local clock whose drift rate from real time has a known bound.

Asynchronous distributed system

- It has no assumption about time.
- There is no bound on process execution speeds.
- Each step may take an arbitrary long time.
- There is no bound on message transmission delays.
- A message may be received after an arbitrary long time.
- There is no bound on clock drift rates.
- The drift rate of a clock is arbitrary.

Event ordering

In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after, or concurrently with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.

❖ For example, consider a mailing list with users X, Y, Z, and A.

1. User X sends a message with the subject Meeting.
2. Users Y and Z reply by sending a message with the subject RE: Meeting.

- In real time, X's message was sent first, Y reads it and replies; Z reads both X's message and Y's reply and then sends another reply, which references both X's and Y's messages.
- But due to the independent delays in message delivery, the messages may be delivered in the order is shown in figure 10.
- It shows user A might see the two messages in the wrong order.

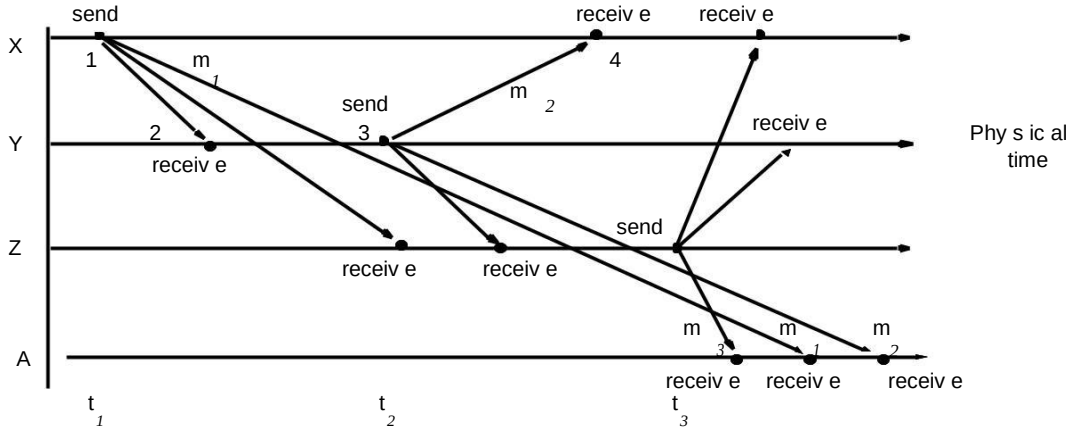


Figure. Real-time ordering of events

→ Some users may view two messages in the wrong order, for example, user A might see → *Item* is a sequence number that shows the order of receiving emails.

Item	*From*	*Subject*
23	Z	Re: Meeting
24	X	Meeting
26	Y	Re: Meeting

Since clocks cannot be synchronized perfectly across distributed system, Lamport proposed a model of logical time that provides ordering among events in a distributed system.

E.g. in the previous example we know that the message is received after it was sent. Hence a logical order can be derived here,

x sends m1 before y receives m1

Y sends m2(reply) before x receives m2(reply).

We also know that reply are send after receiving message.

Hence, we can say that,

Y receives m1 before sending m2.

FAILURES MODEL

In a distributed system both processes and communication channels may fail – that is, they may depart from what is considered to be correct or desirable behavior. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures.

Omission failures • The faults classified as *omission failures* refer to cases when a **process** or **communication Channel** fails to perform actions that it is supposed to do.

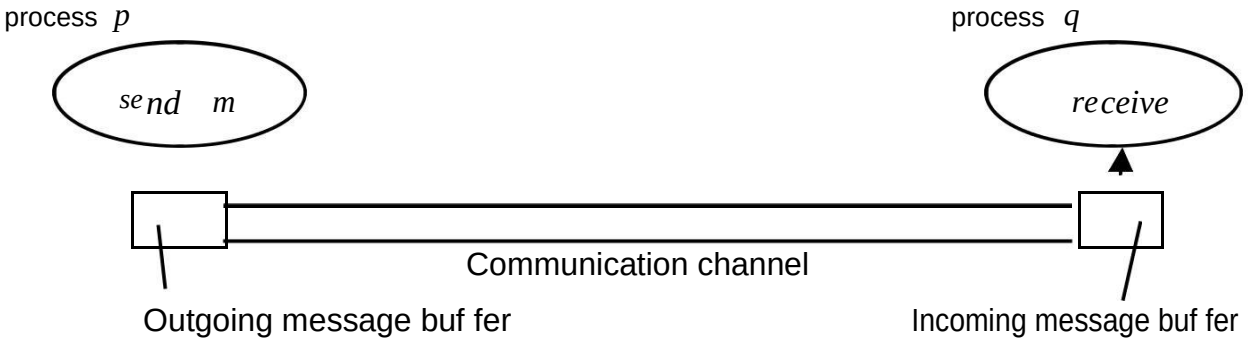
Process omission failures: The chief omission failure of a process is to crash. When we say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever.

