CS6601 – DISTRIBUTED SYSTEMS

QUESTION BANK

UNIT 1
PART A

1. **What is a distributed system?**

   A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages. The components interact with each other in order to achieve a common goal.

2. **Mention few examples of distributed systems.**

   Some examples of distributed systems are web search, Massively multiplayer online games(MMOGs), Financial trading markets, SOA based systems etc.

3. **Mention the trends in distributed systems.**

   Following are the trends in distributed systems:
   - Emergence of pervasive networking technology
   - Emergence of ubiquitous computing coupled with desire to support user mobility in distributed systems
   - Increasing demand for multimedia services
   - The view of distributed systems as a utility.

4. **What are backbones in intranets?**

   The intranets are linked together by backbones. A *backbone* is a network link with a high transmission capacity, employing satellite connections, fiber optic cables and other high-bandwidth circuits.

5. **Write short notes about webcasting.**

   Webcasting is an application of distributed multimedia technology. Webcasting is the ability to broadcast continuous media, typically audio or video, over the Internet. It is now commonplace for major sporting or music events to be broadcast in this way often attracting large numbers of viewers.

6. **Define cloud computing.**

   A cloud is defined as a set of Internet-based application, storage and computing services sufficient to support most users' needs, thus enabling them to largely or totally dispense with local data storage and application software. The term cloud computing refers to the practice of using a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server or a personal computer.

7. **What is a cluster computer? Mention its goals.**

   A cluster computer is a set of interconnected computers that cooperate closely to provide a single, integrated high performance computing capability. It consists of a set of loosely or tightly connected computers. Computer clusters have each node set to perform the same task, controlled and scheduled by software.

8. **Write short notes on mobile and ubiquitous computing.**

   Mobile Computing is a technology that allows transmission of data, voice and video via a computer or any other wireless enabled device without having to be connected to a fixed physical link. It involves mobile communication, mobile hardware, and mobile software.

   Ubiquitous computing (ubicomp) is a concept in software engineering and computer science where computing is made to appear anytime and everywhere. In contrast to desktop computing, ubiquitous computing can occur using any device, in any location, and in any format. This paradigm is also described as pervasive computing.

9. **What does the term remote invocation mean?**

   Remote invocation mechanism facilitates to create a distributed application. It provides a remote communication using two objects stub and skeleton. In this client-server approach remote object plays main role and it is an object whose method can be invoked from another JVM. In the client side, stub acts as a gateway. In the server side, skeleton acts as the gateway.

10. **What is the role of middleware?**

The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. In addition to solving the problems of heterogeneity, middleware provides a uniform computational model for use by the programmers of servers and distributed applications.

11. What are the challenges of distributed systems?

The main challenges of distributed system are:

- Heterogeneity
- Openness
- Security
- Scalability
- Failure handling
- Concurrency
- Transparency
- Quality of service

12. What is mobile code? Give an example.

The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination. Example is Java applet. The need for mobile code is that the code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

13. What determines the openness of distributed systems?

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

14. Mention the characteristics of open distributed systems.

- Open systems are characterized by the fact that their key interfaces are published.
- Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
- Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

15. What are the 2 security challenges that are not fully met by distributed systems?

- ***Denial of service attacks*:** One of the security problems is that a user may wish to disrupt a service for some reason. This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it. This is called a denial service*of* attack. There have been several denial of service attacks on well-known web services.
- ***Security of mobile code*:** Mobile code needs to be handled with care. Consider someone who receives an executable program as an electronic mail attachment. The possible effects of running the program are unpredictable.

16. When a system can be described as scalable in nature?

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. Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet.

17. What are the challenges faced by a scalable distributed system?

- **Controlling the cost of physical resources**: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it.
- **Controlling the performance loss:** Consider the management of a set of data whose size is proportional to the number of users or resources in the system.
- **Preventing software resources running out:** An example of lack of scalability is shown by the numbers used as Internet (IP) addresses

- **Avoiding performance bottlenecks:** In general, algorithms should be decentralized to avoid having performance bottlenecks.

18. What are the techniques used for dealing failures in a distributed system.

- ***Detecting failures*:** Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file.
- ***Masking failures*:** Some failures that have been detected can be hidden or made less severe.
- ***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.
- ***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.
- ***Redundancy*:** Services can be made to tolerate failures by the use of redundant components.

19. How the availability of a system can be measured?

The availabilityof a system is a measure of the proportion of time that it is available for use. When one of the components in a distributed system fails, only the work that was using the failed component is affected. A user may move to another computer if the one that they were using fails; a server process can be started on another computer.

20. Define Transparency. What are its types?

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.

Its types are:

- Access transparency
- Location transparency
- Concurrency transparency
- Replication transparency
- Failure transparency
- Mobility transparency
- Performance transparency
- Scaling transparency

21. What are the non-functional properties of a system that affects its quality of service?

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.

22. What are the main technological components of a web?

Then main technological components of a web are:

- **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
- **Hypertext Transfer Protocol (HTTP)** is client-server system architecture, with standard rules for interaction by which browsers and other clients fetch documents another resource from web servers.

23. What is HTML and HTTP?

HTML:HTML stands for HyperText Markup Language. It is a well-known mark up language used to develop web pages. It has been around for a long time and is commonly used in webpage design. HTML is written using HTML elements, which consist of tags, primarily and opening tag and a closing tag.

HTTP: HTTP, on the other hand, stands for Hypertext Transfer Protocol. It is a means of data communication for the World Wide Web. It is an application protocol for distributed,

collaborative, hypermedia information systems. HTTP is the protocol to exchange or transfer hypertext.

24. Why HTTP called as request-reply protocol?

In case of HTTP protocol, the client sends a request message to the server containing the URL of the required resource. The server looks up the path name and, if it exists, sends back the resource's content in a reply message to the client. Otherwise, it sends back an error response such as the familiar '404 Not Found'. Hence it is called as request-reply protocol.

PART B

1. Discuss in detail about the real time examples of distributed systems.

There are somespecific examples of distributed systems to illustrate the diversity and indeed complexity of distributed systems provision today.

Such examples are:

Web Search

Web search has emerged as a major growth industry in the last decade, with recent figures indicating that the global number of searches has risen to over 10 billion per calendar month. The task of a web search engine is to index the entire contents of the World Wide Web. This is a very complex task, as current estimates state that the Web consists of over 63 billion pages and one trillion unique webaddresses. Given that most search engines analyze the entire web content and then carry out sophisticated processing on this enormous database, this task itself represents a major challenge for distributed systems design.Google, the market leader in web search technology, has put significant effort into the design of a sophisticated distributed system infrastructure to support search

Highlights of this infrastructureinclude:

• an underlying physical infrastructure consisting of very large numbers ofnetworked computers located at data centres all around the world;

• a distributed file system designed to support very large files and heavily optimizedfor the style of usage required by search and other Google applications

• an associated structured distributed storage system that offers fast access to very large datasets;

• a lock service that offers distributed system functions such as distributed lockingand agreement

• a programming model that supports the management of very large parallel anddistributed computations across the underlying physical infrastructure.

Massive Multiplayer Online Games(MMOGs)

Massively multiplayer online games offer an immersive experience whereby very large numbers of users interact through the Internet with a persistent virtual world. Leading examples of such games include Sony's EverQuest II and EVE Online from the Finnish company CCP Games.The number of players is also rising, with systems able to support over 50,000 simultaneous online players.The engineering of MMOGs represents a major challenge for distributed systems technologies, particularly because of the need for fast response times to preserve the user experience of the game.

A number of solutions have been proposed for the design of massively multiplayer online games:

1. Perhaps surprisingly, the largest online game, EVE Online, utilises a *client-server* architecture where a single copy of the state of the world is maintained on a centralized server and accessed by client programs running on players' consoles or other devices. To support large numbers of clients, the server is a complex entity in its own right consisting of a cluster architecture featuring hundreds of computer nodes.

2. Other MMOGs adopt more distributed architectures where the universe is partitioned across a (potentially very large) number of servers that may also be geographically distributed.

Financial Trading

The financial industry has long been at the cutting edge of distributed systems technology with its need, in particular, for real-time access to a wide range of information sources.

Note that the emphasis in such systems is on the communication and processing of items of interest, known as *events* in distributed systems, with the need also to deliver events reliably and in a timely manner to potentially very large numbers of clients who have a stated interest in such information items. Examples of such events include a drop in a share price, the release of the latest unemployment figures, and so on. This requires

a very different style of underlying architecture from the styles mentioned above and such systems typically employ what are known as *distributed event-based systems*.

This approach is primarily used to develop customized algorithmic trading strategies covering both buying and selling of stocks and shares, in particular looking for patterns that indicate a trading opportunity and then automatically responding by placing and managing orders.

2. Explain briefly about the various trends in distributed systems.

Distributed systems are undergoing a period of significant change and this can be traced back to a number of influential trends:

- the emergence of pervasive networking technology;
- the emergence of ubiquitous computing coupled with the desire to support usermobility in distributed systems;
- the increasing demand for multimedia services;
- the view of distributed systems as a utility.

Pervasive networking and the modern Internet

The modern Internet is a vast interconnected collection of computer networks of many different types, with the range of types increasing all the time and now including, for example, a wide range of wireless communication technologies such as WiFi, WiMAX, Bluetooth and third-generation mobile phone networks. The net result is that networking has become a pervasive resource and devices can be connected at any time and in any place.

The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email and file transfer. The role of a *firewall* is to protect an intranet by preventing unauthorized messages from leaving or entering. A firewall is implemented by filtering incoming and outgoing messages. Internet Service Providers (ISPs) are companies that provide broadband links and other types of connection to individual users and small organizations. A *backbone* is a network link with a high transmission capacity, employing satellite connections, fibre optic cables and other high-bandwidth circuits.

Mobile and ubiquitous computing

Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include laptop computers, mobile phones, Personal Digital Assistants(PDAs) etc.

The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their 'home' intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them. They can continue to access the Internet; they can continue to access resources in their home intranet; and there is increasing provision for users to utilize resources such as printers or even sales points that are conveniently nearby as they move around. The latter is also known as *location-aware* or *context-aware computing*. Mobility introduces a number of challenges for distributed systems,

including the need to deal with variable connectivity and indeed disconnection, and the need to maintain operation in the face of device mobility.

Ubiquitous computing is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings. Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. But they are distinct, in general. Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital. Similarly, mobile computing has advantages even if it involves only conventional, discrete computers and devices such as laptops and printers.

Distributed multimedia systems

Another important trend is the requirement to support multimedia services in distributed systems. Multimedia support can usefully be defined as the ability to support a range of media types in an integrated manner. One can expect a distributed system to support the storage, transmission and presentation of what are often referred to as discrete media types, such as pictures or text messages. A distributed multimedia system should be able to perform the same functions for continuous media types such as audio and video; that is, it should be able to store and locate audio or video files, to transmit them across the network to support the presentation of the media types to the user and optionally also to share the media types across a group of users. The benefits of distributed multimedia computing are considerable in that a wide range of new services and applications can be provided on the desktop.

Webcasting is an application of distributed multimedia technology. Webcasting is the ability to broadcast continuous media, typically audio or video, over the Internet.

Distributed computing as a utility

With the increasing maturity of distributed systems infrastructure, a number of companies are promoting the view of distributed resources as a commodity or utility, drawing the analogy between distributed resources and other utilities such as water or electricity.

This model applies to both physical resources and more logical services:

Physical resources such as storage and processing can be made available to networked computers, removing the need to own such resources on their own. At one end of the spectrum, a user may opt for a remote storage facility for file storage requirements and/or for backups.

Software services can also be made available across the global Internet using this approach.

The term *cloud computing* is used to capture this vision of computing as a utility. A cloud is defined as a set of Internet-based application, storage and computing services sufficient to support most users' needs, thusenabling them to largely or totally dispense with local data storage and application software.

A *cluster computer* is a set of interconnected computers that cooperate closely to provide a single, integrated high performance computing capability. The overall goal of cluster computers is to provide a range of cloud services, including high-performance computing capabilities, mass storage and richer application services such as web search.

3. Give a detailed explanation about the need for focus on resource sharing.

Users are so accustomed to the benefits of resource sharing that they may easilyoverlook their significance. We routinely share hardware resources such as printers, dataresources such as files, and resources with more specific functionality such as searchengines.

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.

The term *service is used* 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 bymeans 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-servercomputing*. 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*. The same process may be both a client and a server, since servers sometimes invoke operations on other servers. The terms 'client' and 'server' apply only to the roles played in a single request. Clients are active (making requests) and servers are passive (only waking up when they receive requests); servers run continuously, whereas clients last only as long as the applications of which they form a part.

4. Explain about distributed multimedia systems.

Another important trend is the requirement to support multimedia services in distributedsystems. Multimedia support can usefully be defined as the ability to support a range ofmedia types in an integrated manner. One can expect a distributed system to support thestorage, transmission and presentation of what are often referred to as discrete mediatypes, such as pictures or text messages. A distributed multimedia system should be ableto perform the same functions for continuous media types such as audio and video; thatis, it should be able to store and locate audio or video files, to transmit them across thenetwork (possibly in real time as the streams emerge from a video camera), to supportthe presentation of the media types to the user and optionally also to share the mediatypes across a group of users.

The benefits of distributed multimedia computing are considerable in that a wide range of new (multimedia) services and applications can be provided on the desktop, including access to live or pre-recorded television broadcasts, access to film libraries offering video-on-demand services, access to music libraries, the provision of audio andvideo conferencing facilities and integrated telephony features including IP telephony or related technologies such as Skype, a peer-to-peer alternative to IP telephony.

