

Department of Computer Science and Engineering

Lecture Notes



III B. Tech

Subject: Distributed Systems

Code: 80528

Academic Year 2020-21

Regulations: MR18

Malla Reddy Engineering College

(Autonomous)

(Approved by AICTE & Affiliated to JNTUH, Hyderabad)

Maisammaguda, Dhulapally(Post via Kompally) Secunderabad - 500100

Malla Reddy Engineering College (Autonomous)

L T P

3 - -

Course Code: 80528

Credits: 3

B.Tech.

DISTRIBUTED SYSTEMS

(Professional Elective)

Objective: To understand the Distributed computing Environment.

Module I: Basic Concepts Characterization of Distributed Systems – Examples – Resource Sharing and the Web – Challenges – System Models – Architectural and Fundamental Models – Networking and Internetworking – Types of Networks – Network Principles – Internet Protocols – Case Studies.
[09

Periods]

Module II: Processes and Distributed Objects A: IPC and its APIs Inter-process Communication – The API for the Internet Protocols – External Data Representation and Marshalling – Client –Server Communication – Group Communication – Case Study – Distributed Objects and Remote Invocation – Communication Between Distributed Objects – Remote Procedure Call – Events and Notifications – Java RMI – Case Study.
[09 Periods]

Module III: Operating System Issues I The OS Layer – Protection – Processes and Threads – Communication and Invocation – OS Architecture – Security – Overview – Cryptographic Algorithms – Digital Signatures – Cryptography Pragmatics – Case Studies – Distributed File Systems – File Service Architecture – Sun Network File System – The Andrew File System.
[09

Periods]

Module IV: Operating System Issues II] Name Services – Domain Name System – Directory and Discovery Services – Global Name Service – X.500 Directory Service – Clocks – Events and Process States – Synchronizing Physical Clocks – Logical Time And Logical Clocks – Global States – Distributed Debugging – Distributed Mutual Exclusion – Elections – Multicast Communication Related Problems.
[09

Periods

Module V: Distributed Transaction Processing Transactions – Nested Transactions – Locks – Optimistic Concurrency Control – Timestamp Ordering – Comparison – Flat and Nested Distributed Transactions – Atomic Commit Protocols – Concurrency Control in Distributed Transactions – Distributed Deadlocks – Transaction Recovery – Overview of Replication And Distributed Multimedia Systems [09 Periods]

Text Books: 1.George Coulouris, Jean Dollimore and Tim Kindberg, —Distributed Systems Concepts and Design, 3rd Edition, Pearson Education, 2002.

2.Andrew S. Tanenbaum, Maarten van Steen, Distributed Systems, —Principles and Paradigms, Pearson Education, 2002. References: 1.Sape Mullender, —Distributed Systems, 2nd Edition, Addison Wesley, 1993.

3.M. L. Liu, —Distributed Computing Principles and Applications, Pearson Education, 2004.

4.Mugesh Singhal, Niranjana G Shivaratri, —Advanced Concepts in Operating Systems, Tata McGraw Hill Edition, 2001.

MODULE- 1

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,

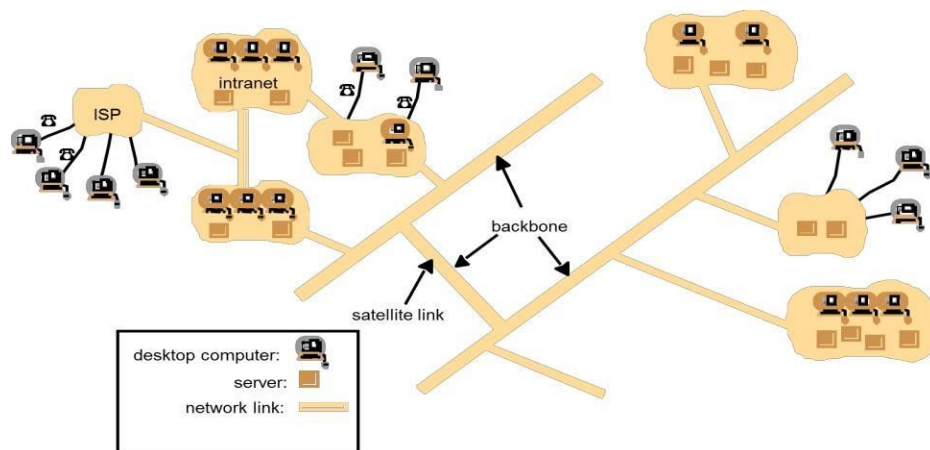
The Internet

Intranets

Mobile and Ubiquitous computing.

The Internet:

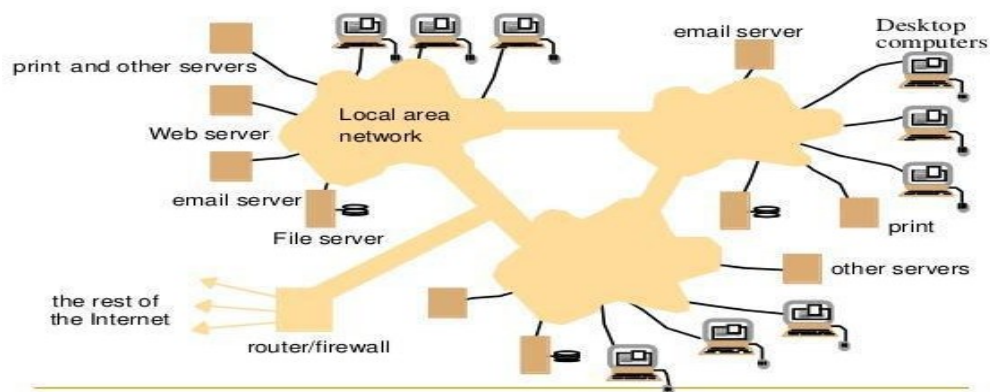
Internet is a very large distributed system. It enables users, wherever they are, to make use of services like www, email, file transfer. The set of services is open-ended. Refer figure below which shows a typical portion of internet. Internet connects millions of LANs and MANs to each other.



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.

Examples of Distributed Systems - A typical Intranet



- ✓ 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:



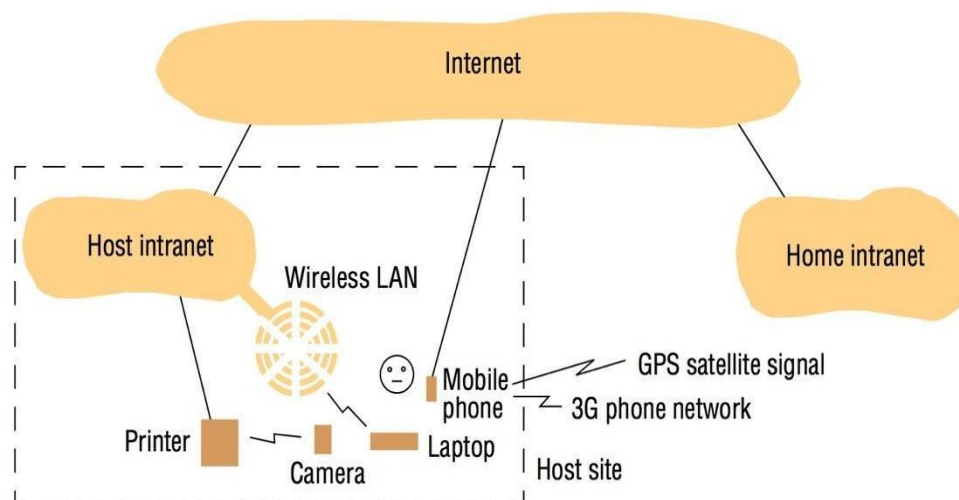
Integration of portable computing devices like Laptops, smartphones, handheld devices, pagers, digital cameras, smart watches, devices embedded in appliances like refrigerators, washing machines, cars etc. with the distributed systems became possible because of the technological advances in device miniaturization 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 look 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.downloadIt.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.

<i>Date</i>	<i>Computers</i>	<i>Web servers</i>	<i>Percentage</i>
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:

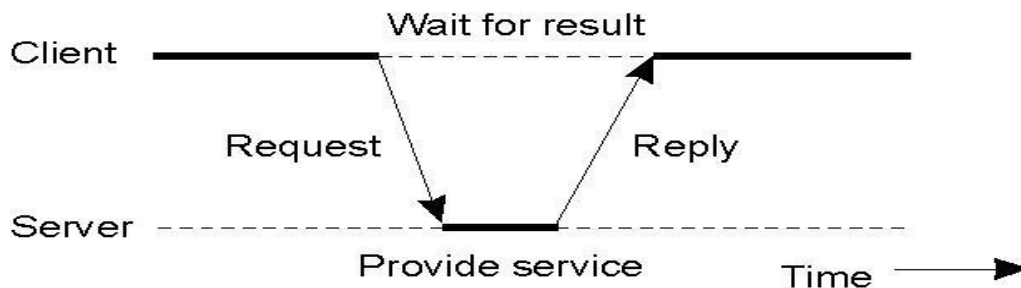
- Performance
- Reliability
- Security

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.

