UNIT-2

SYSTEM MODELS

➔Introduction:-

There are 3 important ways to design the distributed systems:

"Physical models" capture the hardware composition of a system in terms of computers (and other devices, such as mobile phones) and their interconnecting networks.

"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. Client-server and peer-to-peer are two of the most commonly used forms of architectural model for distributed systems.

"Fundamental models" examine individual aspects of a distributed system. Fundamental models examine 3 important aspects of distributed systems. Message communication over a computer network can be affected by delays, can suffer from a variety of failures and is vulnerable to security attacks. These issues are addressed by three models:

• ***The interaction model*** considers the structure and sequencing of the communication between the elements of the system.

• ***The failure model*** consider the ways in which a system may fail to operate correctly

• ***The security model*** considers how the system is protected against attempts to interfere with its correct operation.

Difficulties and threats for distributed systems **:** Here are some of the problems that the designers of distributed systems face.

>***Widely varying modes of use***: The component parts of systems are subject to wide variations in workload – for example, some web pages are accessed several million times a day. Some parts of a system may be disconnected, or poorly connected some of the time – for example, when mobile computers are included in a system. Some applications have special requirements for high communication bandwidth and low latency – for example, multimedia applications.

>***Wide range of system environments***: A distributed system must accommodate heterogeneous hardware, operating systems and networks. The networks may differ widely in performance – wireless networks operate at a fraction of the speed of local networks. Systems of widely differing scales, ranging from tens of computers to millions of computers must be supported.

>***Internal problems***: Non-synchronized clocks, conflicting data updates and many modes of hardware and software failure involving the individual system components.

>***External threats***: Attacks on data integrity and secrecy, denial of service attacks.

➔Physical models:-

A physical model is a representation of the underlying hardware elements of a distributed system that abstracts specific details of the computer and networking technologies employed.

Baseline physical model: A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.

Beyond this baseline model, we can usefully identify three generations of distributed systems.

Early distributed systems: These systems emerged in the late 1970s and early 1980s in response to the emergence of local area networking technology, usually Ethernet. These systems typically consisted of between 10 and 100 nodes interconnected by a local area network, with limited Internet connectivity and supported a small range of services such as shared local printers and file servers as well as email and file transfer across the Internet. Individual systems were largely homogeneous and openness was not a concern.

Internet-scale distributed systems: These systems started to emerge in the 1990s in response to the dramatic growth of the Internet. This is an extensible set of nodes interconnected by a *network of networks* (the Internet). The level of heterogeneity in such systems is significant in terms of networks, computer architecture, operating systems, languages employed and the development teams involved. This has led to an increasing emphasis on open standards and associated middleware technologies such as CORBA and more recently, web services.

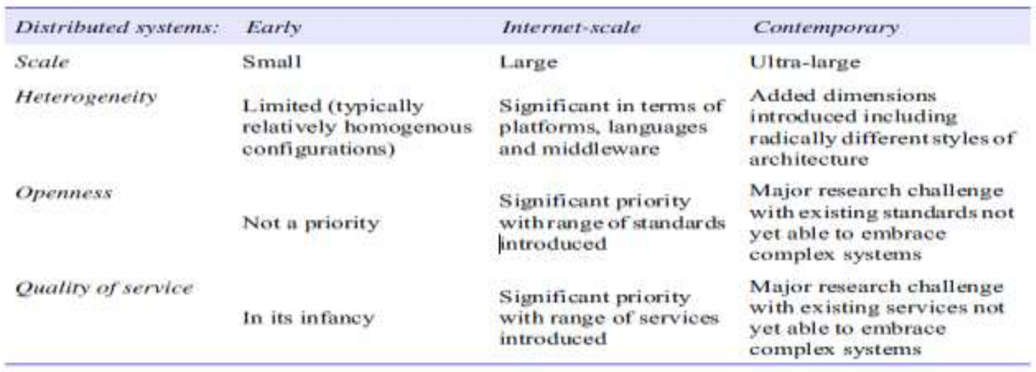
Contemporary distributed systems: In the above systems, nodes were typically desktop computers and therefore relatively static (that is, remaining in one physical location for extended periods), discrete (not embedded within other physical entities) and autonomous (to a large extent independent of other computers in terms of their physical infrastructure). The key trends identified are

• The emergence of *mobile computing* has led to physical models where nodes such as laptops or smart phones move from location to location leading to the need for added capabilities such as support for spontaneous interoperation.

• The emergence of *ubiquitous computing* has led to a move from discrete nodes to architectures where computers are embedded in everyday objects such as washing machines or in smart homes.

• The emergence of *cloud computing* and, in particular, cluster architectures have led to a move from autonomous nodes performing a given role to pools of nodes.

The net result is a physical architecture with a significant increase in the level of heterogeneity, involve up to hundreds of thousands of nodes.



Distributed systems:	*Early*	*Internet-scale*	*Contemporary*
Scale	Small	Large	Ultra-large
Heterogeneity	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
Openness	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
Quality of service	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

Generations of distributed systems

Distributed system of systems:
A recent report captures the complexity of modern distributed systems by referring to such architectures as "*systems of systems*" (mirroring the view of the Internet as a network of networks).
A "*system of systems"* can be defined as a complex system consisting of a series of subsystems that are systems in their own right and that come together to perform a particular task or tasks.

➔Architectural models:-

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 future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. This adopts a three-stage approach:

1) Looking at the core underlying ***architectural elements*** of the modern distributed systems,

2) examining composite ***architectural patterns*** that can be used in isolation or, in combination, in developing more sophisticated distributed systems solutions;

3) And finally, considering ***middleware*** platforms that are available to support the various styles of programming.

1) Architectural elements:-

To understand the fundamental building blocks of a distributed system, we need to consider four key questions:

• What are the ***entities that are communicating*** in the distributed system?

• How do they communicate, or, more specifically, what ***communication paradigm*** is used?

• What are the ***roles and responsibilities*** do they have in the overall architecture?

• How are they mapped on to the ***physical distributed infrastructure*** (what is their *placement*)?

→Communicating entities: It is helpful to address the first question from a *system-oriented* and a *problem-oriented* perspective.

From a **system perspective**, the answer is the entities that communicate in a distributed system are typically ***processes***.

• In some primitive environments, such as sensor networks, the underlying operating systems may not support process abstractions, and hence the entities that communicate in such systems are ***nodes***.
• In most distributed system environments, processes are supplemented by ***threads,*** so, it is threads that are the endpoints of communication.