Webcasting is an application of distributed multimedia technology. Webcasting is the ability to broadcast continuous media, typically audio or video, over the Internet. It is now commonplace for major sporting or music events to be broadcast in this way, often attracting large numbers of viewers

Distributed multimedia applications such as webcasting place considerable demands on the underlying distributed infrastructure in terms of:

• providing support for an (extensible) range of encoding and encryption formats, such as the MPEG series of standards (including for example the popular MP3 standard otherwise known as MPEG-1, Audio Layer 3) and HDTV;

• providing a range of mechanisms to ensure that the desired quality of service can be met;

• providing associated resource management strategies, including appropriate scheduling policies to support the desired quality of service;

• providing adaptation strategies to deal with the inevitable situation in open systems where quality of service cannot be met or sustained.

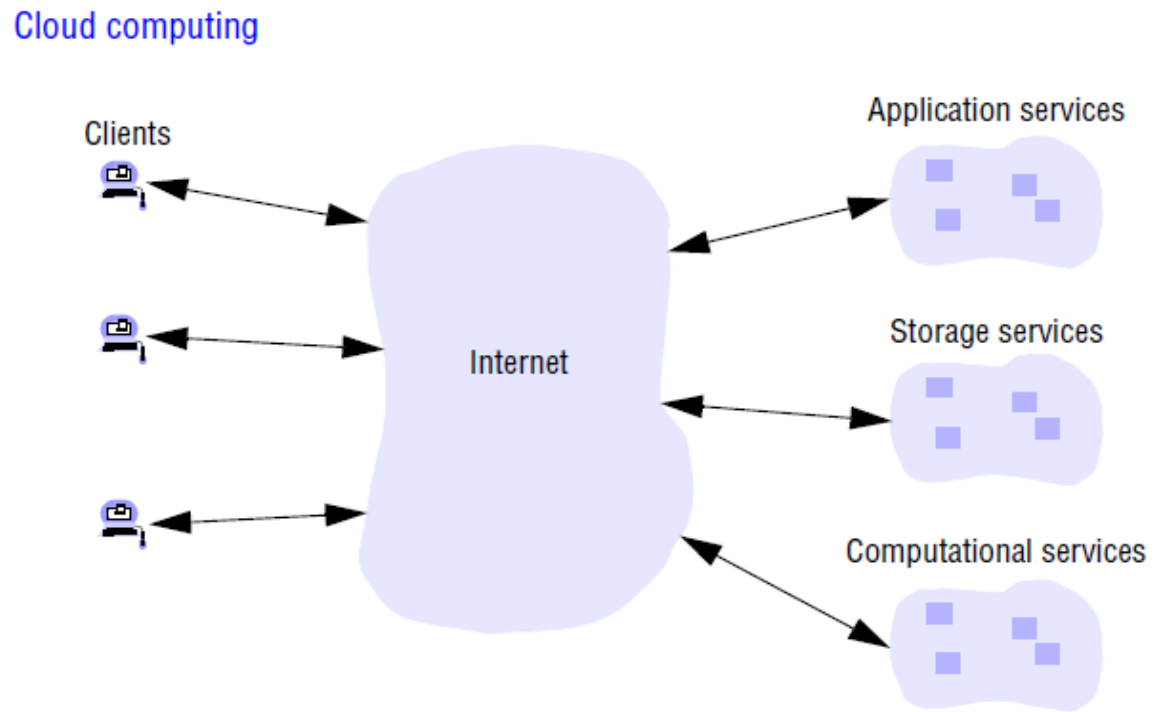
5. Briefly explain about how distributed computing serves as a utility.

With the increasing maturity of distributed systems infrastructure, a number ofcompanies are promoting the view of distributed resources as a commodity or utility,drawing the analogy between distributed resources and other utilities such as water orelectricity. With this model, resources are provided by appropriate service suppliers andeffectively rented rather than owned by the end user. This model applies to both physicalresources and more logical services:

Physical resources such as storage and processing can be made available to networked computers, removing the need to own such resources on their own. At one end of the spectrum, a user may opt for a remote storage facility for file storage requirements (for example, for multimedia data such as photographs, music or video) and/or for backups. Similarly, this approach would enable a user to rent one or more computational nodes, either to meet their basic computing needs or indeed to perform distributed computation. At the other end of the spectrum, users can access sophisticated *data centres* or indeed computational infrastructure using the sort of services now provided by companies such as Amazon and Google.

Software services can also be made available across the global Internet using this approach. Indeed, many companies now offer a comprehensive range of services for effective rental, including services such as email and distributed calendars.

The term *cloud computing* is used to capture this vision of computing as a utility. Acloud is defined as a set of Internet-based application, storage and computing servicessufficient to support most users' needs, thus enabling them to largely or totally dispensewith local data storage and application software. Clouds are generally implemented on cluster computers to provide the necessaryscale and performance required by such services. A *cluster computer* is a set of interconnected computers that cooperate closely to provide a single, integrated high performance computing capability.



The overall goal of cluster computers is to provide a range of cloud services, including high-performance computing capabilities, mass storage (for example through data centres), and richer application services such as web search. Grid computing is the collection of computerresources from multiple locations to reach a common goal. The grid can be thought of as adistributed system with non-interactive workloads that involve a large number of files. Grid computing is a distributed architecture of large numbers of computers connected to solve a complex problem.

6. Describe about the scalability and failure handling in distributed systems.

Scalability and Failure handling are two important challenges in handling distributed systems.

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.

The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computers increases. It must be possible to add server computers to avoid the performance bottleneck that would arise if a single file server had to handle all file access requests.

Controlling the performance loss: Consider the management of a set of data whose size is proportional to the number of users or resources in the system – for example, the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System, which is used mainly to look up DNS names such as www.amazon.com.

Preventing software resources running out: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses .

Avoiding performance bottlenecks: In general, algorithms should be decentralized to avoid having performance bottlenecks. We illustrate this point with reference to the predecessor of the Domain Name System, in which the name table was kept in a single master file that could be downloaded to any computers that needed it. That was fine when there were only a few hundred computers in the Internet, but it soon became a serious performance and administrative bottleneck.

Failure Handling

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 techniques for dealing with failures has been in implementation:

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file.

Masking failures: Some failures that have been detected can be hidden or made lesssevere.

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.

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.

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.

Redundancy: Services can be made to tolerate failures by the use of redundant components. Consider the following examples:
1. There should always be at least two different routes between any two routers in the Internet.
2. In the Domain Name System, every name table is replicated in at least twodifferent servers.
3. A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server; the servers can be designed to detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

Distributed systems provide a high degree of availability in the face of hardware faults. The *availability* of a system is a measure of the proportion of time that it is available for use. When one of the components in a distributed system fails, only the work that was using the failed component is affected.

7. Discuss how concurrency, transparency and quality of service are met in distributed systems.

Concurrency, transparency and quality of service are some of the challenges faced in a distributed system.

Concurrency
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 example 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. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

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. The ANSA Reference Manual [ANSA 1989] and the International Organization for Standardization's Reference Model for Open Distributed Processing (RM-ODP) [ISO 1992] identify eight forms of transparency. We have paraphrased the original ANSA definitions, replacing their migration transparency with our own mobility transparency, whose scope is broader:

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 orsoftware 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; theirpresence 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.

8. Discuss in detail about heterogeneity and openness 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.

The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. In addition to solving the problems of heterogeneity, middleware provides a uniform computational model for use by the programmers of servers and distributed applications. Possible models include remote object invocation, remote eventnotification, remote SQL access and distributed transaction processing.The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

The *virtual machine* approach provides a way of making code executable on a variety of host computers: the compiler for a particular language generates code for a virtual machine instead of a particular hardware order code. For example, the Java compiler produces code for a Java virtual machine, which executes it by interpretation. The Java virtual machine needs to be implemented once for each type of computer to enable Java programs to run.

Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

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 extendedat the hardware level by the addition of computers to the network and at the softwarelevel by the introduction of new services and the reimplementation of old ones, enablingapplication programs to share resources. A further benefit that is often cited for opensystems is their independence from individual vendors.

In short,

• Open systems are characterized by the fact that their key interfaces are published.

• Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.

• Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

9. Briefly explain about the challenges faced in distributed systems.

Major challenges faced in a distributed system are:

- Heterogeneity
- Openness
- Security
- Scalability
- Failure Handling
- Concurrency
- Transparency
- Quality of Service

Security

Many of the information resources that are made available and maintained in distributedsystems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality , integrity and availability

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.

Following two security challenges have not yet been fully met:

Denial of service attacks: Another security problem is that a user may wish to disrupt a service for some reason. This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it. This is called a *denial of service* attack.

Security of mobile code: Mobile code needs to be handled with care. Consider someone who receives an executable program as an electronic mail attachment: the possible effects of running the program are unpredictable.

(Along with the above content the answers for Q.No 6,7 & 8of Part B has to be included)

10. Explain in detail about World Wide Web.

The World Wide Web is an evolving system forpublishing and accessing resources and services across the Internet. Through commonlyavailable web browsers, users retrieve and view documents of many types, listen toaudio streams and view video streams, and interact with an unlimited set of services.The Web began life at the European center for nuclear research (CERN),Switzerland, in 1989 as a vehicle for exchanging documents between a community ofphysicists connected by the Internet [Berners-Lee 1999]. A key feature of the Web is thatit 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.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.

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.

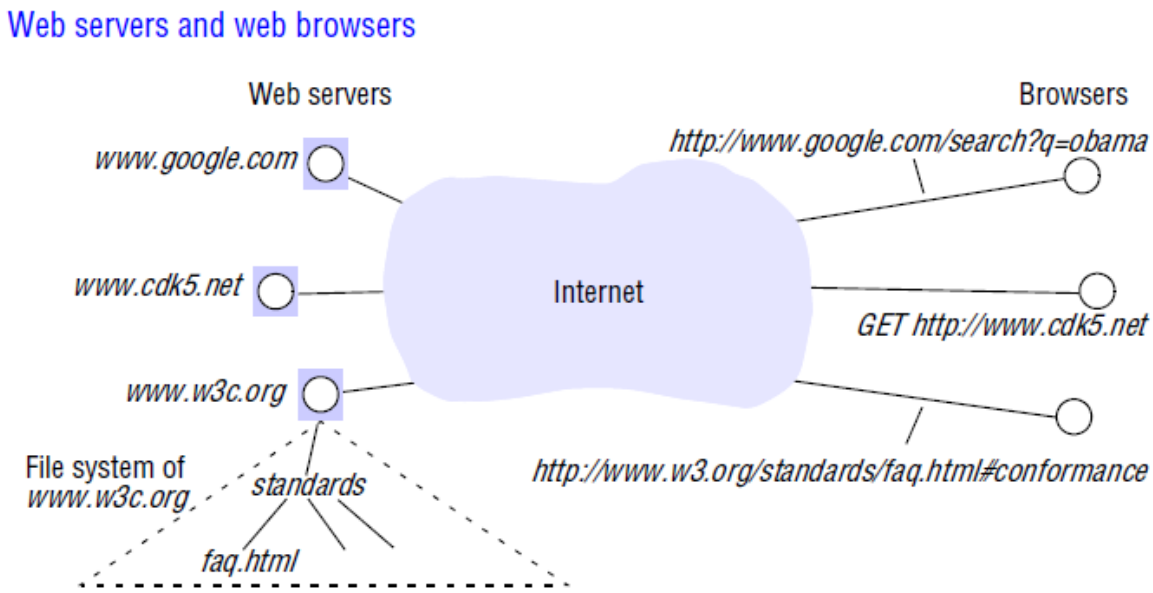
HTML

The HyperText Markup Language is used to specify the textand images that make up the contents of a web page, and to specify how they are laidout and formatted for presentation to the user. A web

page contains such structured itemsas headings, paragraphs, tables and images. HTML is also used to specify links andwhich resources are associated with them.

URLs

The purpose of a Uniform Resource Locator is to identify a resource. Indeed, the term used in web architecture documents is Uniform Resource *Identifier* (URI), but in this book the better-known term URL will be used when no confusion can arise. Browsers examine URLs in order to access the corresponding resources.HTTP URLs are the most widely used, for accessing resources using the standardHTTP protocol. An HTTP URL has two main jobs: to identify which web servermaintains the resource, and to identify which of the resources at that server is required.



HTTP

The HyperText Transfer Protocol defines the ways in which browsers and other types of client interact with web servers.

Request-reply interactions: HTTP is a 'request-reply' protocol. The client sends a request message to the server containing the URL of the required resource. The server looks up the path name and, if it exists, sends back the resource's content in a reply message to the client. Otherwise, it sends back an error response such as the familiar '404 Not Found'. HTTP defines a small set of operations or *methods* that can be performed on a resource. The most common are GET, to retrieve data from theresource, and POST, to provide data to the resource.

Content types: Browsers are not necessarily capable of handling every type of content. When a browser makes a request, it includes a list of the types of content it prefers – for example, in principle it may be able to display images in 'GIF' format but not 'JPEG' format.

UNIT II
PART A

1. **What are the issues in distributed system?**
   - There is no global time in a distributed system, so the clocks on different computers do not necessarily give the same time as one another.
   - All communication between processes is achieved by means of messages. Message communication over a computer network can be affected by delays, can suffer from a variety of failures and is vulnerable to security attacks.
2. **What are the difficulties and threats for distributed systems?**
   - Widely varying modes of use
   - Wide range of system environments

- Internal problems
- External threats
- Attacks on data integrity and secrecy
- denial of service attacks

3. **What is a physical model?**

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

4. **What is meant by Distributed systems of systems/ Ultra-Large-Scale (ULS) distributed systems?**

   A system of systems (mirroring the view of the Internet as a network of networks) can be defined as a complex system consisting of a series of subsystems that are systems in their own right and that come together to perform a particular task or tasks.

5. **What are the three generations of distributed systems?**
   - Early distributed systems
   - Internet-scale distributed systems
   - Contemporary distributed systems

6. **What is a Web Service?**

   The World Wide Web Consortium (W3C) defines a web service as a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based message exchanges via Internet-based protocols.