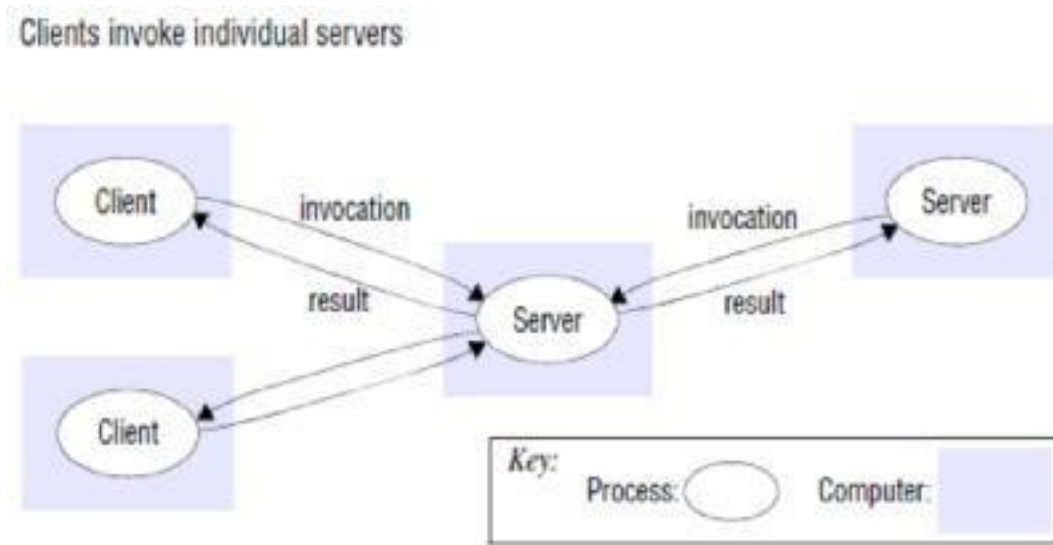
Client Server:

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

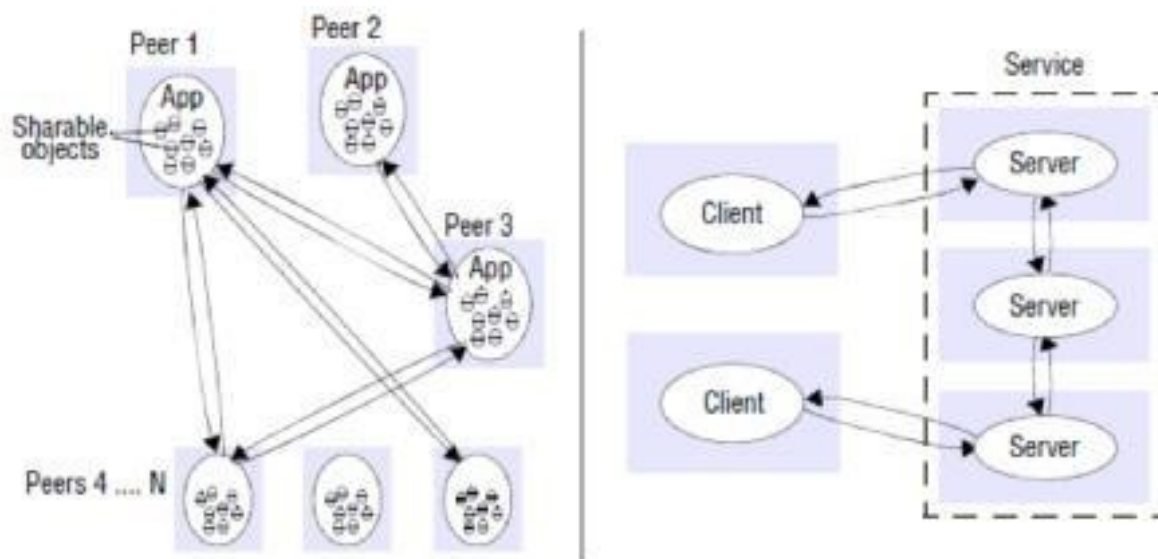
engine site using HTTP requests to access web servers throughout the Internet. Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers. In this example, the server tasks (responding to user queries) 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.

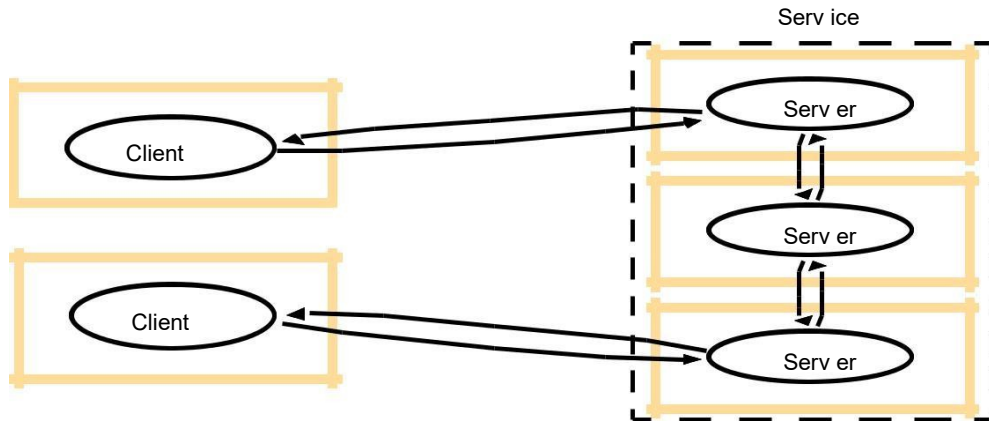
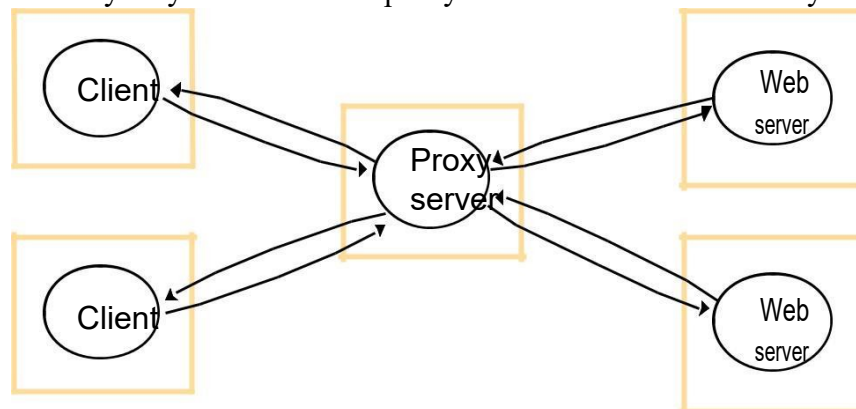


Figure. A service provided by multiple servers **Proxy servers and caches**

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-date 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.

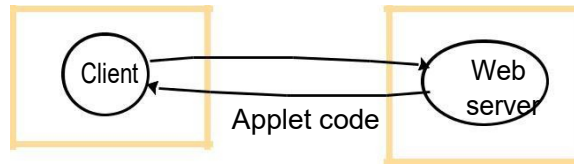


Mobile code

- 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

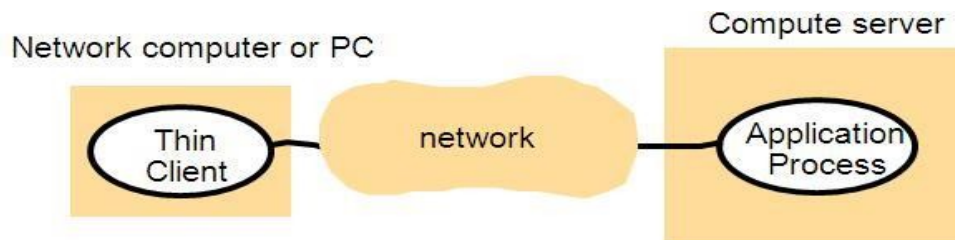
- A running program (code and data) that travels from one computer to another in a network carrying out of a task, usually on behalf of some other process.
- Examples of the tasks that can be done by mobile agents are:
 - ❖ To collecting information.
 - ❖ To install and maintain software maintain on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

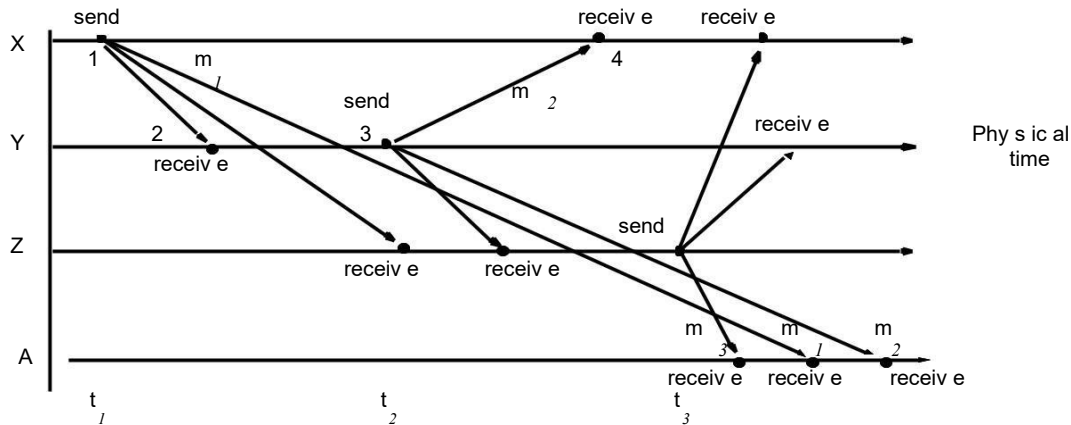


Figure. Real-time ordering of events

→ 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 m_1 before y receives m_1

Y sends m_2 (reply) before x receives m_2 (reply).