From a **programming perspective**,

Objects: Objects have been introduced to enable and encourage the use of object oriented approaches in distributed systems. In distributed object-based approaches, a computation consists of a number of interacting objects representing natural units of decomposition for the given problem domain. Objects are accessed via interfaces, with an associated interface definition language (or IDL).

Components: Components resemble objects in that they offer problem-oriented abstractions for building distributed systems and are also accessed through interfaces. The key difference is that components specify not only their (provided) interfaces but also the assumptions they make in terms of other components/interfaces that must be present for a component to fulfill its function – in other words, making all dependencies explicit and providing a more complete contract for system construction. Component-based middleware often provides additional support for key areas such as deployment and support for server-side programming.

Web services: Web services are closely related to objects and components, again taking an approach based on encapsulation of behavior and access through interfaces. The World Wide Web consortium (W3C) defines a web service as:
… *a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based message exchanges via Internet-based protocols.*

→Communication paradigms:- we consider 3 types of communication paradigms:

1. Inter process communication;
2. Remote invocation;
3. Indirect communication.

1.Inter process communication refers to the communication between processes in distributed systems, including message-passing primitives, direct access to the API offered by Internet protocols (socket programming) and support for multicast communication.

2. Remote invocation represents the most common communication paradigm in distributed systems, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation.

(i) Request-reply protocols: Request-reply protocols are effectively a pattern imposed on an underlying message-passing service to support client-server computing. Such protocols involve a pair wise exchange of messages from client to server and then from server back to client, with the first message containing an encoding of the operation to be executed at the server and also an array of bytes holding associated arguments and the second message containing any results of the operation, again encoded as an array of bytes.

***(ii)Remote procedure calls*:** The concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing. In RPC, procedures in processes on remote computers can be called as if they are procedures in the local address space. This approach supports client-server computing with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally.RPC systems therefore offer access and location transparency.

(iii)Remote method invocation: Remote method invocation (RMI) strongly resembles remote procedure calls but in a world of distributed objects. With this approach, a calling object can invoke a method in a remote object. As with RPC, the underlying details are generally hidden from the user. RMI implementations may, go further by supporting object identity and the associated ability to pass object identifiers as parameters in remote calls.

The above sets of techniques all have one thing in common: communication represents a two-way relationship between a sender and a receiver with senders explicitly directing messages/invocations to the associated receivers. Receivers are also generally aware of the identity of senders, and in most cases both parties must exist at the same time.

3. Indirect Communication allows decoupling between senders and receivers.
In particular:
• Senders do not need to know who they are sending to (***space uncoupling***).
• Senders and receivers do not need to exist at the same time (***time uncoupling***)

Indirect communication techniques are:

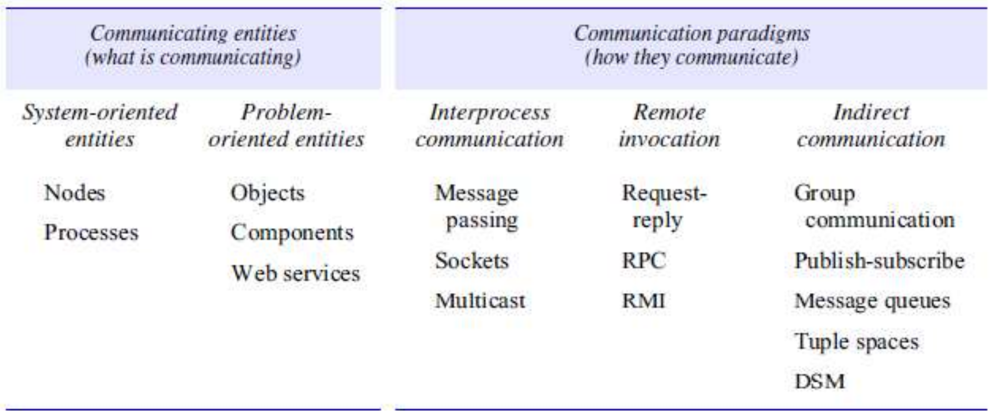
>*Group communication*: Group communication is concerned with the delivery of messages to a set of recipients and hence is a multiparty communication paradigm supporting one-to-many communication. Group communication relies on the abstraction of a group which is represented in the system by a group identifier. Recipients elect to receive messages sent to a group by joining the group. Senders then send messages to the group via the group identifier.

>*Publish-subscribe systems*: Many systems, such as the financial trading can be classified as information-dissemination systems wherein a large number of producers (or publishers) distribute information items of interest (events) to a similarly large number of consumers (or subscribers). In Publish-subscribe systems, an intermediary service efficiently ensures information generated by producers is routed to consumers who desire this information.

>*Message queues*: Whereas publish-subscribe systems offer a one-to-many style of communication, message queues offer a point-to-point service whereby producer processes can send messages to a specified queue and consumer processes can receive messages from the queue or be notified of the arrival of new messages in the queue.

>*Tuple spaces*: Tuple spaces offer a further indirect communication service by supporting a model whereby processes can place arbitrary items of structured data, called tuples, in a persistent tuple space and other processes can either read or remove such tuples from the tuple space by specifying patterns of interest. Since the tuple space is persistent, readers and writers do not need to exist at the same time.

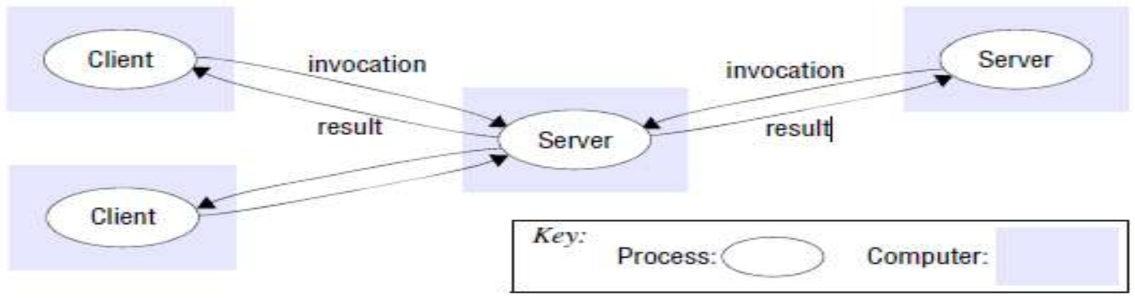
>*Distributed shared memory*: Distributed shared memory (DSM) systems provide an abstraction for sharing data between processes that do not share physical memory. Programmers are nevertheless presented with a familiar abstraction of reading or writing (shared) data structures as if they were in their own local address spaces, thus presenting a high level of distribution transparency.



Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

COMMUNICATING ENTITES AND COMMUNICATION PARADIGMS

→Roles and responsibilities:- In a distributed system processes – or indeed objects, components or services, including web services – interact with each other to perform a useful activity, for example, to support a chat session. we examine two architectural styles from the role of individual processes: **client-server and peer-to-peer.**

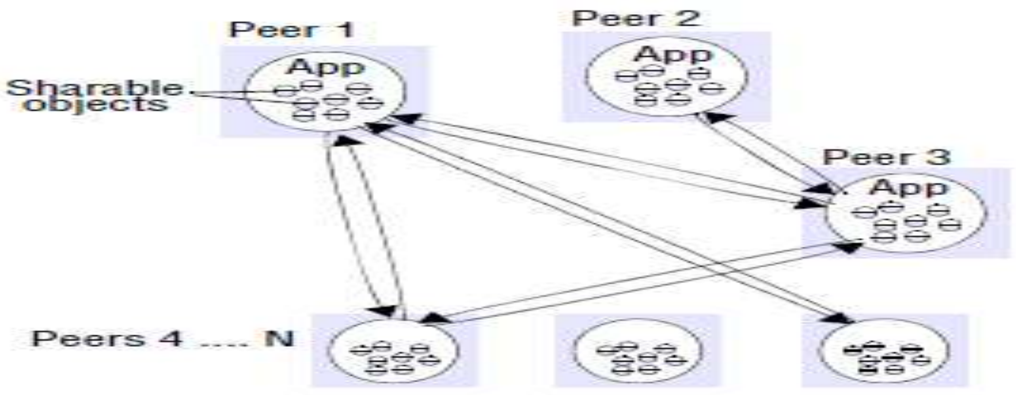


CLIENTS INVOKE INDIVIDUAL SERVERS

Client-server: Above diagram illustrates the simple structure in which processes take on the roles of being clients or servers. 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, such as web server is often a client of a local file server that manages the files in which the web pages are stored. Another web-related example is *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. In this example, the server tasks (responding to user queries) and the crawler tasks (making requests to other web servers) are entirely independent.

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. Sharing the computing and communication loads incurred in accessing them amongst a much larger number of computers and network links led to the development of Peer-to-peer systems. The aim of the peer-to-peer architecture is to exploit the resources (both data and hardware) in a large number of participating computers for the fulfillment of a given task or activity. Peer-to-peer applications and systems have been successfully constructed that enable tens or hundreds of thousands of computers to provide access to data and other resources that they collectively store and manage.



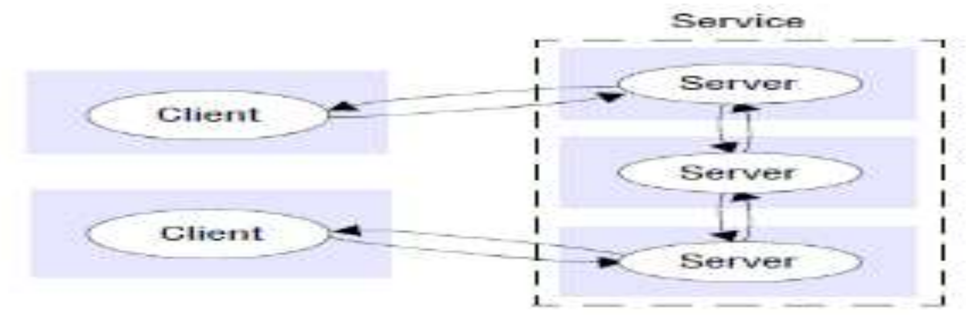
PEER-TO-PEER ARCHITECTURE

Above diagram illustrates the peer tot peer systems. 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. Each object is replicated in several computers to further distribute the load and to recover in the event of disconnection of individual computers. The need to place individual objects and retrieve them and to maintain replicas amongst many computers represents this architecture substantially more complex than the client-server architecture.

→Placement:- The final issue to be considered is how entities such as objects or services map on to the underlying physical distributed infrastructure which will consist of a potentially large number of machines interconnected by a network. Placement is crucial in terms of determining the properties of the distributed system, such as reliability and security. The question of where to place a given client or server in terms of machines and processes within machines is a matter of careful design. Placement needs to take into account the patterns of communication between entities, the reliability of given machines and their current loading, the quality of communication between different machines and so on. Placement must be determined with strong application knowledge, and there are few universal guidelines to obtaining an optimal solution. Placement strategies are:

- Mapping of services to multiple servers;
- caching;
- Mobile code;
- Mobile agents.

>Mapping of services to multiple servers: Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes (see given diagram). The servers may partition the set of objects on which the service is based and distribute those objects between themselves or they may maintain replicated copies on several hosts.

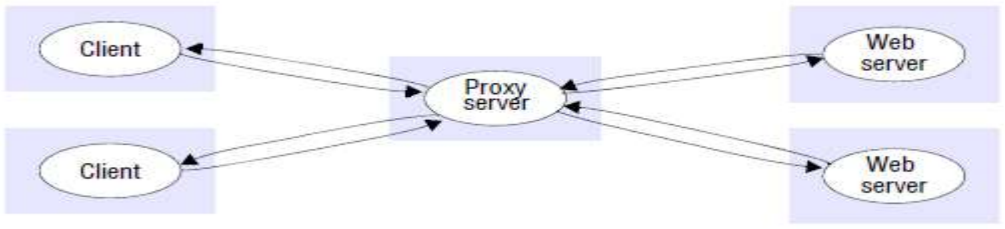


A SERVICE PROVIDED BY MULTIPLE SERVERS

Web is a common example for partitioned data. An example of a service based on replicated data is the Sun Network Information Service (NIS), which is used to enable all the computers on a LAN to access the same user authentication data when users log in. Each NIS server has its own replica of a common password file containing a list of users' login names and encrypted passwords.

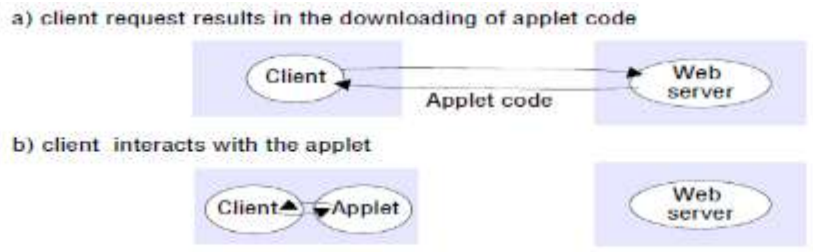
>Caching: A *cache* is a store of recently used data objects that is closer to one client or a particular set of clients. When a new object is received from a server it is added to the local cache store, replacing some existing objects if necessary. When an object is needed by a client process, the caching service first checks the cache and supplies the object from there if an up-to-date copy is available. If not, an up-to-date copy is fetched.