7. **What is a cache?**

   A cache is a store of recently used data objects that is closer to one client or a particular set of clients. When a new object is received from a server it is added to the local cache store, replacing some existing objects if necessary.

8. **Define Mobile Agent.**

   A mobile agent is a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, and eventually returning with the results. A mobile agent may make many invocations to local resources at each site it visits.

9. **What are Thin clients?**

   Thin client refers to a software layer that supports a window-based user interface that is local to the user while executing application programs or, more generally, accessing services on a remote computer.

10. **What is meant by Reflection?**

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

11. **What do you mean by Masking failures?**

    A service masks a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages, effectively converting an arbitrary failure into an omission failure. Masking can be done by means of replication.

12. **Define marshalling and unmarshalling**

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

13. **What is meant by XML (eXtensible Markup Language)?**

- XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web.
- It is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable.
- XML data is known as self-describing or self-defining which means that the structure of the data is embedded with the data.
- XML was designed for writing structured documents for the Web.
- XML documents, being textual, can be read by humans.

14. What do you mean by XML prolog?

- Every XML document must have a prolog as its first line. The prolog must at least specify the version of XML in use (which is currently 1.0).
- **Example:** <?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>

15. What are XML namespaces?

An XML namespace is a set of names for a collection of element types and attributes that is referenced by a 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.

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

16. What do you mean by XML schemas?

An XML schema defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text.

17. What is a Remote object reference?

A remote object reference is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Remote object references must be generated in a manner that ensures uniqueness over space and time.

18. What is a multicast operation?

A multicast operation is the mechanism where an operation 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.

19. What is a overlay network?

An overlay network is a virtual network consisting of nodes and virtual links, which sits on top of an underlying network (such as an IP network).

20. What are the design issues for RPC.

- The style of programming promoted by RPC programming with interfaces;
- The call semantics associated with RPC;
- The key issue of transparency and how it relates to remote procedure calls.

21. What is a service interface?

The term service interface is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

22. What is an IDL?

Interface Definition Language (IDL) is designed to allow procedures implemented in different languages to invoke one another. An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.

23. What do you mean by Maybe semantics?

With maybe semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

• Omission failures-If the request or result message is lost;

• Crash failures-When the server containing the remote operation fails.

24. What is meant by At-least-once semantics?

With At-least-once semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received.

25. What is meant by At-most-once semantics?

With At-most-once semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

26. What is meant by Garbage collection?

It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed. A language such as Java, that can detect automatically when an object is no longer accessible recovers the space and makes it available for allocation to other objects. This process is called garbage collection.

27. What do you mean by a Servant?

A servant is an instance of a class that provides the body of a remote object. It is the servant that eventually handles the remote requests passed on by the corresponding skeleton. Servants live within a server process.

28. What is meant by Group communication?

Group communication offers a service whereby a message is sent to a group and then this message is delivered to all members of the group. In this action, the sender is not aware of the identities of the receivers.

29. Define Closed and open groups.

A group is said to be closed if only members of the group may multicast to it. A process in a closed group delivers to itself any message that it multicasts to the group. A group is open if processes outside the group may send to it.

30. Define Overlapping and non-overlapping groups.

In overlapping groups, entities (process-es or objects) may be members of multiple groups, and non-overlapping groups imply that membership does not overlap (that is, any process belongs to at most one group).

31. What is meant by FIFO ordering?

First-in-First-out (FIFO) ordering (also referred to as source ordering) is concerned with preserving the order from the perspective of a sender process, in that if a process sends one message before another, it will be delivered in this order at all processes in the group.

32. What is meant by Causal ordering?

Causal ordering takes into account causal relationships between messages, in that if a message happens before another message in the distributed system this so-called causal relationship will be preserved in the delivery of the associated messages at all processes.

33. What do you mean by publish-subscribe system?

A publish-subscribe system is a system where publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events.

PART B

1. Explain the various Architectural elements, architectural styles and architectural patterns.

To understand the fundamental building blocks of a distributed system, it is necessary to consider **four key questions**:

- What are the entities that are communicating in the distributed system?
- How do they communicate, or, more specifically, what communication paradigm is used?
- What roles and responsibilities do they have in the overall architecture?
- How are they mapped on to the physical distributed infrastructure?

Communicating entities

1.System-oriented entities
2.Problem-oriented entities

Communication paradigms

Three types of communication paradigm:

1.Interprocess communication

- refers to the low-level support for communication between processes in distributed systems

2.Remote invocation

- cover a range of techniques based on a two-way exchange between communicating entities in a distributed system and resulting in the calling of a remote operation, procedure or method.

3.Indirect communication

Key techniques for indirect communication include:

a.Group communication

- Group communication is concerned with the delivery of messages to a set of recipients and hence is a multiparty communication paradigm supporting one-to-many communication.

b.Publish-subscribe systems/ distributed event-based systems

- Many systems, such as the financial trading can be classified as information-dissemination systems wherein a large number of producers (or publishers) distribute information items of interest (events) to a similarly large number of consumers (or subscribers).

c.Message queues

- Message queues offer a point-to-point service whereby producer processes can send messages to a specified queue and consumer processes can receive messages from the queue or be notified of the arrival of new messages in the queue.

d.Tuple spaces/Generative communication

- Processes can place arbitrary items of structured data, called tuples, in a persistent tuple space and other processes can either read or remove such tuples from the tuple space by specifying patterns of interest.

e.Distributed shared memory

- Distributed shared memory (DSM) systems provide an abstraction for sharing data between processes that do not share physical memory.

Two architectural styles

1.Client-server Architecture

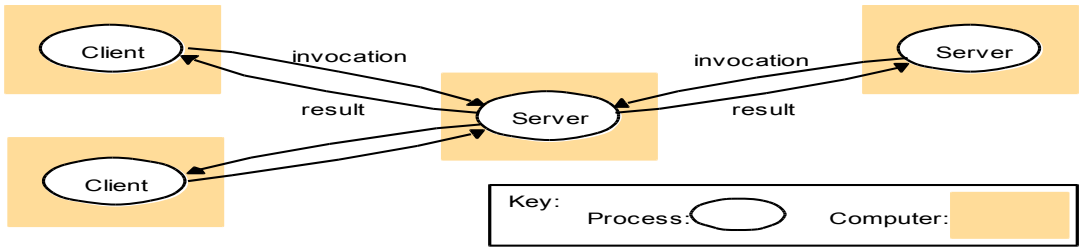


Figure 2.3 Clients invoke individual Servers

Figure 2.3 illustrates the simple structure in which processes take on the roles of being clients or servers. 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.

2.Peer-to-peer Architecture

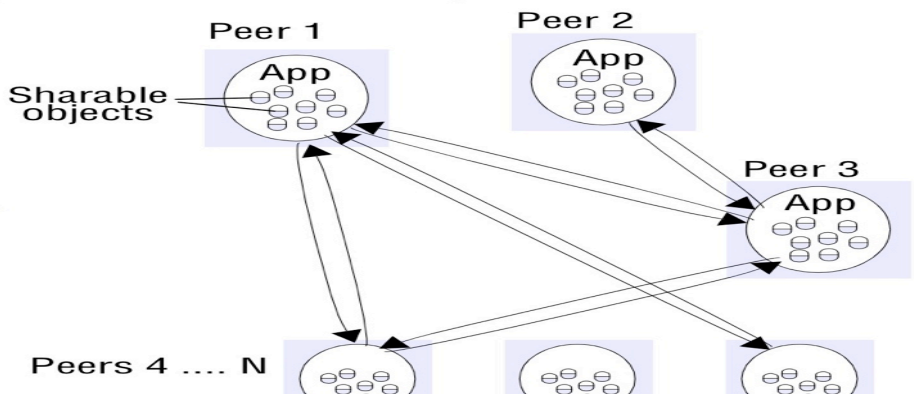


Figure 2.4a Peer-to-peer Architecture

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.

Architectural Patterns

Architectural patterns build on the more primitive architectural elements. They are not themselves necessarily complete solutions but rather offer partial insights that, when combined with other patterns, lead the designer to a solution for a given problem domain.

1.Layering Architectural Pattern

In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details, or indeed of any other layers beneath them.

In terms of distributed systems, this equates to a **vertical organization of services** into service layers. A distributed service can be provided by one or more server processes, interacting with each other and with client processes.

Given the complexity of distributed systems, it is often helpful to organize such services into layers. A common view of a layered architecture in Figure 2.7 introduces the important terms platform and middleware, which we define as follows:

Figure 2.7 Software and hardware service layers in distributed systems

Platform

A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates communication and coordination between processes.

Example

- Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are major examples.

Middleware

- Middleware was defined as a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers.
- Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and resource-sharing support for distributed applications.

2.Tiered architecture

Tiered architectures are complementary to layering. Whereas layering deals with the vertical organization of services into layers of abstraction, tiering is a technique to **organize functionality of a given layer** and place this functionality into appropriate servers.

Two- and three-tiered architecture

To illustrate this, consider the functional decomposition of a given application, as follows:

Presentation logic, which is concerned with handling user interaction and updating the view of the application as presented to the user;

Application logic, which is concerned with the detailed application-specific processing associated with the application (also referred to as the **business logic**);

Data logic, which is concerned with the persistent storage of the application, typically in a database management system.

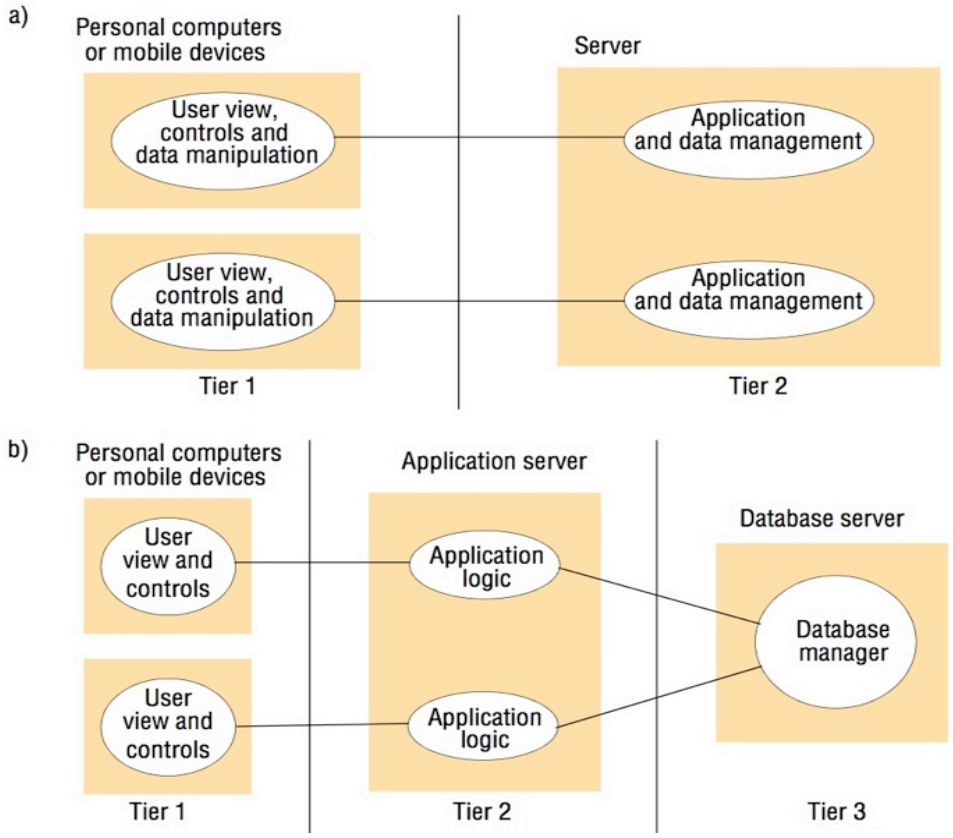
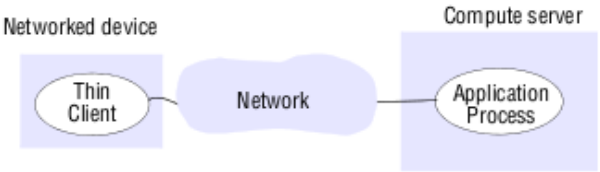


Figure 2.8 Two-ti

3.Thin clients

Thin client refers to a software layer that supports a window-based user interface that is local to the user while executing application programs or, more generally, accessing services on a remote computer.

Figure 2.10 Thin clients and computer servers



Advantage

- simple local devices (including, for example, smart phones and other resource-constrained devices) can be significantly enhanced with a excess of networked services and capabilities.

Drawback

- highly interactive graphical activities such as CAD and image processing, where the delays experienced by users are increased to unacceptable levels by the need to transfer image between the thin client and the application process.

4.Other commonly occurring patterns

a.Proxy pattern

A commonly recurring pattern in distributed systems designed particularly to support location transparency in remote procedure calls or remote method invocation. With this approach, a proxy is

created in the local address space to represent the remote object. This proxy offers exactly the same interface as the remote object, and the programmer makes calls on this proxy object and hence does not need to be aware of the distributed nature of the interaction.

b.Web Service Architectural Pattern

An architectural pattern supporting interoperability in potentially complex distributed infrastructures. In particular, this pattern consists of the trio of service provider, service requester and service broker (a service that matches services provided to those requested), as shown in Figure 2.11.