Other processes may be able to detect such a crash by the fact that the process repeatedly fails to respond to invocation messages. However, this method of crash detection relies on the use of *timeouts* – that is, a method in which one process allows a fixed period of time for something to occur. In an asynchronous system a timeout can indicate only that a process is not responding – it may have crashed or may be slow, or the messages may not have arrived.

A process crash is called *fail-stop* if other processes can detect certainly that the process has crashed. Fail-stop behavior can be produced in a synchronous system if the processes use timeouts to detect when other processes fail to respond and messages are guaranteed to be delivered. For example, if processes *p* and *q* are programmed for *q* to reply to a message from *p*, and if process *p* has received no reply from process *q* in a maximum time measured on *p*'s local clock, then process *p* may conclude that process *q* has failed.

Communication omission failures: Consider the communication primitives *send* and *receive*. A process *p* performs a *send* by inserting the message *m* in its outgoing message buffer. The communication channel transports *m* to *q*'s incoming message buffer. Process *q* performs a *receive* by taking *m* from its incoming message buffer and delivering it as shown below. The outgoing and incoming message buffers are typically provided by the operating system.

- ✓ The communication channel produces an omission failure if it does not transport a message from *p*'s outgoing message buffer to *q*'s incoming message buffer.
- ✓ This is known as 'dropping messages' and is generally caused by lack of buffer space at the receiver or at an intervening gateway, or by a network transmission error.
- ✓ The loss of messages between the sending process and the outgoing message buffer called as *send omission failures*,
- ✓ loss of messages between the incoming message buffer and the receiving process called as *receive-omission failures*,
- ✓ and to loss of messages in between is called as *channel omission failures*.



Arbitrary failures • The term *arbitrary* or *Byzantine* failure is used to describe the worst possible failure semantics, in which any type of error may occur. For example, a process may set

wrong values in its data items, or it may return a wrong value in response to an invocation. An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps. Arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.

Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare because the communication software is able to recognize them and reject the faulty messages. For example, checksums are used to detect corrupted messages, and message sequence numbers can be used to detect nonexistent and duplicated messages.

The omission failures are classified together with arbitrary failures shown in Figure

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Figure: Omission and arbitrary failures

Timing failures • Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Figure: Timing failures

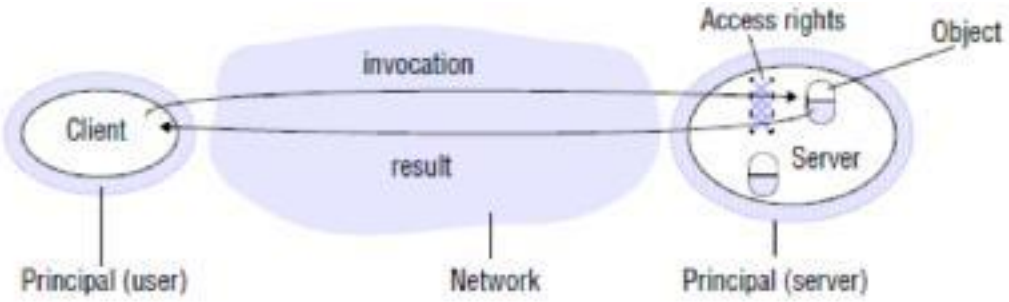
SECURITY MODEL

Sharing of resources as a motivating factor for distributed systems, we described their architecture in terms of processes, potentially encapsulating higher-level abstractions such as objects, components or services, and providing access to them through interactions with other processes. That architectural model provides the basis for our security model: the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access. Protection is described in terms of objects, although the concepts apply equally well to resources of all types.

Protecting objects:

Figure below shows a server that manages a collection of objects on behalf of some users. The users can run client programs that send invocations to the server to perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client. Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages. To support this, *access rights* specify who is allowed to perform the operations of an object – for example, who is allowed to read or to write its state. Thus we must include users in our model as the beneficiaries of access rights. We do so by associating with each invocation and each result the authority on which it is issued. Such an authority is called a *principal*. A principal may be a user or a process. In our illustration, the invocation comes from a user and the result from a server.