We also know that reply are send after receiving message.

Hence, we can say that,

Y receives m_1 before sending m_2 .

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.

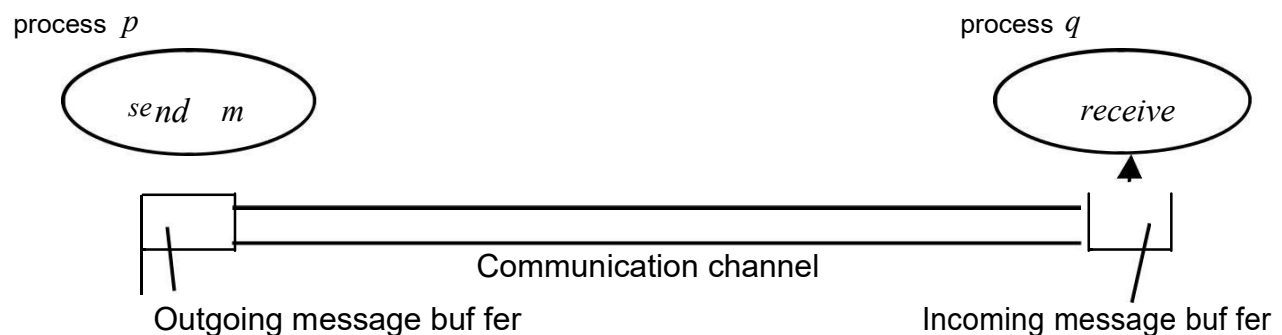
Process omission failures: The chief omission failure of a process is to crash. When we say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever.

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

<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
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 <i>send</i> , 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.

<i>Class of Failure</i>	<i>Affects</i>	<i>Description</i>
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

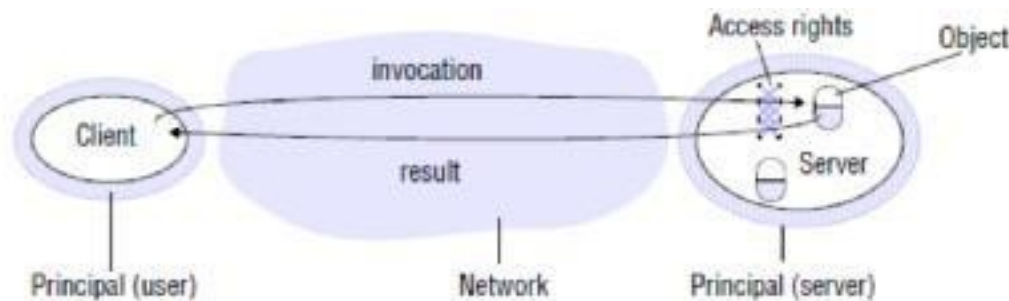
SECURITY MODEL

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.

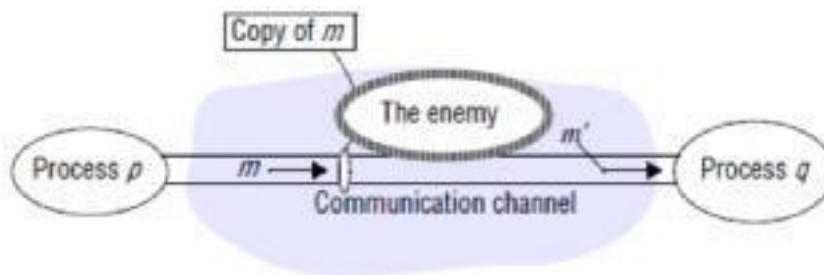
The server is responsible for verifying the identity of the principal behind each invocation and checking that they have sufficient access rights to perform the requested operation on the particular object invoked, rejecting those that do not. The client may check the identity of the principal behind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

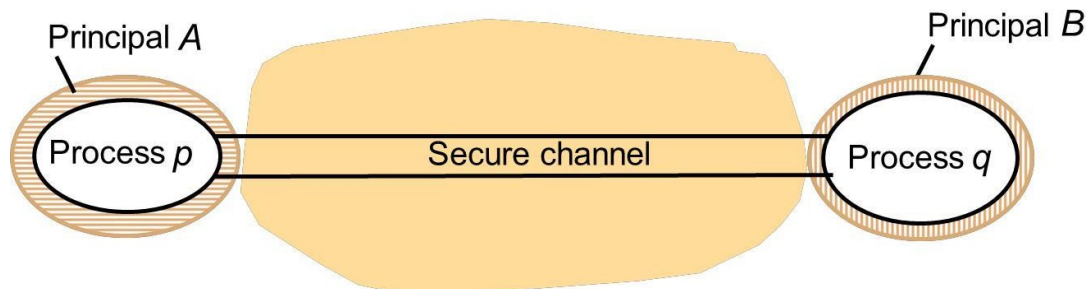
Threats to processes: A process that is designed to handle incoming requests may receive a message from any other process in the distributed system, and it cannot necessarily determine the identity of the sender. Communication protocols such as IP do include the address of the source computer in each 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. 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*.



MODULE- 2

Interprocess communication in the Internet provides both datagram and stream communication. The Java APIs for these are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols. This is complemented by a study of protocols for the representation of collections of data objects in messages and of references to remote objects.

Multicast is an important requirement for distributed applications and must be provided even if underlying support for IP multicast is not available. This is typically provided by an overlay network constructed on top of the underlying TCP/IP network. Overlay networks can also provide support for file sharing, enhanced reliability and content distribution.

The characteristics of interprocess communication:

Message passing between a pair of processes can be supported by two message communication operations, *send* and *receive*, defined in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

Synchronous and asynchronous communication • A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving processes may be either synchronous or asynchronous. In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both *send* and *receive* are *blocking* operations. Whenever a *send* is issued the sending process (or thread) is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued by a process (or thread), it blocks until a message arrives.

In the *asynchronous* form of communication, the use of the *send* operation is *nonblocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The *receive* operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

Μεσसाγε δεστινατιονς

→ A local port is a message destination within a computer, specified as an integer.

→ A port has an exactly one receiver but can have many senders.

Ρελιαβιλιτυ

→ A reliable communication is defined in terms of validity and integrity.

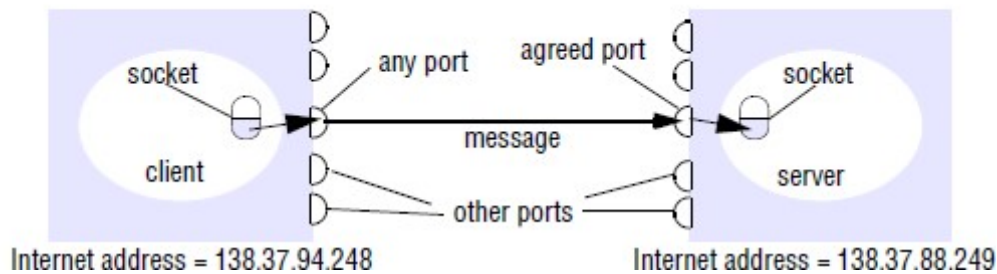
→ A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost.

→ For integrity, messages must arrive uncorrupted and without duplication.

Σοχκετς

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in Figure below.

Sockets and ports



For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number (216) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. (Processes using IP multicast are an exception in that they do share ports. However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

ΥΔΠ δαταγραμ χομμυνιχατιον

Datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process *sends* it and another *receives* it. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port. A

server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply. The following are some issues relating to datagram communication:

-        

The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of upto 2^{16} bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment the message into chunks of that size.

-        

Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication. The *send* operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination. On arrival, the message is placed in a queue for the socket that is bound to the destination port. The method *receive* blocks until a datagram is received, unless a timeout has been set on the socket. If the process that invokes the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread.

-        

The *receive* that blocks for ever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

-        

The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from.

Φαίλινε μνδελ φνρ ΥΔΠ δαταγρλμσ □

Reliable communication in terms of two properties: integrity and validity. The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams suffer from the following failures:

Omission failures: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination. To simplify the discussion, we regard send-omission and receive-omission failures as omission failures in the communication channel.

Ordering: Messages can sometimes be delivered out of sender order.

Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

Use of UDP • For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

- The need to store state information at the source and destination;
- The transmission of extra messages;
- Latency for the sender.