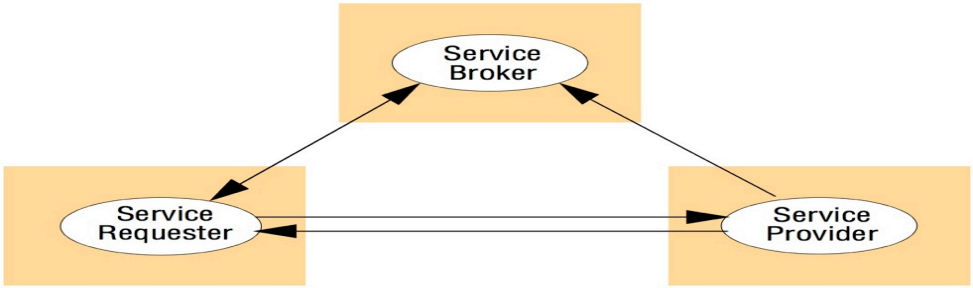


Figure 2.11 Web Service Architectural Pattern

c.Reflection

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

2. Discuss about the three types of Fundamental models.

Types

1. Interaction Models
2. Failure Model
3. Security Model

1.Interaction Models

Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes. Distributed systems are composed of many processes, interacting in complex ways. **Example**

- Multiple server processes may cooperate with one another to provide a service
- A set of peer processes may cooperate with one another to achieve a common goal

Two significant factors affecting interacting processes in a distributed system

a. Performance of communication channels
b. Computer clocks and timing events

Two variants of the interaction model

a.Synchronous distributed systems
b.Asynchronous distributed systems

a.Synchronous distributed systems

Hadzilacos and Toueg [1994] define a synchronous distributed system to be one in which the following bounds are defined:

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

b.Asynchronous distributed systems

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

- **Process execution speeds –** for example, one process step may take only a picosecond and another a century; all that can be said is that each step may take an arbitrarily long time.
- **Message transmission delays –** for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.
- **Clock drift rates –** again, the drift rate of a clock is arbitrary.

2.Failure Model

The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos and Toueg [1994] provide a taxonomy that distinguishes between the failures of processes and communication channels. These are presented under the headings:

a) Omission failures
b) Arbitrary failures
c) Timing failures
d) Masking failures

a)Omission failures

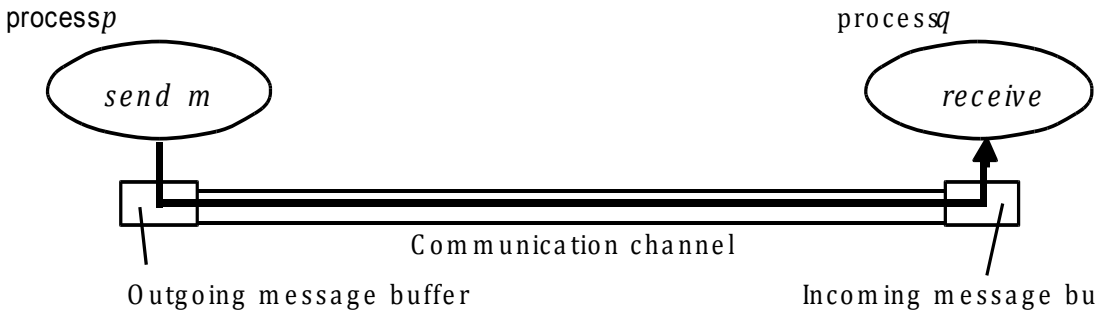
The faults classified as omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do.

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.

Communication omission failures

Consider the communication primitives send and receive. A process p performs a send by inserting the message m in its outgoing message buffer. The communication channel transports m to q's incoming message buffer. Process q performs a receive by taking m from its incoming message buffer and delivering it (see Figure 2.14). 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.

Hadzilacos and Toueg [1994] refer to the loss of messages between the sending process and the outgoing message buffer as **send-omission failures**, to loss of messages between the incoming message buffer and the receiving process as **receive-omission failures**, and to loss of messages in between as **channel omission failures**.

b)Arbitrary failures/ Byzantine failure

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.

c)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. Any one of these failures may result
in responses being unavailable to clients within a specified time interval.

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

Figure 2.16 Timing failures

Class of failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.

d)Masking failures

A service masks a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages, effectively converting an arbitrary failure into an omission failure. Masking can be done by means of replication.

3.Security model

The security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

Protecting objects

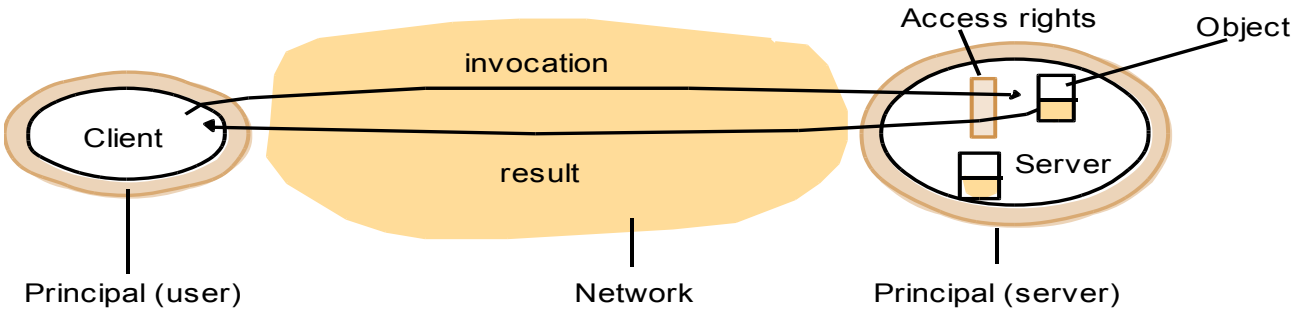


Figure 2.17 Objects and principals

Figure 2.17 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.

The users are 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.

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.

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. The following discussion introduces a model for the analysis of security threats.

The enemy/The Adversary

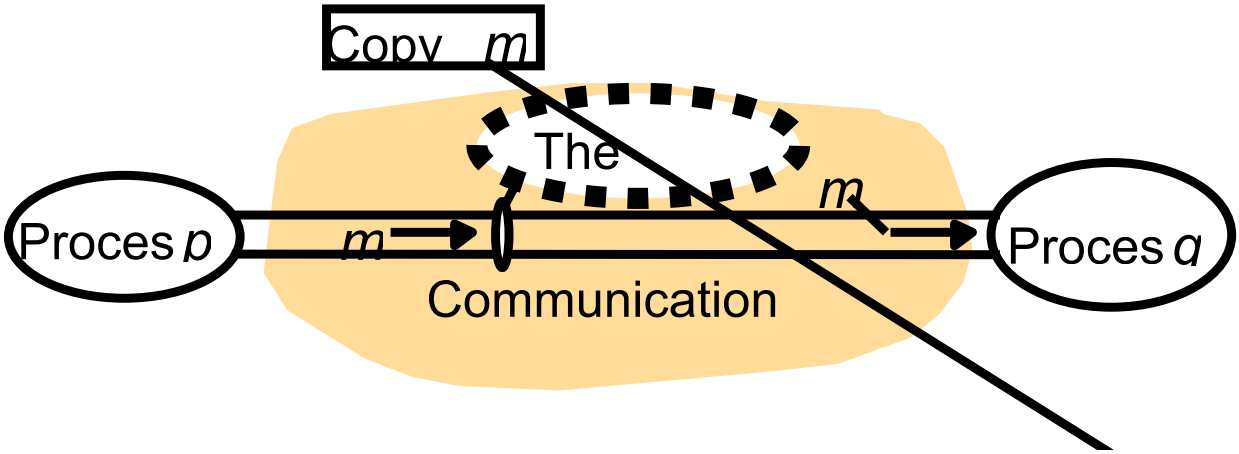


Figure 2.18 The Enemy

To model security threats, we postulate an enemy that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in Figure 2.18. 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 threats from a potential enemy include

a)Threats to processes

b)Threats to communication channels

Defeating security threats

a)Cryptography and shared secrets

Suppose that a pair of processes (for example, a particular client and a particular server) share a secret; that is, they both know the secret but no other process in the distributed system knows it.

Cryptography is the science of keeping messages secure, and encryption is the process of scrambling a message in such a way as to hide its contents.

b)Authentication

Authentication of messages means proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to guarantee its authenticity.

Secure channels

A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in Figure 2.19. A secure channel has the following properties:

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing.
- A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered.

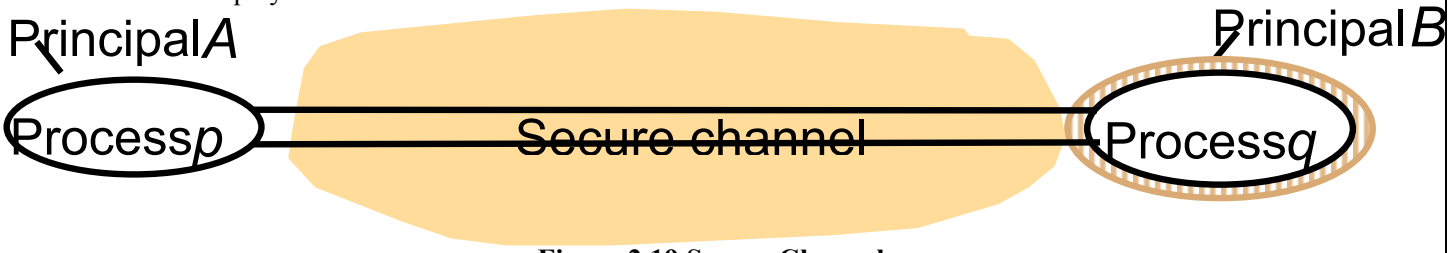


Figure 2.19 Secure Channels

c)Other possible threats from an enemy

Denial of service

This is a form of attack in which the enemy interferes with the activities of authorized users by making excessive and pointless invocations on services or message transmissions in a network, resulting in overloading of physical resources (network bandwidth, server processing capacity). Such attacks are usually made with the intention of delaying or preventing actions by other users.

Mobile code

Mobile code receives and executes program code from elsewhere, such as the email attachment. Such code may easily play a Trojan horse role, purporting to fulfil an innocent purpose but in fact including code that accesses or modifies resources that are legitimately available to the host process but not to the originator of the code.

3. Discuss in detail the API for the internet protocols.

Generally the interprocess communication takes place using the internet protocols. TCP and UDP are such mainly used internet protocols used.

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.

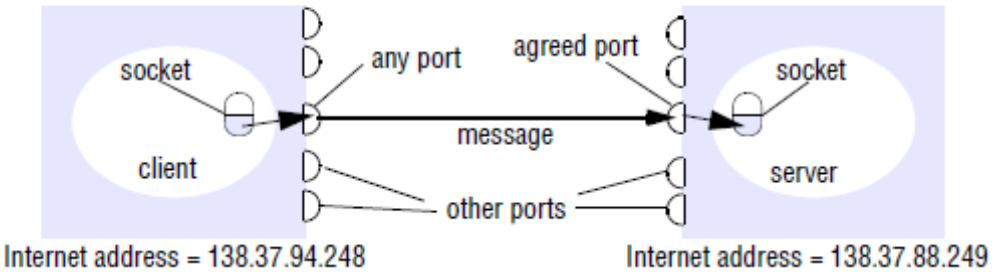
Synchronous and asynchronous communication

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 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 hasbeen copied to a local buffer, and the transmission of the message proceeds in parallelwith the sending process.

A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but can have many senders. Processes may use multiple ports to receive messages.

Sockets and ports



Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process. Each socket is associated with a particular protocol – either UDP or TCP. As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames

UDP Datagram communication

A 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 *serverport* – 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.

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.

API for UDP

The Java API provides datagram communication by

means of two classes: *DatagramPacket* and *DatagramSocket*.
DatagramPacket: This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the Internet address and local port number of the destination socket, as follows:

Datagram packet

array of bytes containing message	length of message	Internet address	port number

An instance of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it.

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 a no-argument constructor that allows the system to choose a free local port

UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
     // args give message contents and server hostname
     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 e){System.out.println("Socket: " + e.getMessage());
      } catch (IOException e){System.out.println("IO: " + e.getMessage());
      } finally { if(aSocket != null) aSocket.close();}
   }
}
```

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

```
import java.net.*;
import java.io.*;
public class UDPServer{
   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 e){System.out.println("Socket: " + e.getMessage());
     } catch (IOException e) {System.out.println("IO: " + e.getMessage());
     } finally {if (aSocket != null) aSocket.close();}
   }
}
```

TCP Stream Communication

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:

Message sizes: 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.
Flow control: 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 duplication and ordering: 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.

Message destinations: A pair of communicating processes establishes 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.

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.

JAVA API for TCP streams

The Java interface to TCP streams 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.

Socket: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error
occurs. The *Socket* class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket.

TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
      // arguments supply message and hostname of destination
      Socket s = 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 e){
        System.out.println("Sock:"+e.getMessage());
      } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
      } catch (IOException e){System.out.println("IO:"+e.getMessage());
      } finally {if(s!=null) try {s.close();}catch (IOException e){/*close failed*/}}
    }
}
```

4. Briefly explain external data representation and multicast communication.

External data representation

The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structure must be **flattened** (converted to a sequence of bytes) before transmission and rebuilt on arrival.

An agreed standard for the representation of data structures and primitive values is called an **external data representation**.

Three alternative approaches to external data representation and marshalling are discussed:

1. CORBA's common data representation
2. Java's object serialization
3. XML (Extensible Markup Language)

1.CORBA's Common Data Representation (CDR)

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

- It consists 15 **primitive types**:
  - Short (16 bit)
  - Long (32 bit)
  - Unsigned short
  - Unsigned long
  - Float(32 bit)
  - Double(64 bit)
  - Char
  - Boolean(TRUE,FALSE)
  - Octet(8 bit)
  - Any(can represent any basic or constructed type)
- **Constructed types** are shown below:

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

Figure 4.8 CORBA CDR message

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}

2.Java's object serialization

- In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.
- An object is an instance of a Java class.
  Example, the Java class equivalent to the Person struct
  - 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
- In Java, the term **serialization** refers to the activity of flattening an object or a connected set of

objects into a serial form that is suitable for storing on disk or transmitting in a message, for example, as an argument or the result of an RMI. **Deserialization** consists of restoring the state of an object or a set of objects from their serialized form.