Web browsers maintain a cache of recently visited web pages and other web resources in the client's local file system, using a special HTTP request to check with the original server that cached pages are up-to date before displaying them.
Web proxy servers (below figure) provide a shared cache of web resources for the client machines at a site or across several sites. The purpose of proxy servers is to increase the availability and performance of the service by reducing the load on the wide area network and web servers. Proxy servers can take on other roles; for example, they may be used to access remote web servers through a firewall.



WEB PROXY SERVER

>Mobile code: Applets are a well-known and widely used example of mobile code – the user running a browser selects a link to an applet whose code is stored on a web server; the code is downloaded to the browser and runs there, as shown in below Figure . An advantage of running the downloaded code locally is that it can give good interactive response since it does not suffer from the delays or variability of bandwidth associated with network communication. Mobile code is a potential security threat to the local resources in the destination computer. Therefore browsers give applets limited access to local resources.


a) client request results in the downloading of applet code
Client
Web server
Applet code
b) client interacts with the applet
Client
Applet
Web server

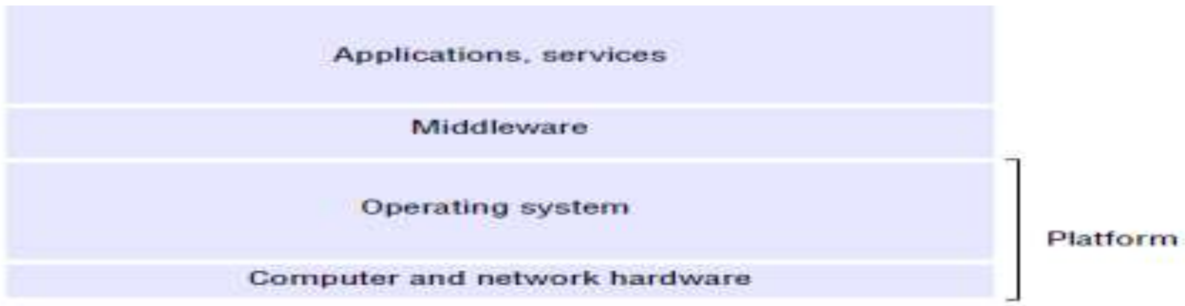
WEB APPLETS

>Mobile agents: A mobile agent is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf. Mobile agents (like mobile code) are a potential security threat to the resources in computers that they visit. The environment receiving a mobile agent should decide which of the local resources it should be allowed to use, based on the identity of the user on whose behalf the agent is acting – their identity must be included in a secure way with the code and data of the mobile agent. In addition, mobile agents can themselves be vulnerable – they may not be able to complete their task if they are refused access to the information they need. The tasks performed by mobile agents can be performed by other means. For example, web crawlers that need to access resources at web servers throughout the Internet work quite successfully by making remote invocations to server processes. For these reasons, the applicability of mobile agents may be limited.

2)Architectural patterns:-

Many architectural patterns have been identified for distributed systems. we present several key architectural patterns in distributed systems, including layering and tiered architectures and the related concept of thin clients.

Layering : The concept of layering is a familiar one and is closely related to abstraction. In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details. A distributed service can be provided by one or more server processes, interacting with each other and with client processes. For example, a network time service is implemented on the Internet based on the Network Time Protocol (NTP) by server processes running on hosts throughout the Internet that supply the current time to any client that requests it and adjust their version of the current time as a result of interactions with each other.



SOFTWARE AND HARDWARE SERVICE LAYERS IN DISTRIBUTED SYSTEM

Above figure introduces the important terms ***platform*** and ***middleware***,

- A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM are major examples.

- Middleware is defined as a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers. Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and resource-sharing support for distributed applications. In particular, it raises the level of the communication activities of application programs through the support of abstractions such as remote method invocation; communication between a group of processes; notification of events;

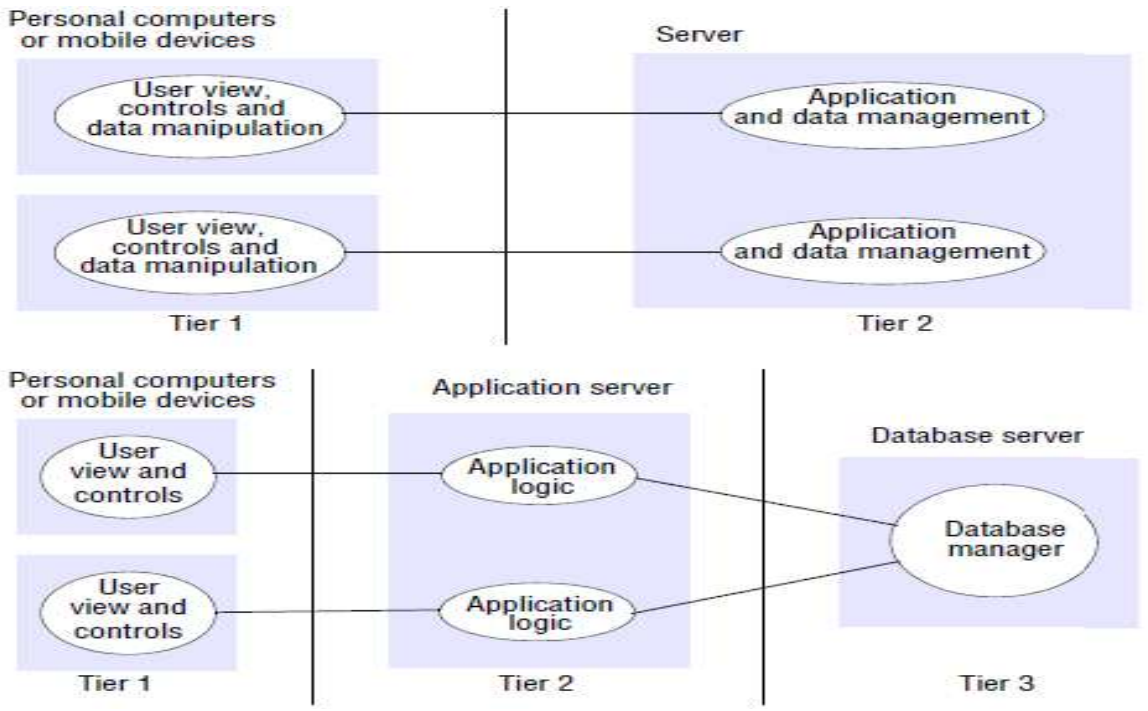
Tiered architecture : Tiered architectures are complementary to layering. Whereas layering deals with the vertical organization of services into layers of abstraction, tiering is a technique to organize functionality of a given layer and place this functionality into appropriate servers.
To illustrate this, consider the functional decomposition of a given application, as follows:

• The ***presentation logic***, which is concerned with handling user interaction and updating the view of the application as presented to the user;

• The ***application logic***, which is concerned with the detailed application-specific processing associated with the application;

• The ***data logic***, which is concerned with the persistent storage of the application, typically in a database management system.