UDP clientsendsamessagetotheserverandgetsareply

```
import java.net.*;
import java.io.*;
public class UDPClient {
    public static void main(String args[]) {
        // argsgivemessagecontentsandserverhostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte[] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                new DatagramPacket(m, m.length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        } catch (SocketException) { System.out.println("Socket: " + e.getMessage()); }
        } catch (IOException) { System.out.println("IO: " + e.getMessage()); }
        } finally { if (aSocket != null) aSocket.close(); }
    }
}
```

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port. It also provides an no-argument constructor that allows the system to choose a free local port.

The class *DatagramSocket* provides methods that include the following:

send and receive: These methods are for transmitting datagrams between a pair of sockets. The argument of *send* is an instance of *DatagramPacket* containing a message and its destination. The argument of *receive* is an empty *DatagramPacket* in which to put the message, its length and its origin. The methods *send* and *receive* can throw *IOExceptions*.

setSoTimeout: This method allows a time out to be set. With a time out set, the *receive* method will block for the times specified and then throw an *InterruptedIOException*.

connect: This method is used for connecting to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDP_Server {
    public static void main(String args[]) {
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000]; while (true) {
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException) { System.out.println("Socket: " + e.getMessage()); }
        } catch (IOException) { System.out.println("IO: " + e.getMessage()); }
        } finally { if (aSocket != null) aSocket.close(); }
    }
}
```

ΤΧΠ στρεαμ χομμυνηατιον

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

• Η εφαρμογή μπορεί να γράφει ή να διαβάζει δεδομένα σε οποιοδήποτε σημείο του δικτύου.

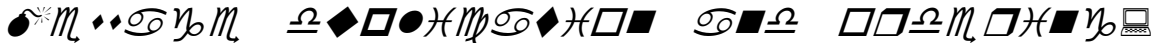
The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.

• Η εφαρμογή μπορεί να ελέγξει το ρυθμό μεταφοράς των δεδομένων.

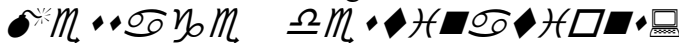
The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required



The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.



Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.



A pair of communicating processes establish a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a *connect* request from client to server followed by an *accept* request from server to client before any communication can take place.

Failure model • To satisfy the integrity property of reliable communication, TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets. For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets. Therefore, messages are guaranteed to be delivered even when some of the underlying packets are lost. But if the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

Use of TCP • Many frequently used services run over TCP connections, with reserved port numbers. These include the following:

HTTP: The Hypertext Transfer Protocol is used for communication between web browsers and web servers;

FTP: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

Telnet: Telnet provides access by means of a terminal session to a remote computer.

SMTP: The Simple Mail Transfer Protocol is used to send mail between computers.

JavaAPIforTCPstreams•TheJavainterface toTCPstreams is provided in the classes *ServerSocket* and *Socket*:

ServerSocket: This class is intended for use by a server to create a socket at a server port for listening for *connect* requests from clients. Its *accept* method gets a *connect* request from the queue or, if the queue is empty, blocks until one arrives. The result of executing *accept* is an instance of *Socket*—a socket to use for communicating with the client.

TCPclientmakesconnection toserver,sendsrequestandreceivesreply

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main(String args[]) {
        //argument supply message and hostname of destination
        Sockets = null;
        try {
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream());
            DataOutputStream out =
                new DataOutputStream(s.getOutputStream());
            out.writeUTF(args[0]); //UTF is a string encoding; see Sec 4.3
            String data = in.readUTF();
            System.out.println("Received: " + data);
        } catch (UnknownHostException) {
            System.out.println("Sock: " + e.getMessage());
        } catch (EOFException) {
            System.out.println("EOF: " + e.getMessage());
        } catch (IOException) {
            System.out.println("IO: " + e.getMessage());
        } finally {
            if (s != null) try { s.close(); } catch (IOException) { /*close failed*/ }
        }
    }
}
```

The **Socket** class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket. The return types of these methods are *InputStream* and *OutputStream*, respectively—abstract classes that define methods for reading and writing bytes.

TCP server makes a connection for each client and then echoes the client's request

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main(String args[]) {
        try {
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while (true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch (IOException) {
            System.out.println("Listen: " + e.getMessage());
        }
    }
}
```

```

class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection(Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream());
            out = new DataOutputStream(clientSocket.getOutputStream());
            this.start();
        } catch (IOException e) {
            System.out.println("Connection: " + e.getMessage());
        }
        public void run() {
            try {
                // anecho server
                String data = in.readUTF();
                out.writeUTF(data);
            } catch (EOFException e) {
                System.out.println("EOF: " + e.getMessage());
            } catch (IOException e) {
                System.out.println("IO: " + e.getMessage());
            } finally {
                try {
                    clientSocket.close();
                } catch (IOException e) {
                    /* close failed */
                }
            }
        }
    }
}

```

Εξωτερικά δεδομένα απεικονίζονται ως μαρσαλλισμένα

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures. There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last. Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character Coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character. One of the following methods can be used to enable any two computers to Exchange binary data values:

- The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.
- The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. **Unmarshalling** is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus marshalling consists of the translation of structured data items and Primitive values into an external data representation. Similarly, Unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

Three alternative approaches to external data representation and marshalling are discussed

- **CORBA's common data representation**, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages
- **Java's object serialization**, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.
- **XML (Extensible Markup Language)**, which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer.

In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually.

CORBA's Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0.

These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any*.

Primitive types: CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender's ordering, which is specified in each message. The recipient translates if it requires a different ordering. For example, a 16-bit *short* occupies two bytes in the message, and for big-endian ordering, the most significant bits occupy the first byte and the least significant bits occupy the second byte.

Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in following Figure.

CORBA CDR for constructed types

Type	Representation
<i>sequence</i>	length(unsigned long) followed by elements in order
<i>string</i>	length(unsigned long) followed by characters in order (can also have wide characters)
<i>array</i>	array elements in order (no length specified because it is fixed)
<i>struct</i>	in the order of declaration of the components
<i>enumerated</i>	unsigned long (the values are specified by the order declared)
<i>union</i>	type tag followed by the selected member

The following figure shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are *string*, *string* and *unsigned long*.

index in sequence of bytes	← 4 bytes →	notes on representation
0-3	5	length of string
4-7	"Smit"	'Smith'
8-11	"h__"	
12-15	6	length of string
16-19	"Lond"	'London'
20-23	"on__"	
24-27	1984	unsigned long

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Marshalling in CORBA • Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message.

For example, we might use CORBA IDL (Interface Definition language) to describe the data structure in the message of above Figure as follows:

```
struct Person {
    string name;
    string place;
    unsigned long year;
};
```

Θαπαοβρεχτσεριαλίζατιον

In Java RMI, both objects and primitive data values may be passed as arguments and result of method invocations. An object is an instance of a Java class. For example, the Java class equivalent to the *Person* struct defined in CORBA IDL might be:

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
```

```

        name=aName;
        place=aPlace; year=
        aYear;
    }
    //followed by methods for accessing the instance variables
}

```

- ✓ The above class states that it implements serializable interface.
- ✓ In java, serialization means flattening an object or a set of objects into a serial form suitable for storing on a disk or transmitting in a message.
- ✓ Deserialization is the restoring of object from serialized form.
- ✓ Information about class (like name, version etc.) are included in serializable form so that it is helpful in deserialization process.
- ✓ Version numbers is intended to change when major changes are made to the class. (usually set by programmer).
- ✓ To serialize an object, its class information is written out followed by the types and names of its instance variables.
- ✓ Each class is given a handle (reference to an object within serialized form).
- ✓ Example, consider serialization of following object
- ✓ Person p=new Person("Smith","London",1934);

Serialized values				Explanation
Person	8-byte version number		h0	class name, version number
3	int year	java.lang.String name	java.lang.String place	number, type and name of instance variables
1984	5 Smith	6 London	h1	values of instance variables

The true serialized form contains additional type markers; h0 and h1 are handles.

Extensible Markup Language (XML)

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term *markup language* refers to a textual encoding that represents both a text and details as to its structure or its appearance. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language.

→ XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures.

→ XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services.

→ XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them.

XML elements and attributes • The following Figure shows the XML definition of the *Person* structure that was used to illustrate marshalling in CORBA CDR and Java.

Φίγυρε XML δεφινιτιον οφ τηε Περσον στρυχτυρε

```
<person id="123456789">
<name>Smith</name>
<place>London</place>
<year>1984</year>
<!-- a comment -->
</person >
```

It shows that XML consists of tags and character data. The character data, for example *Smith* or *1984*, is the actual data. As in HTML, the structure of an XML document is defined by pairs of tags enclosed in angle brackets. In above Figure, *<name>* and *<place>* are both tags.

Elements: An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in Figure consists of the data *Smith* contained within the *<name> ... </name>* tag pair. Note that the element with the *<name>* tag is enclosed in the element with the *<person id="123456789"> ... </person >* tag pair.

Attributes: A start tag may optionally include pairs of associated attribute names and values such as *id="123456789"*, as shown above as attributes. An element is generally a container for data, whereas an attribute is used for labelling that data. In our example, *123456789* might be an identifier used by the application, whereas *name*, *place* and *year* might be displayed.