3.XML (Extensible Markup Language)

- XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web.
- XML documents, being textual, can be read by humans. In practice, most XML documents are generated and read by XML processing software, but the ability to read XML can be useful when things go wrong.

XML elements and attributes

- Figure 4.10 shows the XML definition of the Person structure. 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.

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 4.10 consists of the data Smith contained within the <name> ... </name> tag pair.

Figure 4.10 XML definition of the *Person* structure

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

Attributes

A start tag may optionally include pairs of associated attribute names and values such as id="123456789", as shown above. The syntax is the same as for HTML, in which an attribute name is followed by an equal sign and an attribute value in quotes. Multiple attribute values are separated by spaces.

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.

XML namespaces

An XML namespace is a set of names for a collection of element types and attributes that is referenced by a 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.

Example:

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

XML schemas

An XML schema defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text. The sender of a SOAP message may use an XML schema to encode it, and the recipient will use the same XML schema to validate and decode it.

Figure 4.12
An XML schema for the Person structure

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
        <xsd:element name= "person" type ="personType" />
                <xsd:complexType name="personType">
                        <xsd:sequence>
                                <xsd:element name = "name" type="xs:string"/>
                                <xsd:element name = "place" type="xs:string"/>
                                <xsd:element name = "year" type="xs:positiveInteger"/>
                        </xsd:sequence>
                        <xsd:attribute name= "id" type = "xs:positiveInteger"/>
                </xsd:complexType>
</xsd:schema>
```

Document type definitions

Document type definitions (DTDs) were provided as a part of the XML 1.0 specification for defining the structure of XML documents and are still widely used for that purpose. It cannot describe data types and its definitions are global, preventing element names from being duplicated.

Multicast communication

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.

Characteristics of Multicast messages

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.

IP multicast

IP multicast is built on top of the Internet Protocol (IP). 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. 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.

Reliability and ordering of multicast

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.

5. Describe the concept and types of overlay networks.

An overlay network is a virtual network consisting of nodes and virtual links, which sitson top of an underlying network (such as an IP network) and offers something that is nototherwise provided:

- a service that is tailored towards the needs of a class of application or a particularhigher-level service – for example, multimedia content distribution;
- more efficient operation in a given networked environment – for example routingin an ad hoc network;
- an additional feature – for example, multicast or secure communication.

Types of Overlays

Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].
Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

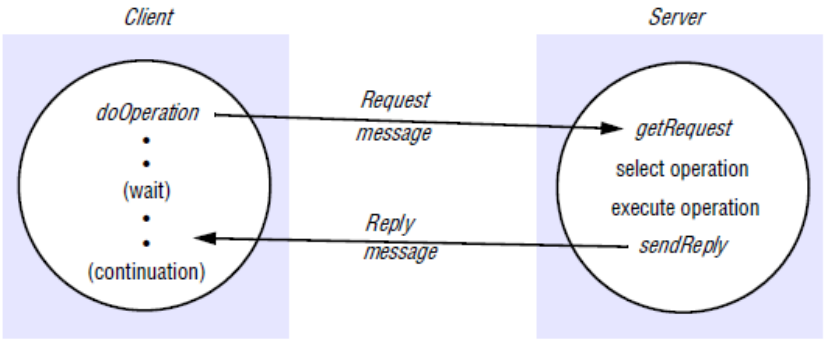
6. Give a detailed explanation about the request-reply protocols.

Request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client. Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later.

Request-reply protocol

The protocol is based on a trio of **communication primitives**, doOperation, getRequest and sendReply, as shown in Figure 5.2. This request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees.

Figure 5.2 Request-reply communication



- **doOperation** - used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply.
- **getRequest** - used by a server process to acquire service requests
- **sendReply** - When the server has invoked the specified operation, it then uses sendReply to send the reply message to the client. When the reply message is received by the client the original doOperation is unblocked and execution of the client program continues.
  The information to be transmitted in a request/reply message is shown in Figure 5.4.
- **First field** - whether the message is a Request or a Reply message.
- **Second field** - requestId, contains a message identifier. A doOperation in the client generates a requestId for each request message, and the server copies these IDs into the corresponding reply messages. This enables doOperation to check that a reply message is the result of the current request, not a delayed earlier call.
- **Third field** - a remote reference.
- **Fourth field** - identifier for the operation to be invoked.

Figure 5.4 Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

7. **Explain the design issues and implementation of RPC and RMI.**
   RPC and RMI play a very important role in the distributed computing. The introduction of reuquest reply protocols has lead to the development of RPC and RMI.

Remote Procedure Call(RPC)

Concept of a remote procedure call (RPC) represents amajor intellectual breakthrough in distributed computing**.** Three main issues involved in the design of RPC are:

- the style of programming promoted by RPC
- programming with interfaces
- the call semantics associated with RPC

The key issue of transparency and how it relates to remote procedure calls. In order to control the possible interactions between modules, an explicit *interface* is defined for each module. The interface of a module specifies the procedures and the variables that can be accessed from other modules.

In a distributed program, the modules can run in separate processes. In the client-server model, in particular, each server provides a set of procedures that are available for use by clients. For example, a file server would provide procedures for reading and writing files. The term *service interface* is used to
refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

CORBA IDL example