Let us first examine the concepts of two- and three-tiered architecture.

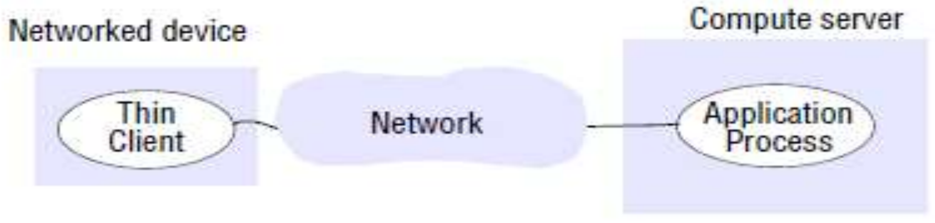

Personal computers or mobile devices
User view, controls and data manipulation
User view, controls and data manipulation
Tier 1
Server
Application and data management
Application and data management
Tier 2
Personal computers or mobile devices
User view and controls
User view and controls
Tier 1
Application server
Application logic
Application logic
Tier 2
Database server
Database manager
Tier 3

TWO TIER AND THREE TIER ARCHITECTURE

In the **two-tier solution**, the three aspects mentioned above must be partitioned into two processes, the client and the server. This is done by splitting the application logic, with some residing in the client and the remainder in the server. The advantage of this scheme is low latency in terms of interaction, with only one exchange of messages to invoke an operation. The disadvantage is the splitting of application logic across a process boundary, with the restriction on which parts of the logic can be directly invoked from which other part.

In the **three-tier solution**, there is a one-to-one mapping from logical elements to physical servers. Each tier also has a well-defined role; for example, the third tier is simply a database offering a relational service interface. The first tier can also be a simple user interface allowing intrinsic support for thin clients .The drawbacks are the added complexity of managing three servers and also the added network traffic and latency associated with each operation. This approach generalizes to n-tiered where a given application domain is partitioned into n logical elements. Wikipedia adopts multitier architecture.

Thin clients :The trend in distributed computing is towards moving complexity away from the end-user device towards services in the Internet. This trend has given rise to interest in the concept of a *thin client*, enabling access to sophisticated networked services, provided for example by a cloud solution, with few assumptions or demands on the client device. The term *thin client* refers to a software layer that supports a window-based user interface that is local to the user while executing application programs or, accessing services on a remote computer.



THIN CLIENTS AND COMPUTER SERVERS

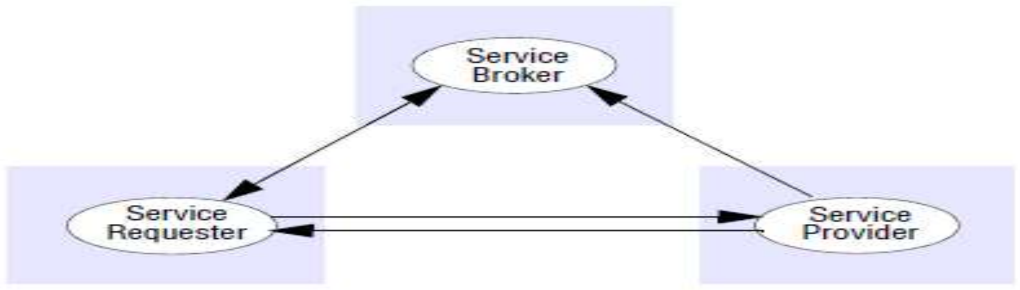
The advantage of this approach is that potentially simple local devices (including, for example, smart phones and other resource-constrained devices) can be significantly enhanced with a number of networked services and capabilities. The main drawback of the thin client architecture is in highly interactive graphical activities such as CAD and image processing, where the delays experienced by users are increased to unacceptable levels by the need to transfer image and vector information between the thin client and the application process, due to both network and operating system latencies.

This concept has led to the emergence of *virtual network computing* (VNC). The concept is straightforward, providing remote access to graphical user interfaces. In this solution, a VNC client (or viewer) interacts with a VNC server through a VNC protocol. Virtual network computing has superseded network computers and proved to be a more flexible solution and now dominates the market place.

Other commonly occurring patterns

• The ***proxy*** pattern is a commonly recurring pattern in distributed systems designed particularly to support location transparency in remote procedure calls or remote method invocation. With this approach, a proxy is created in the local address space to represent the remote object. This proxy offers exactly the same interface as the remote object, and the programmer makes calls on this proxy object and hence does not need to be aware of the distributed nature of the interaction.

• The use of ***brokerage*** in web services can usefully be viewed as an architectural pattern supporting interoperability in complex distributed infrastructures. This pattern consists of the trio of service provider, service requester and service broker (a service that matches services provided to those requested), as shown in below Figure.