Names: The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon. The names continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case-sensitive. Names that start with *xml* are reserved.

Binary data: All of the information in XML elements must be expressed as character data.

XML namespaces • Traditionally, namespaces provide a means for scoping names. An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any other XML document can use an XML namespace by referring to its URL.

Any element that makes use of an XML namespace can specify that namespace as an attribute called *xmlns*, whose value is a URL referring to the file containing the namespace definitions. For example:

```
xmlns:pers = http://www.cdk5.net/person
```

The name after *xmlns*, in this case *pers* can be used as a prefix to refer to the elements in a particular namespace, as shown in following Figure. The *pers* prefix is bound to *http://www.cdk4.net/person* for the *person* element.

Ιλλυστρατιον οφ τηε υσε οφ α ναμεσπαχε ιν τηε Περσον στρυχτυρε

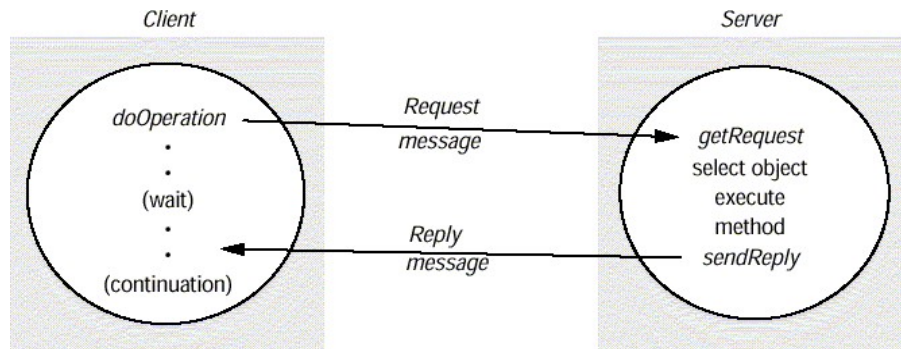
```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
<pers:name> Smith </pers:name>
<pers:place> London </pers:place>
<pers:year> 1984 </pers:year>
</person>
```

Client-Server Communication

The client-server communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.

The request-reply protocol

The request-reply protocol was based on a trio of communication primitives: `doOperation`, `getRequest`, and `sendReply` shown in following Figure.



The designed request-reply protocol matches requests to replies. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.

The following Figure outlines the three communication primitives.

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.

The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message *reply* to the client at its Internet address and port.

The information to be transmitted in a request message or a reply message is shown in following Figure.

messageType	<i>int (0=Request, 1=Reply)</i>
requestId	<i>int</i>
objectReference	<i>RemoteObjectRef</i>
methodId	<i>int or Method</i>
arguments	<i>// array of bytes</i>

The Request-reply protocol message structure contains the following.

- The first field indicates whether the message is a request or a reply message.
- The second field request id contains a message identifier.
- The third field is a remote object reference.
- The fourth field is an identifier for the method to be invoked.

Message identifier

A message identifier consists of two parts:

→ A `requestId`, which is taken from an increasing sequence of integers by the sending process

→ An identifier for the sender process, for example its port and Internet address.

Failure model of the request-reply protocol

If the three primitive *dooperation*, *getRequest*, and *sendReply* are implemented over UDP datagram, they have some communication failures. Such as,

→ omission failure

→ Messages are not guaranteed to be delivered in sender order.

RPC exchange protocols

Three protocols are used for implementing various types of RPC.

- The request (R) protocol.
- The request-reply (RR) protocol.
- The request-reply-acknowledge (RRA) protocol.

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

→ In the R protocol, a single request message is sent by the client to the server.

→ The R protocol may be used when there is no value to be returned from the remote method.

→ The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol. Special acknowledgement messages are not required, because a server reply message is considered as an acknowledgement of the client's request message.

→ RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. The acknowledgement reply message contains the requested from the reply message being acknowledged. This will enable the server to discard entries from its history.

HTTP: an example of a request-reply protocol

HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.

HTTP protocol steps are:

- Connection establishment between client and server at the default server port or at a port specified in the URL
- client sends a request message to the server
- server sends a reply message to the client
- connection is closed

HTTP request message is shown below.

<i>method</i>	<i>URL</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure :. HTTP request message

- **HTTP methods**

- **GET**

- Requests the resource, identified by URL as argument.
 - If the URL refers to data, then the web server replies by returning the data
 - If the URL refers to a program, then the web server runs the program and returns the output to the client.

- **HEAD**

- ❖ This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)

- **POST**

- ❖ Specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request.
 - ❖ This method is designed to deal with:
 - Providing a block of data to a data-handling process
 - Posting a message to a bulletin board, mailing list or news group.
 - Extending a dataset with an append operation

- **PUT**

- ❖ Supplied data to be stored in the given URL as its identifier.

- **DELETE**

- ❖ The server deletes an identified resource by the given URL on the server.

- **OPTIONS**

- ❖ A server supplies the client with a list of methods.
 - ❖ It allows to be applied to the given URL

- **TRACE**

- ❖ The server sends back the request message

HTTP reply message is shown below.

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

Above reply message specifies

- ❖ The protocol version
- ❖ A status code
- ❖ Reason
- ❖ Some headers
- ❖ An optional message body

Group Communication

The pairwise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary, for example, when a service is implemented as a number of different processes in different computers, perhaps to provide fault tolerance or to enhance availability. A *multicast operation* is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired Behaviour of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

- 1. *Fault tolerance based on replicated services:*** A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.
- 2. *Discovering services in spontaneous networking:*** Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
- 3. *Better performance through replicated data:*** Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value is multicast to the processes managing the replicas.
- 4. *Propagation of event notifications:*** Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers.

III μλτιχαστ □ Αν μπλεμεντατιον οφ μλτιχαστ χομμυνιχατιον

IP multicast • *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers – ports belong to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A *multicast group* is specified by a Class D Internet address.

Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send datagrams to a multicast group without being a member.

When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

Multicast routers: IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagrams to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called *time to live*, or TTL for short.

Multicast address allocation: Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA).

Failure model for multicast datagrams • Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. That is, some but not all of the members of the group may receive it. This can be called *unreliable* multicast, because it does not guarantee that a message will be delivered to any member of a group.

Java API to IP multicast • The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a specified local port (6789, in following figure) or any free local port. A process can join a multicast group with a given multicast address by invoking the *joinGroup()* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port. A process can leave a specified group by invoking the *leaveGroup()* method of its multicast socket.

Figure Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
public static void main(String args[]){
// args give message contents & destination multicast group (e.g. "228.5.6.7")
MulticastSocket s = null;
try {
InetAddress group = InetAddress.getByName(args[1]);
s = new MulticastSocket(6789);
s.joinGroup(group);
byte [] m = args[0].getBytes();
DatagramPacket messageOut =
new DatagramPacket(m, m.length, group, 6789);
s.send(messageOut);
byte[] buffer = new byte[1000];
for(int i=0; i< 3; i++) { // get messages from others in group
DatagramPacket messageIn =
new DatagramPacket(buffer, buffer.length);
s.receive(messageIn);
```