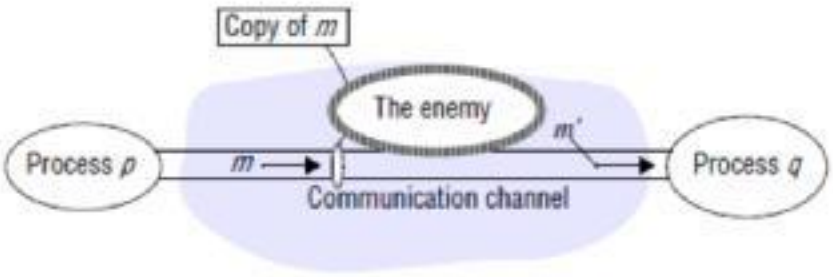
The server is responsible for verifying the identity of the principal behind each invocation and checking that they have sufficient access rights to perform the requested operation on the particular object invoked, rejecting those that do not. The client may check the identity of the principal behind the server to ensure that the result comes from the required server.



Securing processes and their interactions • Processes interact by sending messages. The messages are exposed to attack because the network and the communication service that they use are open, to enable any pair of processes to interact. Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process. Distributed systems are

often deployed and used in tasks that are likely to be subject to external attacks by hostile users. This is especially true for applications that handle financial transactions, confidential or classified information or any other information whose secrecy or integrity is crucial. Integrity is threatened by security violations as well as communication failures. So we know that there are likely to be threats to the processes of which such applications are composed and to the messages travelling between the processes. But how can we analyze these threats in order to identify and defeat them? The following discussion introduces a model for the analysis of security threats.

The enemy • To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in Figure below. Such attacks can be made simply by using a computer connected to a network to run a program that reads network messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an unauthorized manner.



The threats from a potential enemy include *threats to processes* and *threats to communication channels.*

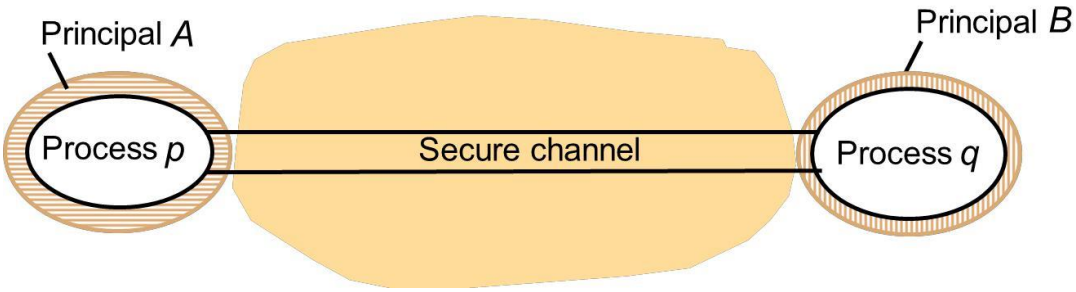
Threats to processes: A process that is designed to handle incoming requests may receive a message from any other process in the distributed system, and it cannot necessarily determine the identity of the sender. Communication protocols such as IP do include the address of the source computer in each message, but it is not difficult for an enemy to generate a message with a forged source address. This lack of reliable knowledge of the source of a message is a threat to the correct functioning of both servers and clients, as explained below:

***Servers*:** Since a server can receive invocations from many different clients, it cannot necessarily determine the identity of the principal behind any particular invocation. Even if a server requires the inclusion of the principal's identity in each invocation, an enemy might generate an invocation with a false identity. Without reliable knowledge of the sender's identity, a server cannot tell whether to perform the operation or to reject it.

***Clients*:** When a client receives the result of an invocation from a server, it cannot necessarily tell whether the source of the result message is from the intended server *Clients*: When a client receives the result of an invocation from a server, it cannot necessarily tell whether the source of

the result message is from the intended server or from an enemy, perhaps 'spoofing' the mail server. Thus the client could receive a result that was unrelated to the original invocation, such as a false mail item (one that is not in the user's mailbox).

Threats to communication channels: An enemy can copy, alter or inject messages as they travel across the network and its intervening gateways. Such attacks present a threat to the privacy and integrity of information as it travels over the network and to the integrity of the system. For example, a result message containing a user's mail item might be revealed to another user or it might be altered to say something quite different. Another form of attack is the attempt to save copies of messages and to replay them at a later time, making it possible to reuse the same message over and over again. For example, someone could benefit by resending an invocation message requesting a transfer of a sum of money from bank account to another. All these threats can be defeated by the use of *secure channels.*



Important Questions:

1. Define a distributed system and explain the same with two examples.
2. Analyse the different challenges of distributed system.
3. Discuss the types of hardware and software resources which can be shared in distributed system with an illustration.
4. Discuss the Software Layers of distributed system architectural model.
5. What are variations of client-server model?
6. Describe the interaction model of distributed system.
7. Write the design requirements for Distributed architectures.
8. Describe the failure model of distributed system.
9. Write about security model in distributed system.