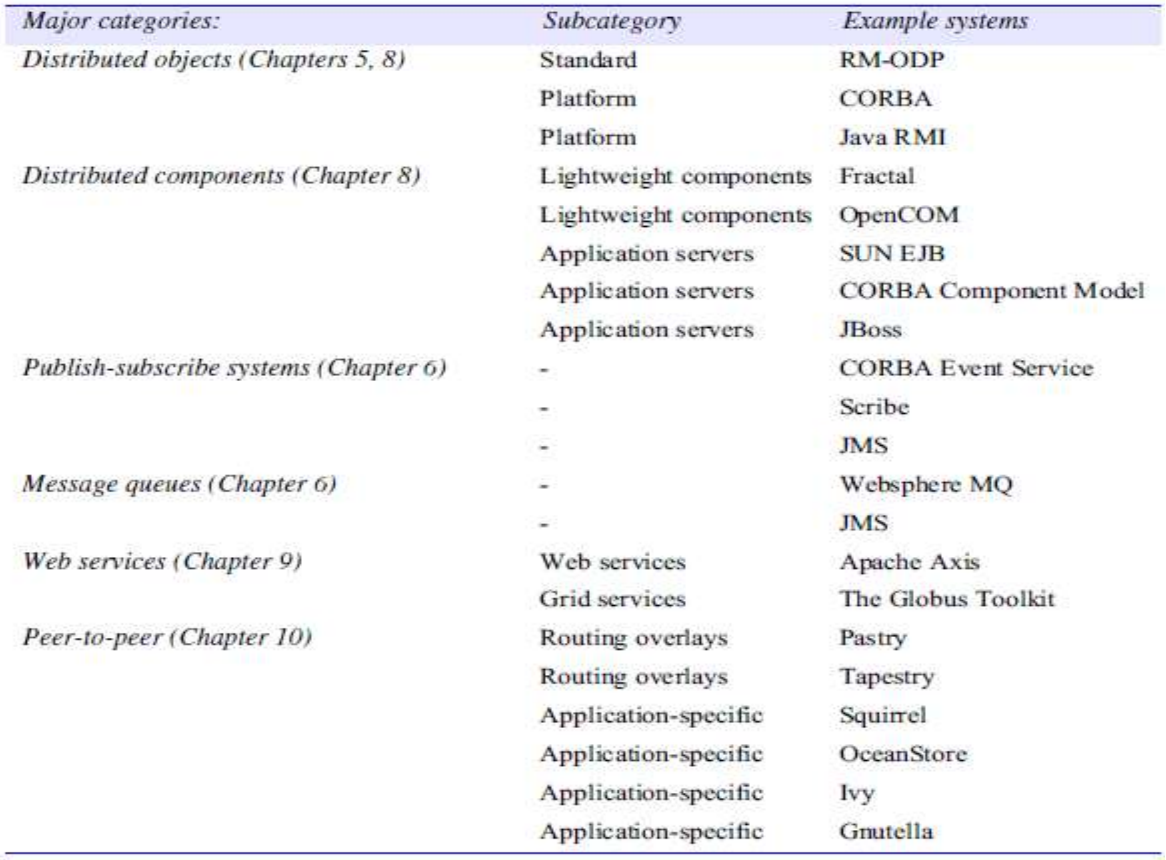


<u>THE WEB SERVICE ARCHITECTURE PATTERN</u>

• ***Reflection*** is a pattern that is increasingly being used in distributed systems as a means of supporting both introspection (the dynamic discovery of properties of the system) and intercession (the ability to dynamically modify structure or behavior).

<u>3)Associated middleware solutions:-</u>The task of middleware is to provide a higher-level programming abstraction for the development of distributed systems and, through layering, to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability.

Categories of middleware : Remote procedure calling packages such as **Sun RPC** and group communication systems such as **ISIS** were amongst the earliest instances of middleware. For example, many distributed object platforms offer distributed event services to complement the more traditional support for remote method invocation. Similarly, many component-based platforms also support web service interfaces and standards, for reasons of interoperability.



Major categories:	*Subcategory*	*Example systems*
Distributed objects (Chapters 5, 8)	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
Distributed components (Chapter 8)	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
Publish-subscribe systems (Chapter 6)	-	CORBA Event Service
	-	Scribe
	-	JMS
Message queues (Chapter 6)	-	Websphere MQ
	-	JMS
Web services (Chapter 9)	Web services	Apache Axis
	Grid services	The Globus Toolkit
Peer-to-peer (Chapter 10)	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

CATEGORIES OF MIDDLEWARE

The top-level categorization of middleware in above figure is driven by the choice of communicating entities and associated communication paradigms, and follows five of the main *architectural models*: distributed objects, distributed components, publish subscribe systems, message queues and web services.

In addition to programming abstractions, middleware can also provide infrastructural distributed system services for use by application programs or other services. These infrastructural services are tightly bound to the distributed programming model that the middleware provides. For example, CORBA provides applications with a range of CORBA services, including support for making applications secure and reliable.

➔Fundamental models:-

All of the models share the design requirements of achieving the performance and reliability characteristics of processes and networks and ensuring the security of the resources in the system. These models are based on the fundamental properties that allow us to be more specific about their characteristics and the failures and security risks they might exhibit.
The purpose of such a model is:
• To make explicit all the relevant assumptions about the systems we are modeling.
• To make generalizations concerning what is possible or impossible, given those assumptions.

The aspects of distributed systems that are captured in fundamental models are:

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.

***Failure*:** The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs or in the network that connects them. 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.

1) Interaction model:-

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 were the Domain Name System, which partitions and replicates its data at servers throughout the Internet and Sun Network Information Service, which keeps replicated copies of password files at several servers in a LAN.

• A set of peer processes may cooperate with one another to achieve a common goal: for example, a voice conferencing system that distributes streams of audio data in a similar manner, but with strict real-time constraints.

Most programmers will be familiar with the concept of an *algorithm* – a sequence of steps to be taken in order to perform a desired computation. Distributed systems composed of multiple processes such as those mentioned above are more complex. Their behavior and state can be described by a *distributed algorithm* – a definition of the steps to be taken by each of the processes of which the system is composed, *including the transmission of messages between them*. Messages are transmitted between processes to transfer information between them and to coordinate their activity.

The rate at which each process proceeds and the timing of the transmission of messages between them cannot be predicted. It is also difficult to describe all the states of a distributed algorithm, because it must deal with the failures of one or more of the processes involved or the failure of message transmissions.

We discuss 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: Communication over a computer network has the following performance characteristics relating to latency, bandwidth and jitter:

• The ***delay*** between the start of a message's transmission from one process and the beginning of its receipt by another is referred to as *latency*. The latency includes:

– The time taken for the first of a string of bits transmitted through a network to reach its destination.

– The delay in accessing the network, which increases significantly when the network is heavily loaded.

– The time taken by the operating system communication services at both the sending and the receiving processes, which varies according to the current load on the operating systems.

• The ***bandwidth*** of a computer network is the total amount of information that can be transmitted over it in a given time. When a large number of communication channels are using the same network, they have to share the available bandwidth.

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

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. Therefore two processes running on different computers can each associate timestamps with their events. However, even if the two processes read their clocks at the same time, their local clocks may supply different time values. This is because computer clocks drift from perfect time and, their drift rates differ from one another. The term *clock drift rate* refers to the rate at which a computer clock deviates from a perfect reference clock.

There are several approaches to correcting the times on computer clocks. For example, computers may use radio receivers to get time readings from the Global Positioning System (GPS) with an accuracy of about 1 microsecond. But GPS receivers do not operate inside buildings, nor can the cost be justified for every computer. Instead, a computer that has an accurate time source such as GPS can send timing messages to other computers in its network. The resulting agreement between the times on local clocks is affected by variable message delays.