```

System.out.println("Received:" + new String(messageIn.getData()));
}
        eaveGroup(group);
} catch (SocketException e){System.out.println("Socket: " + e.getMessage());
} catch (IOException e){System.out.println("IO: " + e.getMessage());
} finally { if(s != null) s.close();}
}
}

```

Πελαβιλιτψ ανδ ορδερινγ οφ μλτιχαστ

A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message. Also, when a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.

Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so. Ordering is another issue. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members. In addition, messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

Some examples of the effects of reliability and ordering • We now consider the effect of the failure semantics of IP multicast as follows

1. ***Fault tolerance based on replicated services:*** Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.
2. ***Discovering services in spontaneous networking:*** One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services.
3. ***Better performance through replicated data:*** Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.
4. ***Propagation of event notifications:*** The particular application determines the qualities required of multicast.

Some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them. The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.

Distributed Objects and Remote Invocation:

Introduction communication between distributed objects

Distributed objects^[1] are objects that are distributed across different address spaces, either in multiple computers connected via a network or even indifferent processes on the same computer, but which work together by sharing data and invoking methods. This often involves location transparency, where remote objects appear the same as local objects.

The main method of distributed object communication is with remote method invocation

Invoking a method on a remote object is known as **remote method invocation**) generally by message-passing

Message-passing: one object sends a message to another object in a remote machine or process to perform some task. The results are sent back to the calling object.

The remote procedure call (RPC) approach extends the common programming Abstraction of the procedure call to distributed environments, allowing a calling Process to call a procedure in a remote node as if it is local.

Remote method invocation (RMI) is similar to RPC but for distributed objects, with Added benefits in terms of using object-oriented programming concepts in Distributed systems and also extending the concept of an object reference to the Global distributed environments, and allowing the use of object references as Parameters in remote invocations

Remote procedure call – client calls the procedures in a server program that is running in a different process

Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process

Event notification – objects receive notification of events at other objects for which they have registered

Middleware Roles

provide high-level abstractions such as RMI enable location transparency free from specifics of communication protocols

operating systems and communication hardware



Fig middle ware layer

Communication between distributed objects and other objects

Life cycle : Creation, migration and deletion of distributed objects is different from local objects

Reference : Remote references to distributed objects are more complex than simple pointers to memory addresses

Request Latency : A distributed object request is orders of magnitude slower than local method invocation

Object Activation : Distributed objects may not always be available to serve an object request at any point in time

Parallelism: Distributed objects may be executed in parallel.

Communication : There are different communication primitives available for distributed objects requests

Failure: Distributed objects have far more points of failure than typical local objects.

Security: Distribution makes them vulnerable to attack.

Distributed object model:

The term **distributed objects** usually refers to software modules that are designed to work together, but reside either in multiple computers connected via a network or in different processes inside the same computer.

Διστριβυτεδ οβφεχτσ

The state of an object consists of the values of its instance variables since object-based programs are logically partitioned, the physical distribution of objects into different

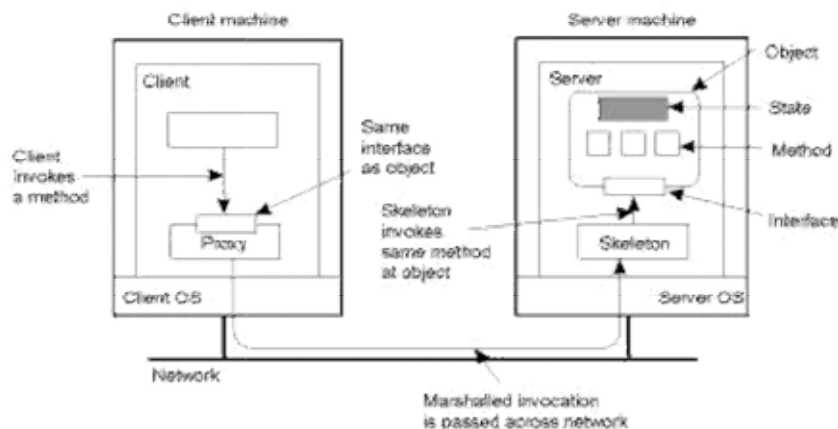
processes or computers in a distributed system. Distributed object systems may adopt the client-server architecture. objects are managed by servers and their clients invoke their methods using remote method invocation.

In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message.

Distributed objects can assume other architectural models. For example, objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance, and objects can be migrated with a view to enhancing their performance and availability.

Another advantage of treating the shared state of a distributed program as a collection of objects is that an object may be accessed via RMI, or it may be copied into a local cache and accessed directly, provided that the class implementation is available locally.

Distributed Objects



Δεσινγν ισσυεσ οφ PMI

RMI Invocation Semantics:

Invocation semantics depend upon implementation of Request Reply Protocol used by RMI

It maybe, used At-least-once, At-most-once

Transparency:

Partial failure, higher latency, Different semantics for remote objects,

For e.g. wait/notify Current consensus: remote invocations should be made transparent in the sense that syntax of a remote invocation is the same as the syntax of local invocation (access

transparency) but programmers should be able to distinguish between remote and local objects by looking at their interfaces, e.g. in Java RMI, remote objects implement the Remote interface

Issues in implementing RMI

Parameter passing

Request reply protocol (handling failures at client and server)

Supporting constant objects, object adapters, dynamic invocations, etc

The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI architects have succeeded; creating a system that extends the safety and robustness of the Java architecture to the distributed computing world.

The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service.

Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class.

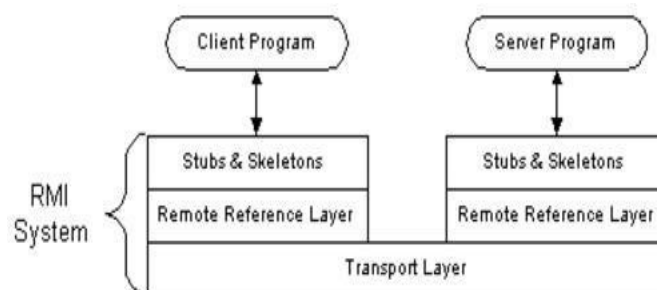
Therefore, the key to understanding RMI is to remember that interfaces define behavior and classes

Implementation of RMI:

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via *Remote Object Activation*.

The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



Distributed Garbage collection

Distributed garbage collection (DGC) in computing is a particular case of **garbage collection** where references to an object can be held by a remote client.

One of the joys of programming for the Java platform is not worrying about memory allocation. The JVM has an automatic garbage collector that will reclaim the memory from any object that has been discarded by the running program.

One of the design objectives for RMI was seamless integration into the Java programming language, which includes garbage collection. Designing an efficient single-machine garbage collector is hard; designing a distributed garbage collector is very hard.

The RMI system provides a reference counting distributed garbage collection algorithm based on Modula-3's Network Objects.

This system works by having the server keep track of which clients have requested access to remote objects running on the server. When a reference is made, the server marks the object as "dirty" and when a client drops the reference; it is marked as being "clean."

DGC uses some combination of the classical garbage collection (GC) techniques, tracing and reference counting. It has to cooperate with local garbage collectors in each process in order to keep global counts, or to globally trace accessibility of data.

In general, remote processors do not have to know about internal counting or tracing in a given process, and the relevant information is stored in interfaces associated with each process.

DGC is complex and can be costly and slow in freeing memory. One cheap way of avoiding DGC algorithms is typically to rely on a time lease set or configured on the remote object; it is the stub's task to periodically renew the lease on the remote object.

If the lease has expired, the server process (the process owning the remote object) can safely assume that either the client is no longer interested in the object, or that a network partition or crash obstructed lease renewal, in which case it is "hard luck" for the client if it is in fact still interested.

Hence, if there is only a single reference to the remote object on the server representing a remote reference from that client, that reference can be dropped, which will mean the object will be garbage collected by the local garbage collector on the server at some future point in time.

Distributed systems typically require distributed garbage collection. If a client holds a proxy to an object in the server, it is important that the server does not garbage-collect that object until the client releases the proxy. Most third-party distributed systems, such as RMI, handle the distributed garbage collection, but that does not necessarily mean it will be done efficiently. The overhead of distributed garbage collection and remote reference maintenance in RMI can slow network communications by a significant amount when many objects are involved.

Of course, if you need distributed reference maintenance, you cannot eliminate it, but you can reduce its impact. You can do this by reducing the number of temporary objects that may have

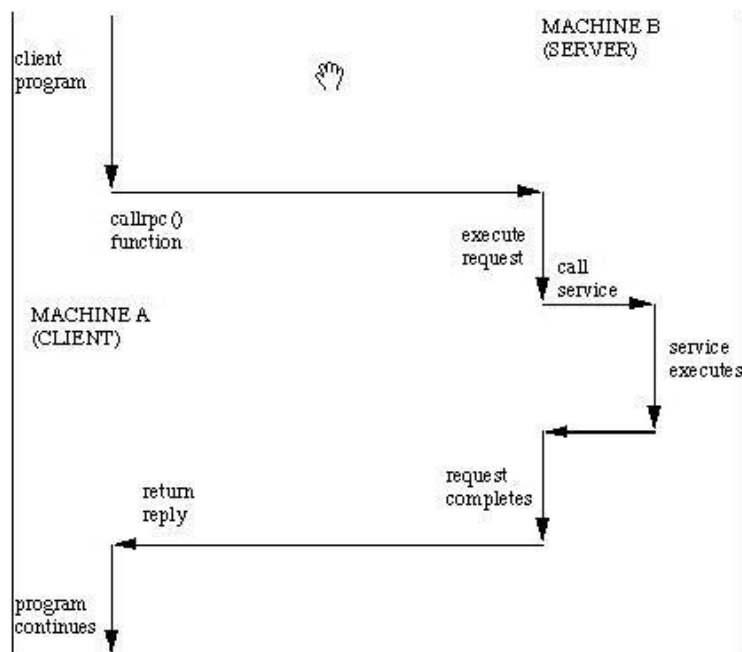
distributed references. The issue is considerably more complex in a multiuser distributed environment, and here you typically need to apply special optimizations related to the products you use in order to establish your multiuser environment.

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. (A procedure call is also sometimes known as a function call or a subroutine call.) **RPC** uses the client/server model.

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

The flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out.

When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.



Remote Procedure Calling Mechanism A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) the program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously. Each version contains a number of procedures that can be called remotely. Each procedure has a procedure number.

Events and notification

- Events of changes/updates...

- notifications of events to parties interested in the events

- publish events to send

- subscribe events to receive

main characteristics in distributed event-based systems:

- a way to standardize communication in heterogeneous systems (not designed to communicate directly)

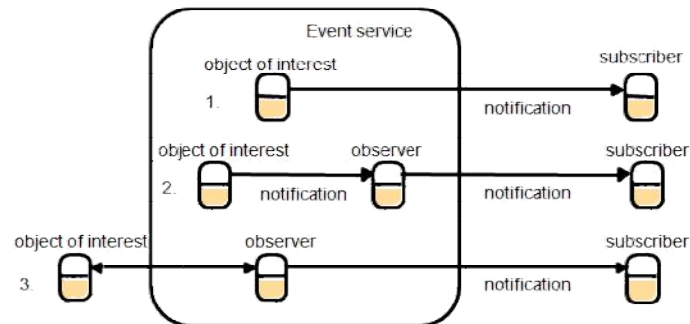
- asynchronous communication (no need for a publisher to wait for each subscriber-
- subscribers come and go)