```
// In file Person.idl
struct Person {
    string name;
    string place;
    long year;
};
interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p) ;
    void getPerson(in string name, out Person p);
    long number();
};
```

Interface Definition Languages(IDL)

An RPC mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces, allowing input and output parameters to be mapped onto the language's normal use of parameters.

Interface definition languages (IDLs) are designed to allow procedures implemented in different languages to invoke one another. An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.

RPC call semantics

Request-reply protocols showed that *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

Retry request message: Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

Duplicate filtering: Controls when retransmissions are used and whether to filter out duplicate requests at the server.

Retransmission of results: Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

Maybe semantics: With *maybe* semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

omission failures if the request or result message is lost

crash failures when the server containing the remote operation fails.

At-least-once semantics: With *at-least-once* semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception informing it that no result was received. *At-least-once* semantics can be achieved by the retransmission of request messages, which masks the omission failures of the request or result message. *At-least-once* semantics can suffer from the following types of failure:

• crash failures when the server containing the remote procedure fails;

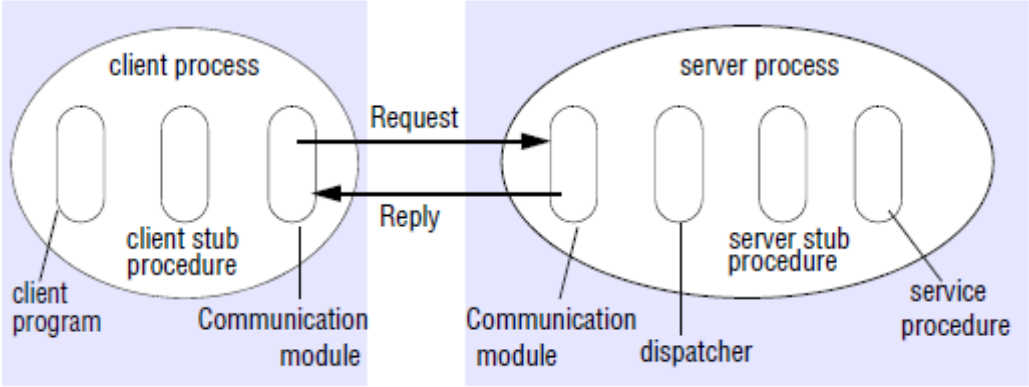
• arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values to be stored or returned.

At-most-once semantics: With *at-most-once* semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

Implementation of RPC

The client that accesses a service includes one *stub procedure* for each procedure in the service interface. The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results. The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface. The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message. The service procedures implement the procedures in the service interface. The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.

Role of client and server stub procedures in RPC



Remote Method Invocation(RMI)

Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects. In RMI, a calling object can invoke a method in a potentially remote object.

The commonalities between RMI and RPC are as follows:

- They both support programming with interfaces, with the resultant benefits that stem from this approach
- They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once.
- They both offer a similar level of transparency – that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

‘

Design issues for RMI

The object model - An object-oriented program, for example in Java or C++, consists of a collection of interacting objects, each of which consists of a set of data and a set of methods. An object communicates with other objects by invoking their methods, generally passing arguments and receiving results.

Object references: Objects can be accessed via object references. For example, in Java, a variable that appears to hold an object actually holds a reference to that object. To invoke a method in an object, the object reference and method name are given, together with any necessary arguments.

Interfaces: An interface provides a definition of the signatures of a set of methods without specifying their implementation. An object will provide a particular interface if its class contains code that implements the methods of that interface. In Java, a class may implement several interfaces, and the methods of an interface may be implemented by any class. An interface also defines types that can be used to declare the type of variables or of the parameters and return values of methods.

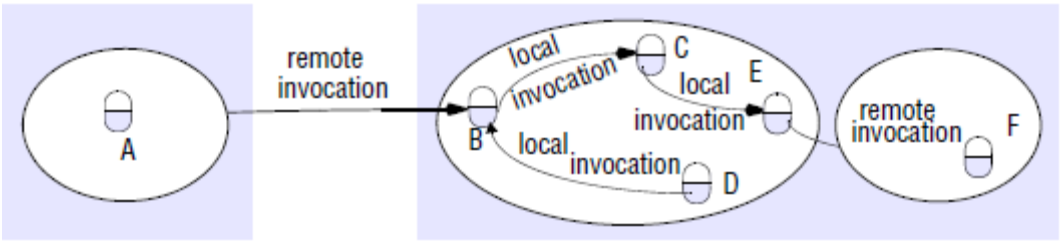
Actions : Action in an object-oriented program is initiated by an object invoking a method in another object. An invocation can include additional information (arguments) needed to carry out the method. The receiver executes the appropriate method and then returns control to the invoking object, sometimes supplying a result. An invocation of a method can have three effects:
1. The state of the receiver may be changed.
2. A new object may be instantiated, for example, by using a constructor in Java or C++.
3. Further invocations on methods in other objects may take place.

Exceptions: Programs can encounter many sorts of errors and unexpected conditions of varying seriousness. During the execution of a method, many different problems may be discovered: for example, inconsistent values in the object's variables, or failures in attempts to read or write to files or network sockets.

Garbage collection: It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed. A language such as Java, that can detect automatically when an object is no longer accessible recovers the space and makes it available for allocation to other objects. This process is called garbage collection.

Distributed objects
The state of an object consists of the values of its instance variables. In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object. Since object-based programs are logically partitioned, the physical distribution of objects into different processes or computers in a distributed system is a natural extension

Remote and local method invocations



The following two fundamental concepts are at the heart of the distributed object model:

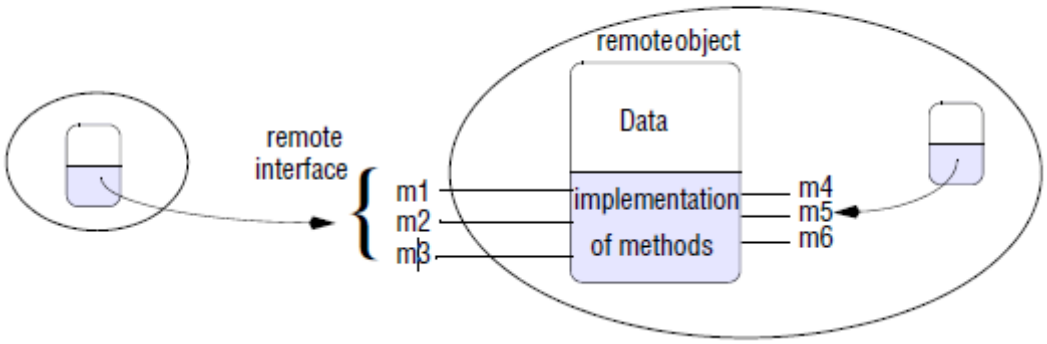
Remote object references: Other objects can invoke the methods of a remote object if they have access to its remote object reference.

Remote interfaces: Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

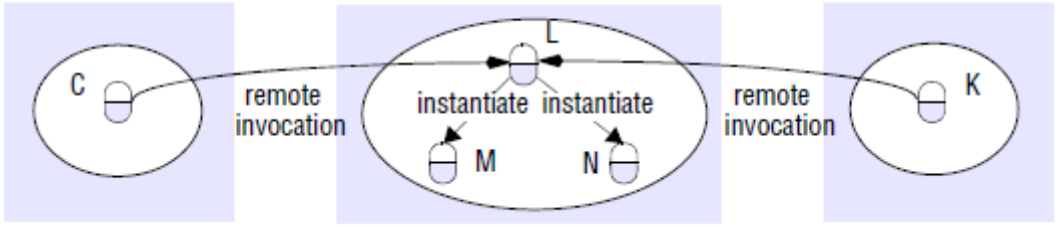
Remote object references: The notion of object reference is extended to allow any object that can receive an RMI to have a remote object reference. A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object.

Remote interfaces: The class of a remote object implements the methods of its remote interface, for example as public instance methods in Java.
The CORBA system provides an interface definition language (IDL), which is used for defining remote interfaces.

A remote object and its remote interface



Instantiation of remote objects

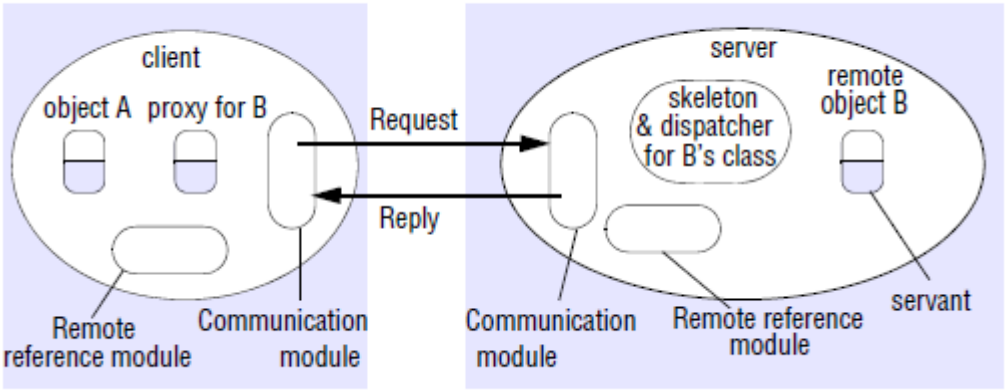


Implementation of RMI

Communication module

The two cooperating communication modules carry out the request-reply protocol, which transmits request and reply messages between the client and server. The operationId and all the marshalling and unmarshalling are the concern of the RMI software, discussed below. The communication modules are together responsible for providing a specified invocation semantics, for example at-most-once.

The role of proxy and skeleton in remote method invocation



Remote reference module

A remote reference module is responsible for translating between local and remote object references and for creating remote object references. To support its responsibilities, the remote reference module in each process has a remoteobject table that records the correspondence between local object references in that process and remote object references

Servants

A servant is an instance of a class that provides the body of a remote object. It is the servant that eventually handles the remote requests passed on by the corresponding skeleton. Servants live within a server process. They are created when remote objects are instantiated and remain in use until they are no longer needed, finally being garbage collected or deleted.

Proxy
The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker; but instead of executing an invocation, it forwards it in a message to a remote object. It hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client. There is one proxy for each remote object for which a process holds a remote object reference.

Dispatcher
A server has one dispatcher and one skeleton for each class representing a remote object. In our example, the server has a dispatcher and a skeleton for the class of remote object B. The dispatcher receives request messages from the communication module. It uses the operationId to select the appropriate method in the skeleton, passing on the request message. The dispatcher and the proxy use the same allocation of operationIds to the methods of the remote interface.

Skeleton
The class of a remote object has a skeleton, which implements the methods in the remote interface. They are implemented quite differently from the methods in the servant that incarnates a remote object.

8. Explain group communication and its implementation issues.

Group communication offers a service whereby a message is sent to a groupand then this message is delivered to all members of the group. In this action, the senderis not aware of the identities of the receivers.

Group communication is an important building block for distributed systems, andparticularly reliable distributed systems, with key areas of application including:

- the reliable dissemination of information to potentially large numbers of clients,including in the financial industry, where institutions require accurate and up-todateaccess to a wide variety of information sources;
- support for collaborative applications, where again events must be disseminatedto multiple users to preserve a common user view – for example, in multiusergames;
- support for a range of fault-tolerance strategies, including the consistent update ofreplicated data or the implementation ofhighly available (replicated) servers;
- support for system monitoring and management, including for example loadbalancing strategies.

Implementation issues

a.Reliability and ordering in multicast

In group communication, all members of a groupmust receive copies of the messages sent to the group, generally with deliveryguarantees.

Reliability in one-to-one communication in terms oftwo properties: integrity (the message received is the same as the one sent, and nomessages are delivered twice) and validity (any outgoing message is eventuallydelivered). To extend thesemantics to cover delivery to multiple receivers, a third property is added – that ofagreement, stating that if the message is delivered to one process, then it is delivered toall processes in the group.

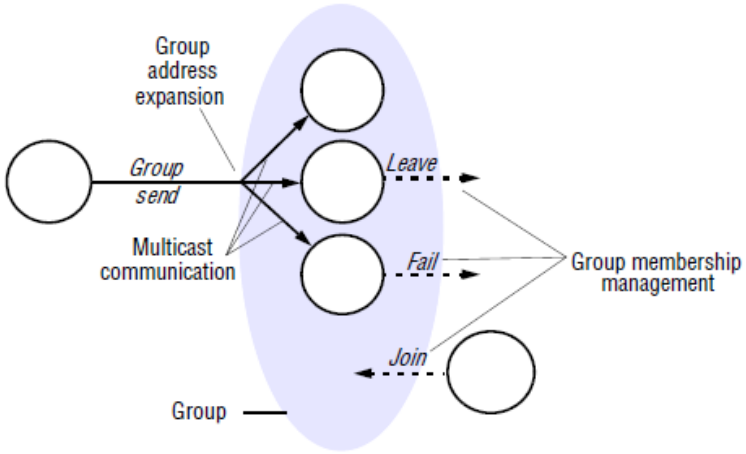
Orderingis not guaranteed by underlying interprocess communication primitives. To counter this,group communication services offer ordered multicast, with the option of one or moreof the following properties (with hybrid solutions also possible):

- **FIFO ordering/ sourceordering**: First-in-first-out (FIFO) ordering is concerned with preserving the order from the perspective of a senderprocess, in that if a process sends one message before another, it will be delivered inthis order at all processes in the group.
- **Causal ordering**: Causal ordering takes into account causal relationships betweenmessages, in that if a message happens before another message in the distributedsystem this so-called causal relationship will be preserved in the delivery of theassociated messages at all processes.
- **Total ordering**: In total ordering, if a message is delivered before another messageat one process, then the same order will be preserved at all processes.

b.Group membership management

The key elements of group communication management are summarized in Figure 6.3, which shows an open group.

Figure 6.3 The role of group membership management



A group membership service has four main tasks:

Providing an interface for group membership changes: The membership serviceprovides operations to create and destroy process groups and to add or withdraw aprocess to or from a group.

Failure detection: The service monitors the group members not only in case theyshould crash, but also in case they should become unreachable because of acommunication failure.

Notifying members of group membership changes: The service notifies the group'smembers when a process is added, or when a process is excluded.

Performing group address expansion: When a process multicasts a message, itsupplies the group identifier rather than a list of processes in the group. Themembership management service expands the identifier into the current group membership for delivery. The service can coordinate multicast delivery withmembership changes by controlling address expansion. That is, it can decideconsistently where to deliver any given message, even though the membership maybe changing during delivery.

9. Discuss about the publish-subscribe systems.

A publish-subscribe system is a system where publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events.

For example, a subscriber could express an interest in all events related to this textbook, such as the availability of a new edition or updates to the related web site. The task of the publish subscribe system is to match subscriptions against published events and ensure the correct delivery of event notifications. A given event will be delivered to potentially many subscribers, and hence publish-subscribe is fundamentally a one-to-many communications paradigm.

Applications of publish-subscribe systems

- financial information systems;
- other areas with live feeds of real-time data (including RSS feeds);
- support for cooperative working, where a number of participants need to be informed of events of shared interest;
- support for ubiquitous computing, including the management of events emanating from the ubiquitous infrastructure (for example, location events);
- a broad set of monitoring applications, including network monitoring in the Internet.

Publish-subscribe is also a key component of Google's infrastructure, including for example the dissemination of events related to advertisements, such as 'ad clicks', to interested parties.

To illustrate the concept further, we consider a simple dealing room system as an example of the broader class of financial information systems.

Characteristics of publish-subscribe systems

Heterogeneity

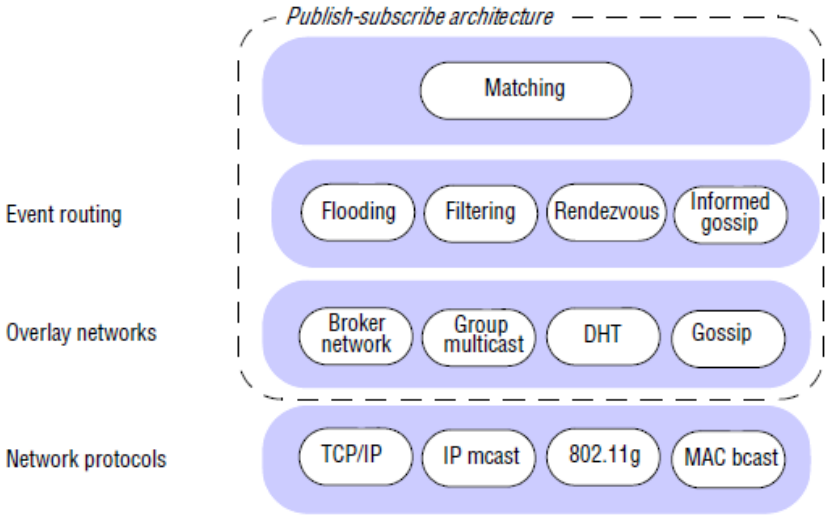
When event notifications are used as a means of communication, components in a distributed system that were not designed to interoperate can be made to work together. All that is required is that event-generating objects publish the types of events they offer, and provide an interface for receiving and dealing with the resultant notifications.

Asynchronicity

Notifications are sent asynchronously by event-generating publishers to all the subscribers that have expressed an interest in them.

Overall systems architecture

Figure 6.10 The architecture of publish-subscribe systems