Two variants of the interaction model: In a distributed system it is hard to set limits on the time that can be taken for process execution, message delivery or clock drift. Two models are there:

Synchronous distributed systems: synchronous distributed system is one in which the following bounds are defined:

• 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.

It is possible to suggest likely upper and lower bounds for process execution time, message delay and clock drift rates in a distributed system, but it is difficult to arrive at realistic values and to provide guarantees of the chosen values.

Asynchronous distributed systems: An asynchronous distributed system is one in which there are no bounds on:

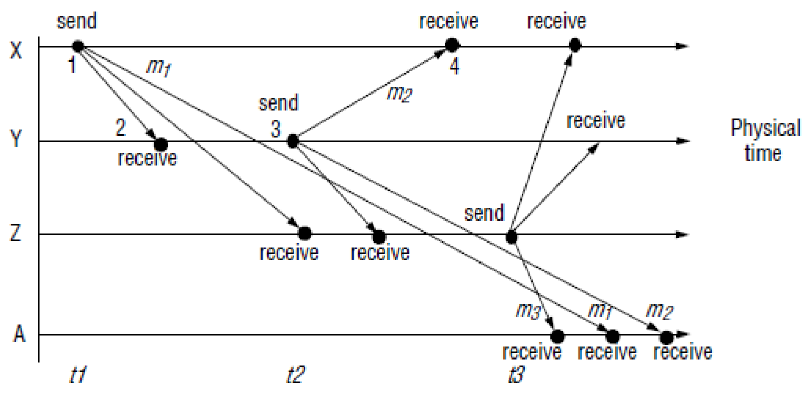
• Process execution speeds – for example, one process step may take only a picoseconds and another a century; all that can be said is that each step may take an arbitrarily long time.

• Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years.

• Clock drift rates – again, the drift rate of a clock is arbitrary.

The asynchronous model allows no assumptions about the time intervals involved in any execution. This exactly models the Internet, in which there is no intrinsic bound on server or network load and therefore on how long it takes, for example, to transfer a file using FTP. Sometimes an email message can take days to arrive.

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.
For example, consider the following set of exchanges between a group of email users, X, Y, Z and A, on a mailing list:
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 is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and 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 as shown in the figure below:



Real Time Ordering Events

If the clocks on X's, Y's and Z's computers could be synchronized, then each message could carry the time on the local computer's clock when it was sent. For example, messages *m1*, *m2* and *m3* would carry times *t1*, *t2* and *t3* where *t1*<*t2*<*t3*. The messages received will be displayed to users according to their time ordering.
Since clocks cannot be synchronized perfectly across a distributed system, Lamport proposed a model of Logical time that can be used to provide an ordering among the events running in different processes in a distributed system.
Logically, we know that a message is received after it was sent. Therefore we can state a logical ordering for pairs of events shown in Figure, for example, considering only the events concerning X and Y:
X sends *m1* before Y receives *m1*;

Y sends *m2* before X receives *m2*.

We also know that replies are sent after receiving messages, so we have the following logical ordering for Y:

Y receives *m1* before sending *m2*.

2)Failure model:-

In a distributed system both processes and communication channels may fail. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. The failure model gives a specification of the faults that can be exhibited by processes and communication channels. Several types of failures are
→Omission failures

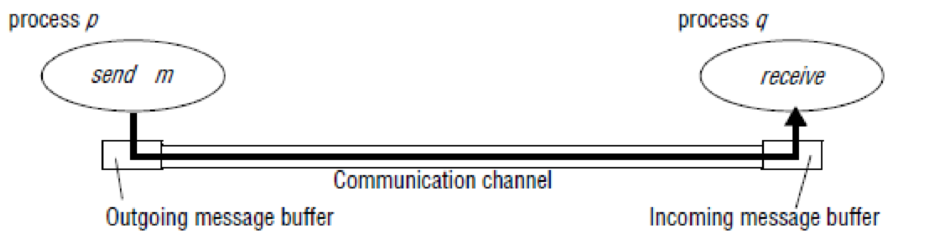
→Arbitrary failures
→Timing failures

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. We say that a process has crashed if it has halted and will not execute any further steps of its program ever. Processes may be able to detect such a crash by the fact that the process repeatedly fails to respond to invocation messages. 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 (see below figure). The outgoing and incoming message buffers are typically provided by the operating system.

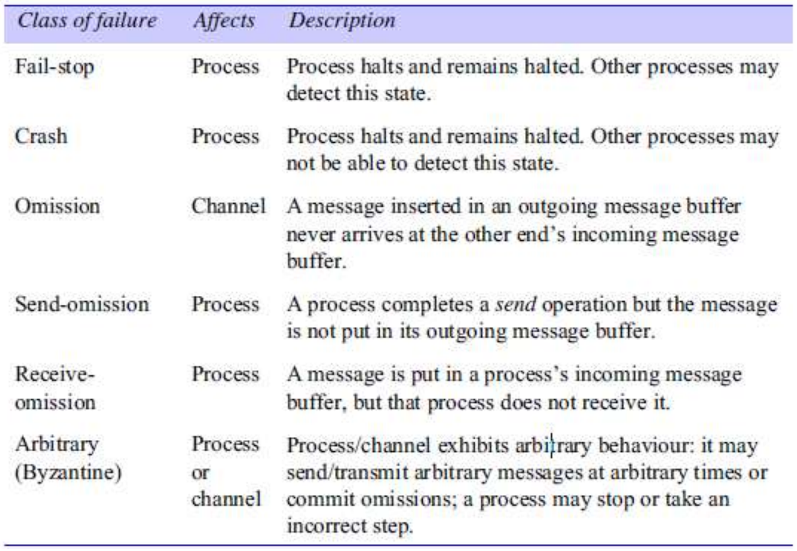


Process and channels

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, detected by a checksum carried with the message data. The loss of messages between the sending process and the outgoing message buffer are termed as ***send-omission failures***, the loss of messages between the incoming message buffer and the receiving process are termed as ***receive-omission failures***, and the loss of messages in between are termed 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. 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.