event types

- each type has attributes (information in it)

- subscription filtering: focus on certain values in the attributes (e.g. "buy" events, but only "buy car" events)

Events and Notifications (4):



Publish-subscribe paradigm: publisher sends notifications, i.e. objects representing events

◁ Subscriber registers interest to receive notifications

The object of interest: where events happen, change of state as a result of its operations being invoked ,, Events: occurs in the object of interest ,,

Notification: an object containing information about an event

Subscriber: registers interest and receives notifications ,,

publisher: generate notifications, usually an object of interest

Observer objects: decouple an object of interest from its subscribers (not important)

Case study JAVA RMI

server program main program: binding instances of servant classes

main method needs to create a security manager to enable Java security. A default security manager, `RMISecurityManager`, is provided

Note: if an RMI server sets no security manager, proxies and classes can only be loaded from the local classpath, in order to protect the program from code that is downloaded as a result of remote method invocations.

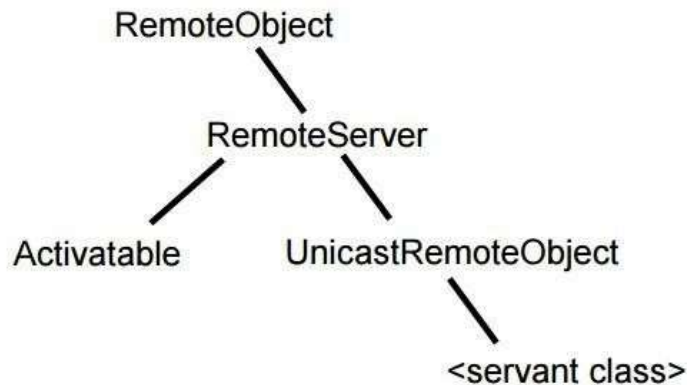
servant classes: ShapeList Servant and Shape Servant, implementing ShapeList and Shape interfaces respectively < servant classes need to extend

UnicastRemoteObject, which provides remote object that live only as long as the process in which they are created

implementation of servant classes are straightforward, no concern of communication details

Classes supporting Java RMI:

Every single servant class needs to extend UnicastRemoteObject



UnicastRemote Object:

automatically creates socket and listens for network requests, and make its services available by exporting them.

RMISecurityManager (): Needed to download objects from network. The downloaded objects are allowed to communicate only with sites they came from.

Default security manager, when none is explicitly set, allows only loading from local file system

Reflection: the class of an object can be determined at runtime, and this class can be examined to determine which methods are available, and even invoke these methods with dynamically created arguments ,,

The key to reflection is the java.lang.Class, which allows much information to be determined about a class. This leads onto the other reflection classes such as java.lang.reflect.Method

Heterogeneity is an important challenge to designers: < Distributed systems must be constructed from a variety of different networks, operating systems, computer hardware and programming languages. The Internet communication protocols mask the difference in networks and middleware can deal with the other differences. ,,

External data representation and marshalling <

CORBA marshals data for use by recipients that have prior knowledge of the types of its components. It uses an IDL specification of the data types <

Java serializes data to include information about the types of its contents, allowing the recipient to reconstruct it. It uses reflection to do this. „, RMI <

Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely. <

local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once <

Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers

MODULE- 3

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. File systems provide directory services, which convert a file name (possibly a hierarchical one) into an internal identifier (e.g. inode, FAT index). They contain a representation of the file data itself and methods for accessing it (read/write). The file system is responsible for controlling access to the data and for performing low-level operations such as buffering frequently used data and issuing disk I/O requests.

A distributed file system is to present certain degrees of transparency to the user and the system: **Access transparency:** Clients are unaware that files are distributed and can access them in the same way as local files are accessed.

Location transparency: A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.

Concurrency transparency: All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.

Failure transparency: The client and client programs should operate correctly after a server failure.

Heterogeneity: File service should be provided across different hardware and operating system platforms.

Scalability: The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).

Replication transparency: To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.

Migration transparency: Files should be able to move around without the client's knowledge. Support fine-grained distribution of data: To optimize performance, we may wish to locate individual objects near the processes that use them.

Tolerance for network partitioning: The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

File service types: To provide a remote system with file service, we will have to select one of two models of operation. One of these is the upload/download model. In this model, there are two fundamental operations: *read file* transfers an entire file from the server to the requesting client, and *write file* copies the file back to the server. It is a simple model and efficient in that it provides local access to the file when it is being used. Three problems are evident. It can be updated. The file service itself provides the file interface (this is mentioned above). Another

wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file. Finally, what happens if others need to modify the same file?

The second model is a remote access model. The file service provides remote operations such as *open*, *close*, *read bytes*, *write bytes*, *get attributes*, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it.

Another important distinction in providing file service is that of understanding the difference between *directory service* and *file service*. A directory service, in the context of file systems, maps human-friendly textual names for files to their internal locations, which can be used by the

component of file distributed file systems is the client module. This is the client-side interface for file and directory service. It provides a local file system interface to client software (for example, the node file system layer of a UNIX kernel).

File service architecture • This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

SUN NFS • Sun Microsystems's *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version.

File Service Architecture: Flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.

Διρεχτορσι σερπιχε:

Provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.

Χλιεντ μολυλε:

It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations. It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.

Φλατ φιλε σερπιχε ιντερφαχε:

Access control:

In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.

Ηιεραρχηιχ φιλε σψστεμ:

A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.

File Group

A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.

- A similar construct is used in a UNIX file system.
- It helps with distributing the load of file serving between several servers.
- File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).

To construct globally unique ID we use some unique attribute of the machine on which it is

Smartworld.com component of file distributed file systems is the client module. This is the client-side interface for Smartworld.asia created. E.g: IP number, even though the file group may move subsequently.

In the local file system they are checked only on the file's access permission attribute.
Every client request is accompanied by the userID and groupID

It is not shown in the Figure 8.9 because they are inserted by the RPC system.

Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

Mount service

Mount operation:

mount(remotehost, remotedirectory, localdirectory)

Τηλε Ανδρεω Φιλε Συστημ (ΑΦΣ)

AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

Whole-file caching: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' *open* requests in preference to remote copies whenever possible.

Scenario • Here is a simple scenario illustrating the operation of AFS:

When a user process in a client computer issues an *open* system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file. The copy is stored in the local UNIX file system in the client computer. The copy is then *opened* and the resulting UNIX file descriptor is returned to the client.

Ναμινγ Σερπιχεσ

What are Naming Services?

- An URL facilitates the localization of a resource exposed on the Web.
- A consistent and uniform naming helps processes in a distributed system to interoperate and manage resource.
- Users refers to each other by means of their names (i.e. email) rather than their system ids– Naming Services are not only useful to locate resources but also to gather additional information about them such as attributes

Ωηατ αρε Ναμινγ Σερπιχεσ?

In a Distributed System, a Naming Service is a specific service whose aim is to provide a consistent and uniform naming of resources, thus allowing other programs or services to localize them and obtain the required metadata for interacting with them.

Κεψ βενεφιτς:

- Resource localization
- Uniform naming
- Device independent address (e.g., you can move domain name/web site from one server to another server seamlessly).

Τηε ρολε οφ ναμεσ ανδ ναμε σερπιχεσ

- An identifier can be stored in variables and retrieved from tables quickly
- Identifier includes or can be transformed to an address for an object
- A name is human-readable value (usually a string) that can be *resolved* to an identifier or address
- because the binding of the named resource to a physical location is deferred and can be changes
- because they are more meaningful to users
- to give identifiers and other useful attributes

Ρεθυριμεντς φορ ναμε σπαχεσ

Allow simple but meaningful names to be used

- to allow similar sub names without clashes
- to group related names

The DNS maps domain names to the attributes of a host computer: its IP address, the type of entry (for example, a reference to a mail server or another host) and, for example, the length of time the host's entry will remain valid. The X500 directory service can be used to map a person's name onto attributes including their email address and telephone number. The CORBA Naming Service maps the name of a remote object onto its remote object reference, whereas the Trading Service maps the name of a remote object onto its remote object reference, together with an arbitrary number of attributes describing the object in terms understandable by human users.

Να με Σερπιγες ανδ της Δομαιν Να με Σψστεμ

A name service stores a collection of one or more naming contexts, sets of bindings between textual names and attributes for objects such as computers, services, and users.

The major operation that a name service supports is to resolve names.

Υνιφορμ Ρεσουρχε Ιδεντιφιερσ