In the bottom layer, publish-subscribe systems make use of a range of interprocess communication services, such as TCP/IP, IP multicast (where available) or more specialized services, as offered for example by wireless networks. The heart of the architecture is provided by the event routing layer supported by a network overlay infrastructure. Event routing performs the task of ensuring that event notifications are routed as efficiently as possible to appropriate subscribers, whereas the overlay infrastructure supports this by setting up appropriate networks of brokers or peer-to-peer structures. The top layer implements matching – that is, ensuring that events match a given subscription.

Within this overall architecture, there is a wide variety of implementation approaches. We illustrate the general principles behind content-based routing:

Flooding: The simplest approach is based on flooding, that is, sending an event notification to all nodes in the network and then carrying out the appropriate matching at the subscriber end. This approach has the benefit of simplicity but can result in a lot of unnecessary network traffic.

Filtering/ filtering-based routing: Apply filtering in the network of brokers. Brokers forward notifications through the network only where there is a path to a valid subscriber.

Advertisements: The pure filtering-based approach described above can generate a lot of traffic due to propagation of subscriptions, with subscriptions essentially using a flooding approach back towards all possible publishers. In systems with advertisements this burden can be reduced by propagating the advertisements towards subscribers in a similar (actually, symmetrical) way to the propagation of subscriptions.

Rendezvous: Another approach to control the propagation of subscriptions (and to achieve a natural load balancing) is the rendezvous approach. To understand this approach, it is necessary to view the set of all possible events as an event space and to partition responsibility for this event space between the set of brokers in the network. In particular, this approach defines rendezvous nodes, which are broker nodes responsible for a given subset of the event space.

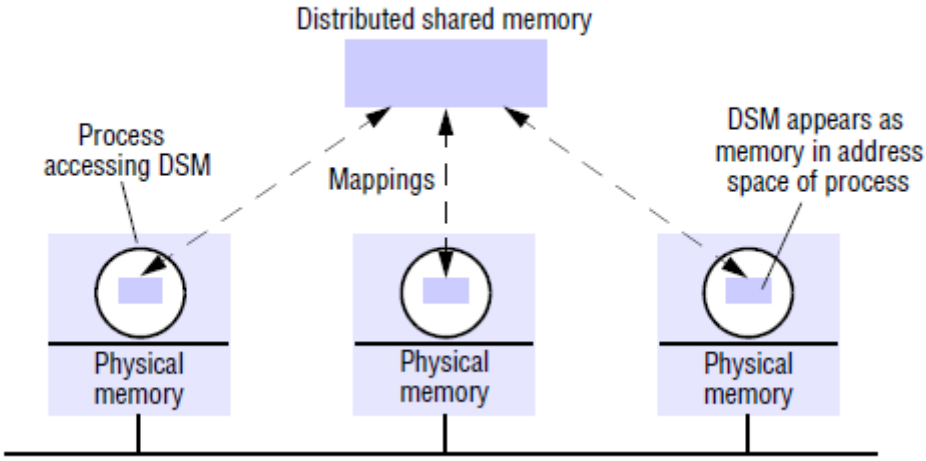
10. Give a detailed description about the various shared-memory approaches.

Distributed Shared memory

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that

do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another. It is as though the processes access a single shared memory, but in fact the physical memory is distributed.

The distributed shared memory abstraction



The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. Message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access.

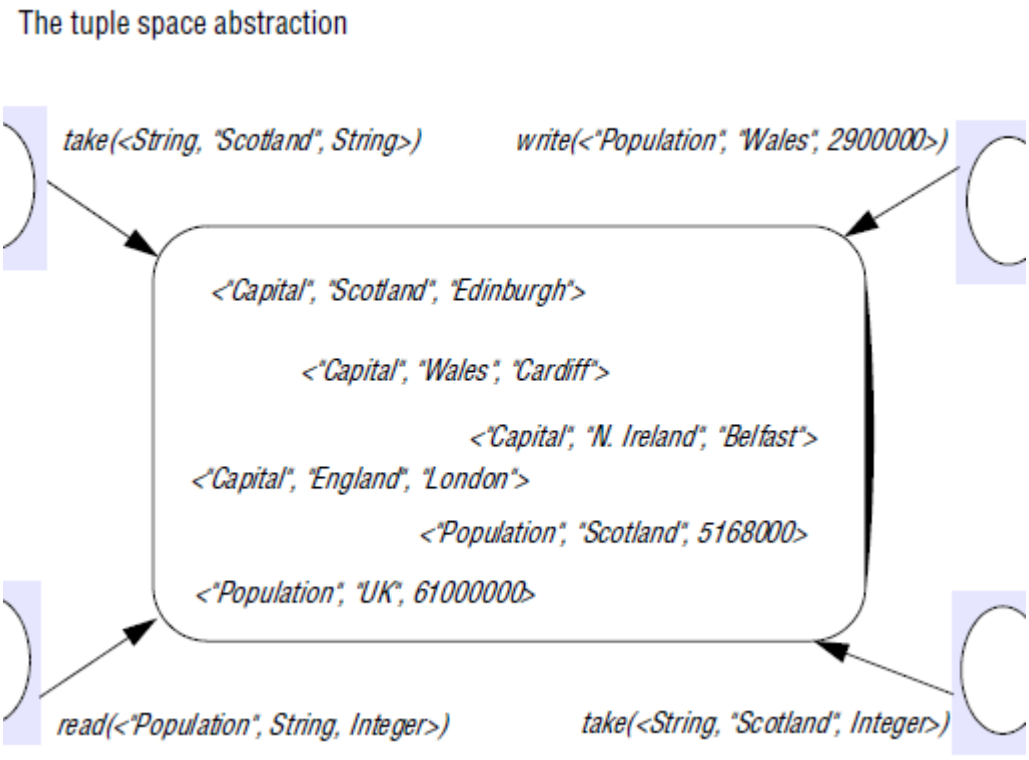
As a communication mechanism, DSM is comparable with message passing rather than with request-reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication.
The DSM and message-passing approaches to programming can be contrasted as follows:

Service offered: Under the message-passing model, variables have to be marshaled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory the processes involved share variables directly, so no marshalling is necessary – even of pointers to shared variables and thus no separate communication operations are necessary. Most implementations
allow variables stored in DSM to be named and accessed similarly to ordinary unshared variables.
Efficiency
Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message-passing platforms on the same hardware .

Tuple Space Communication
Tuple spaces were first introduced by David Gelernter from Yale University as a novel form of distributed computing based on what he refers to as *generative communication* In this approach, processes communicate indirectly by placing tuples in a tuple space, from which other processes can read or remove them. Tuples do not have an address but are accessed by pattern matching on content

In the tuple space programming model, processes communicate through a tuple space – a shared collection of tuples. Tuples in turn consist of a sequence of one or more typed data fields such as *<"fred", 1958>*, *<"sid", 1964>* and *<4, 9.8, "Yes">*. Any combination of types of tuples may exist in the same tuple space. Processes share data by accessing the same tuple space: they place

tuples in tuple space using the *write* operation and read or extract them from tuple space using the *read* or *take* operation. The *write* operation adds a tuple without affecting existing tuples in the space. The *read* operation returns the value of one tuple without affecting the contents of the tuple space. The *take* operation also returns a tuple, but in this case it also removes the tuple from the tuple space.



Properties associated with tuple spaces:

Gelernter [1985] presents some interesting properties associated with tuple space communication, highlighting in particular both space and time uncoupling .

Space uncoupling: A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of a number of potential recipients. This property is also referred to as *distributed naming* in Linda.

Time uncoupling: A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely), and hence the sender and receiver do not need to overlap in time

Tuple Space Operations

write
1. The requesting site multicasts the *write* request to all members of the view;
2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

read
1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others);
4. Step 1 is repeated until at least one response is received.

take *Phase 1: Selecting the tuple to be removed*
1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple
1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

UNIT 1II
PART A

1. What is the use of middleware?

Middleware is a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers. Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and resource sharing support for distributed applications.

2. Write about the parts available in routing algorithm?

- Routing algorithm must make decisions that determine the route taken by each packet as it travels through the network. In circuit-switched network layers such as X.25 and frame relay networks such as ATM the route is determined whenever a virtual circuit or connection is established.
- In packet-switched network layers such as IP it is determined separately for each packet, and the algorithm must be particularly simple and efficient if it is not to degrade network performance.
- It must dynamically update its knowledge of the network based on traffic monitoring and the detection of configuration changes or failures. This activity is less time-critical; slower and more computation-intensive techniques can be used.

3. Define multicast communication?

It is the implementation of group communication. Multicast communication requires coordination and agreement. The aim is for members of a group to receive copies of messages sent to the group . Many different delivery guarantees are possible
Example: agreement on the set of messages received or on delivery ordering.

4. **What are the Application dependencies of Napster?**

Napster took advantage of the special characteristics of the application for which it was designed in other ways:

- Music files are never updated, all the replicas of files need to remain consistent after updates.
- No guarantees are required concerning the availability of individual files – if amusic file is temporarily unavailable, it can be downloaded later. This reduces the requirement for dependability of individual computers and their connections tothe Internet.

5. **Define Routing overlay.**

In peer-to-peer systems a distributed algorithm known as a routing overlay takes responsibility for locating nodes and objects. The name denotes the fact that the middleware takes the form of a layer that is responsible for routing requests from any client to a host that holds the object to which the request is addressed.

6. **What is a file group?**

A collection of files that can be located on any server or moved between servers while maintaining the same names is a file group. Simillar to a UNIX file system helps with distributing the load of file serving between several servers.File groups have identifiers which are unique throughout the system used to refer file groups and files

7. **What is flat file service interface?**

It is RPC interface used by client modules. It is not normally used directly by user level programs. A field is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested.

8. **Write a note on Andrew file system?**

AFS provides transparent access to remote shared files for unix programs running on workstations. Access to AFS files is via the normal unix file primitives, enabling existing unix programs to access AFS files without modification or recompilation.

9. **Write a note on X.500 directory service?**

It is a directory service. It can be in the same way as a conventional name service but it is primarily used to satisfy descriptive queries, designed to discover the names and attributes of other users or system resources.

10. **What is the use of iterative navigation?**

DNS supports the model known as iterative navigation. To resolve a name, a client presents the name to the local name server, which attempts to resolve it. If the local name server has the name, it returns the result immediately.

11. **Define multicast navigation?**

A client multicast the name to be resolved and the required object type to the group of name servers. Only the server that holds the named attributes responds to the request.

12. **What are the major goals of Sun NFS?**

- NFS should be deployable easily.
- NFS should be efficient enough to be tolerable to users.
- Sun NFS should work with existing applications.
- To achieve a high level of support for hardware andoperating system heterogeneity.

13. **What is a Name Service?**

A name service stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The collection is often subdivided into one or more naming contexts: individual subsets of the bindings that are managed as a unit. The major operation that a name service supports is to resolve a name that is, to look up attributes from a given name.

14. **Define Namespace.**

A name space is the collection of all valid names recognized by a particular service. The service will attempt to look up a valid name, even though that name may prove not to correspond to any object that is to be unbound. Name spaces require a syntactic definition to

separate valid names from invalid names. For example, **'...'** is not acceptable as the DNS name of a computer, whereas www.cdk.net is valid.

15. Illustrate the importance of Caching.

Caching is key to a name service's performance and assists in maintaining the availability of both the name service and other services in spite of name server crashes. Its role in enhancing response times by saving communication with name servers is clear. Caching can be used to eliminate high-level name servers – the root server in particular.

16. Define DNS with examples

The Domain Name System is a name service design whose main naming database is used across the Internet. The objects named by the DNS are primarily computers – for which mainly IP addresses are stored as attributes.

The original top-level organizational domains called as *generic domains* in use across the Internet were:

- *com* – Commercial organizations
- *edu* – Universities and other educational institutions
- *gov* – US governmental agencies
- *mil* – US military organizations
- *net* – Major network support centres
- *org* – Organizations not mentioned above
- *int* – International organizations

17. Write short notes on Directory Services.

A service that stores collections of bindings between names and attributes and that looks up entries that match attribute-based specifications is called a directory service. Examples are Microsoft's Active Directory Services, X.500 etc. Directory services are sometimes called yellow pages services, and conventional name services are correspondingly called white pages services, in an analogy with the traditional types of telephone directory. Directory services are also sometimes known as attribute-based name services.

18. Write about LDAP.

The Lightweight Directory Access Protocol (LDAP) is a directory service protocol that runs on a layer above the TCP/IP stack. It provides a mechanism used to connect to, search, and modify Internet directories. The LDAP directory service is based on a client-server model. It is an open, vendor-neutral, industry standard application protocol.

19. What are the non-functional requirements that peer-to-peer middleware must address?

- ***Global scalability***: One of the aims of peer-to-peer applications is to exploit the hardware resources of very large numbers of hosts connected to the Internet
- ***Load balancing***: The performance of any system designed to exploit a large number of computers depends upon the balanced distribution of workload across them.
- ***Optimization for local interactions between neighboring peers.*** The middleware should aim to place resources close to the nodes that access them the most.
- ***Accommodating to highly dynamic host availability***: Most peer-to-peer systems are constructed from host computers that are free to join or leave the system at any time.

20. What is the key problem faced in peer-to-peer middleware.

A key problem in the design of peer-to-peer applications is providing a mechanism to enable clients to access data resources quickly and dependably wherever they are located throughout the network. Napster maintained a unified index of available files for this purpose, giving the network addresses of their hosts.

21. What are the characteristics of peer-to-peer systems?

- Its design ensures that each user contributes resources to the system.
- Although they may differ in the resources that they contribute, all the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
- Its correct operation does not depend on the existence of any centrally administered systems.
- A key issue for their efficient operation is the choice of an algorithm for the placement of data across many hosts and subsequent access to it in a manner that balances the workload and ensures availability without adding undue overheads.

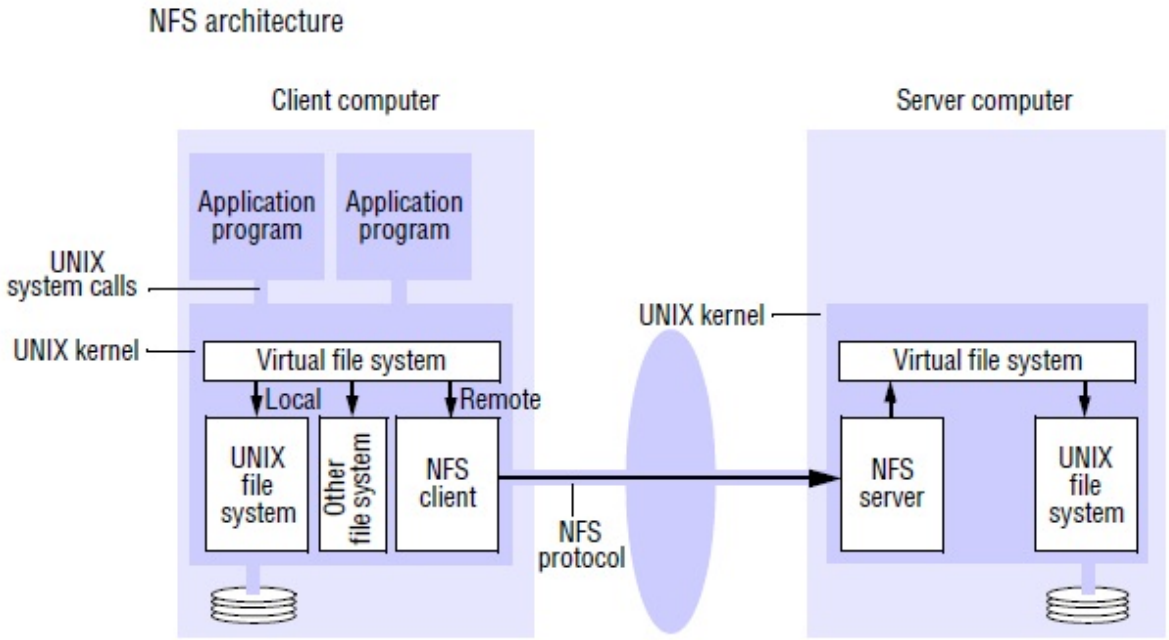
22. What is the use of GUID?

A **Globally Unique Identifier (GUID)** is a unique reference number used as an identifier in computer software. The term GUID typically refers to various implementations of the universally unique identifier (UUID) standard. A GUID can be stored as a 16-byte (128-bit) number. GUIDs are commonly used as the primary key of database tables.

PART B

1. Write a case study on Sun network file system?

The Figureshows the architecture of Sun NFS. It follows the abstract model defined in the preceding section. All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation the NFS protocol (version 3).



The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls. Sun's RPC system, described in Section 5.3.3, was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

Virtual file system • The Figure above makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the filesystems that are currently available both

locally and remotely, and it passes each request to the appropriate local system module (the UNIX file system, the NFS client module or the service module for another file system).