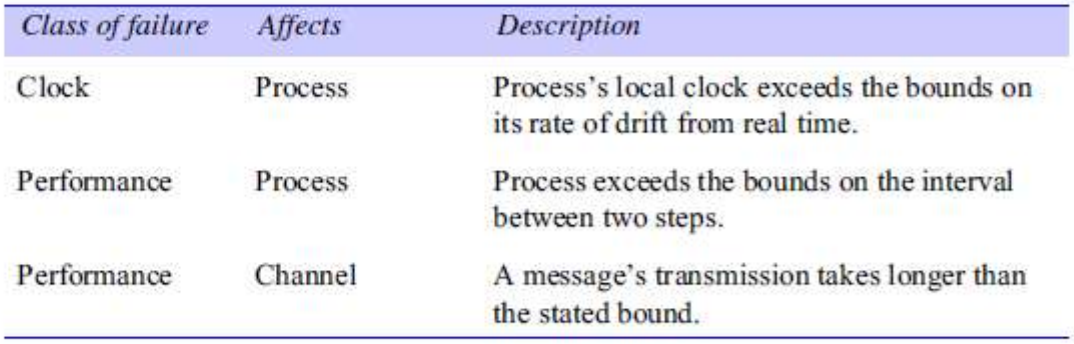


Class of failure	Affects	Description
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send* operation 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 or commit omissions; a process may stop or take an incorrect step.

<u>Omission and arbitrary failures</u>

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. In an asynchronous distributed system, an overloaded server may respond too slowly, but we cannot say that it has a timing failure since no guarantee has been offered.

Timing failures are listed as:



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.

<u>Timing failures</u>

Masking failures : Each component in a distributed system is generally constructed from a collection of other components. For example, multiple servers that hold replicas of data can continue to provide a service when one of them crashes. A knowledge of the failure characteristics of a component can enable a new service to be designed to mask the failure of the components on which it depends. A service *masks* a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For example, checksums are used to mask corrupted messages.

Reliability of one-to-one communication :

The term ***reliable communication*** is defined in terms of validity and integrity as follows:

> ***Validity***: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.
>
> ***Integrity***: The message received is identical to one sent, and no messages are delivered twice.

The threats to integrity come from two independent sources:

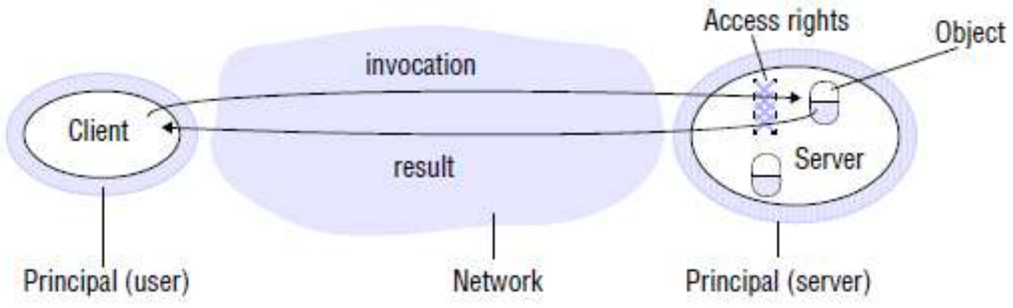
- • Any protocol that retransmits messages but does not reject a message that arrives twice. Protocols can attach sequence numbers to messages so as to detect those that are delivered twice.
- • Malicious users that may inject spurious messages, replay old messages or tamper with messages. Security measures can be taken to maintain the integrity property in the face of such attacks.

3)Security model:-

The architectural model provides the basis for 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.

Protecting objects: The following figure 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.



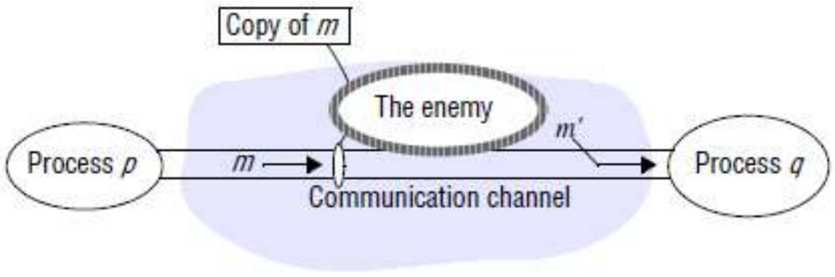
Objects and principals

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.

An authority is associated with each invocation and result. Such an authority is called a "***principal"***. A principal may be user or a process. 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.

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.

The enemy: To model security threats, we postulate an enemy 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 below Figure. 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.



The enemy

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 knowledge of source of a message is a threat to the correct functioning of both servers and clients as:

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

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.

→**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.

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. All these threats can be defeated by the use of *secure channels*, which are described below and are based on cryptography and authentication.

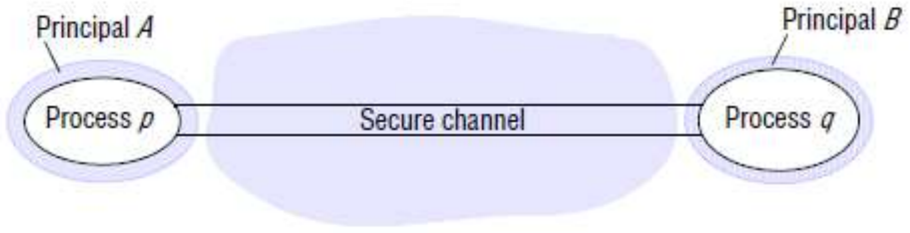
Defeating security threats:

Cryptography and shared secrets: Suppose that a pair of processes share a secret; that is, they both know the secret but no other process in the distributed system knows it. Then if a message exchanged by that pair of processes includes information that proves the sender's knowledge of the shared secret, the receipent knows for sure that sender was the other process in the pair.

"Cryptography" is the science of keeping messages secure, and **"*encryption*"** is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys.

Authentication: The use of shared secrets and encryption provides the basis for the *authentication* of messages – proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to guarantee its authenticity.

Secure channels: A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal. Encryption and authentication are used to build secure channels as a service layer on top of existing communication services.



Secure channels

A secure channel has the following properties:

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing. Therefore if a client and server communicate via a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation.
- A secure channel ensures the privacy and integrity of the data transmitted across it.
- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered.

Other possible threats from an enemy:

Denial of service: This is a form of attack in which the enemy interferes with the activities of authorized users by making excessive and pointless invocations on services or message transmissions in a network, resulting in overloading of physical resources (network bandwidth, server processing capacity).

***Mobile code*:** Mobile code raises new and interesting security problems for any process that receives and executes program code from elsewhere, such as the email attachment. Such code may easily play a Trojan horse role, purporting to fulfill an innocent purpose but in fact including code that accesses or modifies resources that are legitimately available to the host process.