Uniform Resource Identifiers (URIs) came about from the need to identify resources on the Web, and other Internet resources such as electronic mailboxes. An important goal was to identify resources in a coherent way, so that they could all be processed by common software such as browsers. URIs are ‘uniform’ in that their syntax incorporates that of indefinitely many individual types of resource identifiers (that is, URI schemes), and there are procedures for managing the global namespace of schemes. The advantage of uniformity is that it eases the process of introducing new types of identifier, as well as using existing types of identifier in new contexts, without disrupting existing usage.

Uniform Resource Locators: Some URIs contain information that can be used to locate and access a resource; others are pure resource names. The familiar term Uniform Resource Locator (URL) is often used for URIs that provide location information and specify the method for accessing the resource.

Υνιφορμ Ρεσουρχε Ναμεσ:

Uniform Resource Names (URNs) are URIs that are used as pure resource names rather than locators. For example, the URI:

mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com

Ναπιγατιον

Navigation is the act of chaining multiple Naming Services in order to resolve a single name to the corresponding resource.

Namespaces allows for structure in names.

URLs provide a default structure that decompose the location of a resource in

- protocol used for retrieval
- internet end point of the service exposing the resource
- service specific path

This decomposition facilitates the resolution of the name into the corresponding resource

Moreover, structured namespaces allows for iterative navigation...

Ιτερατιωε ναπιγατιον

Reason for NFS iterative name resolution

This is because the file service may encounter a symbolic link (i.e. an *alias*) when resolving a name. A symbolic link must be interpreted in the client’s file system name space because it may point to a file in a directory stored at another server. The client computer must determine which server this is, because only the client knows its mount points

Σερπερ χοντρολλεδ ναπιγατιον

In an alternative model, name server coordinates naming resolution and returns the results to the client. It can be:

- **Recursive:** it is performed by the naming server the server becomes like a client for the next server this is necessary in case of client connectivity constraints
- **Non recursive:** it is performed by the client or the first server the server bounces back the

Smartzworld.com

A name service stores a collection of one or more naming contexts, sets of bindings between

Smartworld.asia

jntuworldupdates.org

next hop to its client

Specworld.in

Νον-ρεχυρσιτωε ανδ ρεχυρσιτωε σερωερ-χοντρολλεδ ναπιγατιον

DNS offers recursive navigation as an option, but iterative is the standard technique. Recursive navigation must be used in domains that limit client access to their DNS information for security reasons.

The Domain Name System is a name service design whose main naming database is used across the Internet.

This original scheme was soon seen to suffer from three major shortcomings:

It did not scale to large numbers of computers. Local organizations wished to administer their own naming systems. A general name service was needed – not one that serves only for looking up computer addresses.

Domain names • The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were:

ΔΝΣ – Τηε Ιντερνετ Δομαιν Ναμε Συστεμ

A distributed naming database (specified in RFC 1034/1305 Name structure reflects administrative structure of the Internet.

Rapidly resolves domain names to IP addresses

- exploits caching heavily
- typical query time ~100 milliseconds

Scales to millions of computers

- partitioned database
- caching

Resilient to failure of a server

- Replication

ΔΝΣ σερωερ φυνχτιονσ ανδ χονφιγυρατιον

Main function is to resolve domain names for computers, i.e. to get their IP addresses

- caches the results of previous searches until they pass their 'time to live'

Other functions:

- get *mail host* for a domain
- reverse resolution - get domain name from IP address
- Host information - type of hardware and OS
- Well-known services - a list of well-known services offered by a host
- Other attributes can be included (optional)

ΔΝΣ ρεσουρχε ρεχορδσ

The DNS architecture allows for recursive navigation as well as iterative navigation. The resolver specifies which type of navigation is required when contacting a name server. However, name servers are not bound to implement recursive navigation. As was pointed out above, recursive navigation may tie up server threads, meaning that other requests might be delayed.

Γλοβαλ Ναμε Σερωιχε (ΓΝΣ)

The GNS manages a naming database that is composed of a tree of directories holding names and values. Directories are named by multi-part pathnames referred to a root, or relative to a

working directory, much like file names in a UNIX file system. Each directory is also assigned

and references. The values stored at the leaves of the directory tree are organized into *value trees*, so that the attributes associated with names can be structured values. Names in the GNS have two parts: *<directory name, value name>*. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree.

Χονσενσους ανδ ρελατεδ προβλεμς

- Problems of agreement
 - For processes to agree on a value (consensus) after one or more of the processes has proposed what that value should be
 - Covered topics: byzantine generals, interactive consistency, totally ordered multicast
- The byzantine generals problem: a decision whether multiple armies should attack or retreat, assuming that united action will be more successful than some attacking and some retreating
- Another example might be space ship controllers deciding whether to proceed or abort. Failure handling during consensus is a key concern

Assumptions

- communication (by message passing) is reliable
- processes may fail
- Sometimes up to f of the N processes are faulty

Consensus Process

Each process p_i begins in an undecided state and proposes a single value v_i , drawn from a set D ($i=1 \dots N$)

Processes communicate with each other, exchanging values

Each process then sets the value of a decision variable d_i and enters the decided state

Requirements for Consensus

- Three requirements of a consensus algorithm
 - **Termination**: Eventually every correct process sets its decision variable
 - **Agreement**: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the *decided* state, then $d_i=d_j$ ($i,j=1,2, \dots, N$)
 - **Integrity**: If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value

The byzantine generals problem

- Problem description
 - Three or more generals must agree to *attack* or to *retreat*
 - One general, the *commander*, issues the order
 - Other generals, the *lieutenants*, must decide to attack or retreat
 - One or more generals may be treacherous
- A *treacherous general* tells one general to attack and another to retreat
- Difference from consensus is that a single process supplies the value to agree on

Requirements

- **Termination**: eventually each correct process sets its decision variable
- **Agreement**: the decision variable of all correct processes is the same
- **Integrity**: if the commander is correct, then all correct processes agree on the value that the commander has proposed (but the commander need not be correct)

The interactive consistency problem

Interactive consistency: all correct processes agree on a vector of values, one for each process.

This is called the decision vector

- Another variant of consensus

- *Termination*: eventually each correct process sets its decision variable
- *Agreement*: the decision vector of all correct processes is the same
- *Integrity*: if any process is correct, then all correct processes decide the correct value for that process

Relating consensus to other problems

Consensus (C), Byzantine Generals (BG), and Interactive Consensus (IC) are all problems concerned with making decisions in the context of arbitrary or crash failures

We can sometimes generate solutions for one problem in terms of another. For example

- We can derive IC from BG by running BG N times, once for each process with that process acting as commander
- We can derive C from IC by running IC to produce a vector of values at each process, then applying a function to the vector's values to derive a single value.
- We can derive BG from C by
 Commander sends proposed value to itself and each remaining process
 All processes run C with received values
 They derive BG from the vector of C values

Consensus in a Synchronous System

Up to f processes may have crash failures, all failures occurring during $f+1$ rounds. During each round, each of the correct processes multicasts the values among themselves

The algorithm guarantees all surviving correct processes are in a position to agree
 Note: any process with f failures will require at least $f+1$ rounds to agree

Limits for solutions to Byzantine Generals

- Some cases of the Byzantine Generals problems have no solutions
 - Lamport *et al* found that if there are only 3 processes, there is no solution
 - Pease *et al* found that if the total number of processes is less than three times the number of failures plus one, there is no solution
- Thus there is a solution with 4 processes and 1 failure, if there are two rounds
 - In the first, the commander sends the values
 - while in the second, each lieutenant sends the values it received

Τηρασηινγ

It can be argued that it is the programmer's responsibility to avoid thrashing. The programmer could annotate data items in order to assist the DSM runtime in minimizing page copying and ownership transfers. The latter approach is discussed in the next section in the context of the Munin DSM system.

Important Questions Module wise

Q.No	Questions	Bloom's Taxonomy Level	CO's	Marks
1.	a)Describe the challenges of the distributed systems with their examples? b)Explain the applications of the distributed systems.	Applying Applying	1 1	8M 4M
OR				
2.	a)Describe all the types of the fundamental models? b) Discuss in detail about the Internet protocols and list internet addresses?	Applying Understanding	1 1	8M 4M
3.	a)Describe the definition of the marshalling and write three alternative approaches in the external data representation? b) Distinguish client-server communication in detail?	Applying Analyzing	2 2	8M 4M
OR				
4.	a)Explain the Inter Process Communication with examples. b)Explain Distributed Object Model and also discuss the design issues of RMI.	Understanding Understanding	2 2	4M 8M
5.	a)Compare Sun network file system and Andrew file system? b)Explain file service architecture in detail.	Analyzing Applying	3 3	6M 6M
OR				
6	a)Explain in detail about cryptography algorithms. b)Explain about distributed file system.	Understanding Understanding	3 3	6M 6M
7.	a)Explain directory and discovery services. b)Discuss about global name service in detail?	Applying Understanding	4 4	6M 6M
OR				
8.	a)Explain clocks, events and process states. b)Distinguish distributed deadlock detection and termination Detection?	Understanding Analyzing	4 4	4M 8M
9.	a)Write a brief note on nested transactions? b) Explain concurrency control in distributed transactions.	Applying Understanding	5 5	6M 6M

OR				
10.	a) Explain in detail about flat and nested distributed transactions with neat diagram.	Understanding	5	8M
	b) Explain Atomic commit protocols in detail.	Understanding	5	4M