The file identifiers used in NFS are called *file handles*. A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file. In UNIX implementations of NFS, the file handle is derived from the file's *i-nodenumber* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):

File handle:	Filesystem identifier	i-node number of file	i-node generation number

NFS adopts the UNIX mountable filesystem as the unit of file grouping defined in the preceding section. The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system). The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed. In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused (for example, in a UNIX *creat* system call). The client obtains the first file handle for a remote file system when it mounts it. File handles are passed from server to client in the results of *lookup*, *create* and *mkdir* operations and from client to server in the argument lists of all server operations.

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs. But unlike our model client module, it emulates the semantics of the standard UNIX file system primitives precisely and is integrated with the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading into client processes so that:

- User programs can access files via UNIX system calls without recompilation or reloading;
- A single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);
- The encryption key used to authenticate user IDs passed to the server (see below) can be retained in the kernel, preventing impersonation by user-level clients.

Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested. The Sun RPC protocol requires clients to send user authentication information (for example, the conventional UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes. These additional parameters are not shown in our overview of the NFS protocol in Figure 12.9; they are supplied automatically by the RPC system.

In its simplest form, there is a security loophole in this access-control mechanism. An NFS server provides a conventional RPC interface at a well-known port on each host and any process can behave as a client, sending requests to the server to access or update a file. The client can modify the RPC calls to include the user ID of any user, impersonating the user without their knowledge or permission. This security loophole has been closed by the use of an option in the RPC protocol for the DES encryption of the user's authentication information. More recently, Kerberos has been integrated with Sun NFS to provide a stronger and more comprehensive solution to the problems of user authentication and security.

NFS server interface • A simplified representation of the RPC interface provided by NFS version 3 servers (defined in RFC 1813 [Callaghan *et al.* 1995]) is shown in Figure below.

NFS server operations (NFS version 3 protocol, simplified)

lookup(dirfh, name) → fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → newfh, attr	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → status	Removes file *name* from directory *dirfh*.
getattr(fh) → attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) → attr	Sets the attributes (mode, user ID, group ID, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) → attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) → attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) → status	Changes the name of file *name* in directory *dirfh* to *toname* in directory *todirfh*.
link(newdirfh, newname, fh) → status	Creates an entry *newname* in the directory *newdirfh* that refers to the file *or* directory *fh*.
symlink(newdirfh, newname, string) → status	Creates an entry *newname* in the directory *newdirfh* of type *symbolic link* with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) → string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) → newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) → status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) → entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) → fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

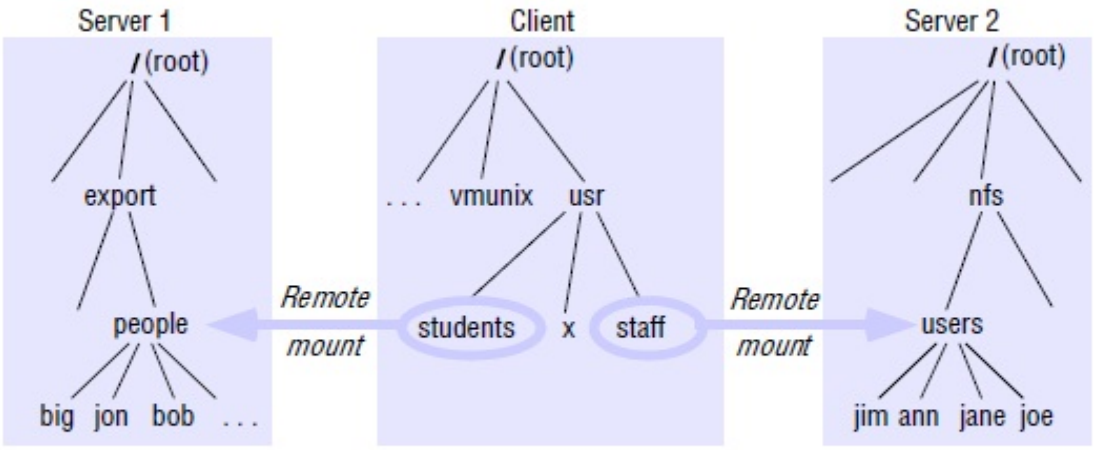
The file and directory operations are integrated in a single service; the creation and insertion of file names in directories is performed by a single *create* operation, which takes the text name of the new file and the file handle for the target directory as arguments. The other NFS operations on directories are *create*, *remove*, *rename*, *link*,
symlink, *readlink*, *mkdir*, *rmdir*, *readdir* and *statfs*. They resemble their UNIX counterparts with the exception of *readdir*, which provides a representationindependent method for reading the contents of directories, and *statfs,* which gives the status information on remote file systems.

Mount service • The mounting of subtrees of remote filesystems by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local filesystems that are available for remote mounting. An access list is associated with each filesystem name indicating which hosts are permitted to mount the
filesystem.

Clients use a modified version of the UNIX *mount* command to request mounting of a remote filesystem, specifying the remote host's name, the pathname of a directory in the remote filesystem and the local name with which it is to be mounted. The remote directory may be any subtree of the required remote filesystem, enabling clients to mount any part of the remote filesystem. The modified *mount* command communicates with the mount service process on the remote host using a *mount protocol*. This is an RPC protocol and includes an operation that takes a directory pathname and returns the file handle of the specified directory if the client has access permission for the relevant filesystem. The location (IP address and port number) of the server and the file handle for the remote directory are passed on to the VFS layer and the NFS client.

Figure below illustrates a *Client* with two remotely mounted file stores. The nodes *people* and *users* in filesystems at *Server 1* and *Server 2* are mounted over nodes *students* and *staff* in *Client*'s local file store. The meaning of this is that programs running at *Client* can access files at *Server 1* and *Server 2* by using pathnames such as */usr/students/jon* and */usr/staff/ann*.

Local and remote filesystems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the subtree located at */export/people* in Server 1; the filesystem mounted at */usr/staff* in the client is actually the subtree located at /nfs/users in Server 2.

Pathname translation • UNIX file systems translate multi-part file pathnames to i-node references in a step-by-step process whenever the *open*, *creat* or *stat* system calls are used. In NFS, pathnames cannot be translated at a server, because the name may cross a 'mount point' at the client – directories holding different parts of a multi-part name may reside in filesystems at different servers. So pathnames are parsed, and their translation is performed in an iterative manner by the client. Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate *lookup* request to the remote server.

Automounter • The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client. The original implementation of the automounter ran as a userlevel UNIX process in each client computer. Later versions (called *autofs*) were implemented in the kernel for Solaris and Linux.

Server caching • Caching in both the client and the server computer are indispensable features of NFS implementations in order to achieve adequate performance.

In conventional UNIX systems, file pages, directories and file attributes that have been read from disk are retained in a main memory *buffer cache* until the buffer space is required for other pages. If a process then issues a read or a write request for a page that is already in the cache, it can be satisfied without another disk access. *Read-ahead* anticipates read accesses and fetches the pages following those that have most recently been read, and *delayed-write* optimizes writes: when a page has been altered (by a write request), its new contents are written to disk only when the buffer page is required for another page. To guard against loss of data in a system crash, the UNIX *sync* operation flushes altered pages to disk every 30 seconds. These caching techniques work in a conventional UNIX environment because all read and write requests issued by userlevel processes pass through a single

cache that is implemented in the UNIX kernel space. The cache is always kept up-to-date, and file accesses cannot bypass the cache.

Client caching • The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes, because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.

2. Briefly explain about the case study on Andrew file system?

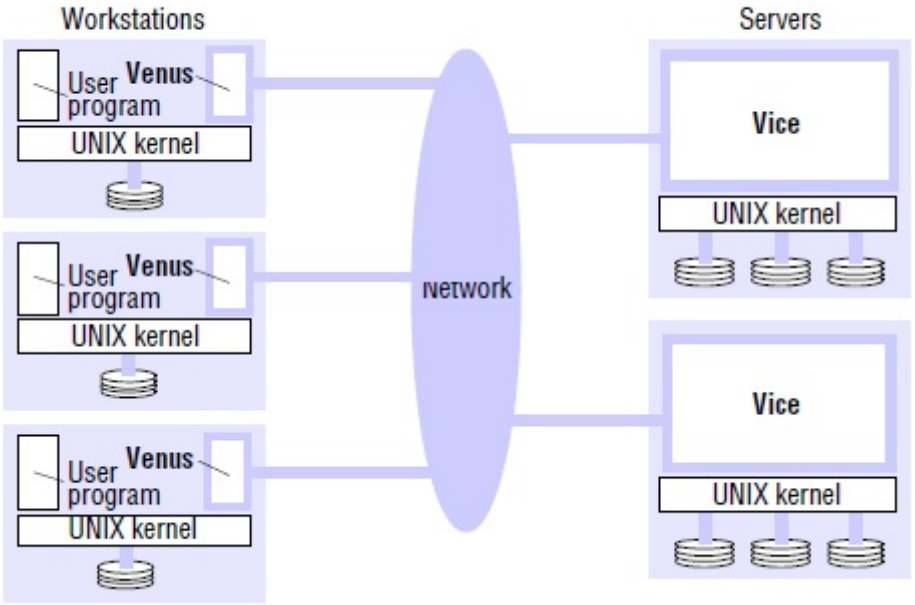
AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy 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.

Implementation

AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*. Figure below shows the distribution of Vice and Venus processes. Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.

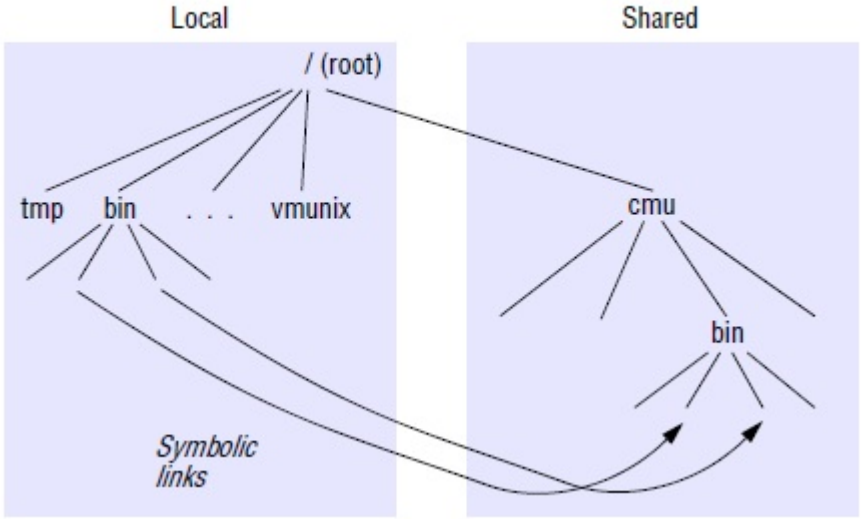
Distribution of processes in the Andrew File System



The files available to user processes running on workstations are either *local* or *shared*. Local files are handled as normal UNIX files. They are stored on a workstation's disk and are available only to local user processes. Shared files are stored on servers, and copies of them are cached on the local

disks of workstations. The name space seen by user processes is illustrated in Figure below. It is a conventional UNIX directory hierarchy, with a specific subtree (called *cmu*) containing all of the shared files. This splitting of the file name space into local and shared files leads to some loss of location transparency, but this is hardly noticeable to users other than system administrators. Local files are used only for temporary files (*/tmp*) and processes that are essential for workstation startup. Other standard UNIX files (such as those normally found in */bin*, */lib* and so on) are implemented as symbolic links from local directories to files held in the shared space. Users' directories are in the shared space, enabling users to access their files from any workstation.

File name space seen by clients of AFS



The UNIX kernel in each workstation and server is a modified version of BSD UNIX. The modifications are designed to intercept *open*, *close* and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer (illustrated in Figure below).

System call interception in AFS

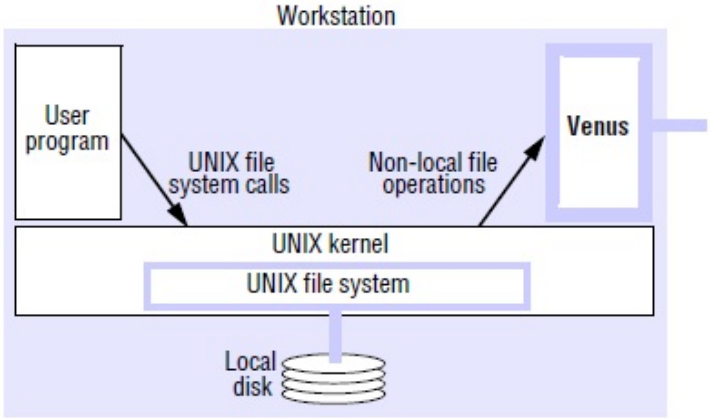


Figure below describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues each of the system calls mentioned in our outline scenario above. The *callback promise* mentioned here is a mechanism for ensuring that cached copies of files are updated when another client closes the same file after updating it.

Implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Cache consistency

When Vice supplies a copy of a file to a Venus process it also provides a *callback promise* – a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file. Callback promises are stored with the cached files on the workstation disks and have two states: *valid* or *cancelled*. When a server performs a request to update a file it notifies all of the Venus processes to which it has issued callback promises by sending a *callback* to each – a callback is a remote procedure call from a server to a Venus process. When the Venus process receives a callback, it sets the *callback promise* token for the relevant file to *cancelled*.

Whenever Venus handles an *open* on behalf of a client, it checks the cache. If the required file is found in the cache, then its token is checked. If its value is *cancelled*, then a fresh copy of the file must be fetched from the Vice server, but if the token is *valid*, then the cached copy can be opened and used without reference to Vice. Figure below shows the RPC calls provided by AFS servers for operations on files.

The main components of the Vice service interface

Fetch(fid) → attr, data	Returns the attributes (status) and, optionally, the contents of the file identified by *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() → fid	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs the server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	Call made by a Vice server to a Venus process; cancels the callback promise on the relevant file.

Note: Directory and administrative operations (*Rename*, *Link*, *Makedir*, *Removedir*, *GetTime*, *CheckToken* and so on) are not shown.

Update semantics • The goal of this cache-consistency mechanism is to achieve the best approximation to one-copy file semantics that is practicable without serious performance degradation. A strict implementation of one-copy semantics for UNIX file access primitives would require that the results of each *write* to a file be distributed to all sites holding the file in their cache before any further accesses can occur. This is not practicable in large-scale systems; instead, the callback promise mechanism maintains a well-defined approximation to one-copy semantics.

For AFS-1, the update semantics can be formally stated in very simple terms. For a client *C* operating on a file *F* whose custodian is a server *S*, the following guarantees of currency for the copies of *F* are maintained:

- After a successful *open*: *latest*(*F, S*)
- After a failed *open*: *failure*(*S*)
- After a successful *close*: *updated*(*F, S*)
- After a failed *close*: *failure*(*S*)

3. Write a case study on X.500 directory service?

X.500 is a directory service which can be used in the same way as a conventional name service, but it is primarily used to satisfy descriptive queries and is designed to discover the names and attributes of other users or system resources. Users may have a variety of requirements for searching and browsing in a directory of network users, organizations and system resources to obtain information about the entities that the directory contains. The uses for such a service are likely to be quite diverse. They range from enquiries that are directly analogous to the use of telephone directories, such as a simple 'white pages' access to obtain a user's electronic mail address or a 'yellow pages' query aimed, for example, at obtaining the names and telephone numbers of garages specializing in the repair of a particular make of car, to the use of the directory to access personal details such as job roles, dietary habits or even photographic images of the individuals.

The ITU and ISO standards organizations defined the *X.500 Directory Service* [ITU/ISO 1997] as a network service intended to meet these requirements. The standard refers to it as a service for access to information about 'real-world entities', but it is also likely to be used for access to information about hardware and software services and devices. X.500 is specified as an application-level service in the Open Systems Interconnection (OSI) set of standards, but its design does not depend to any

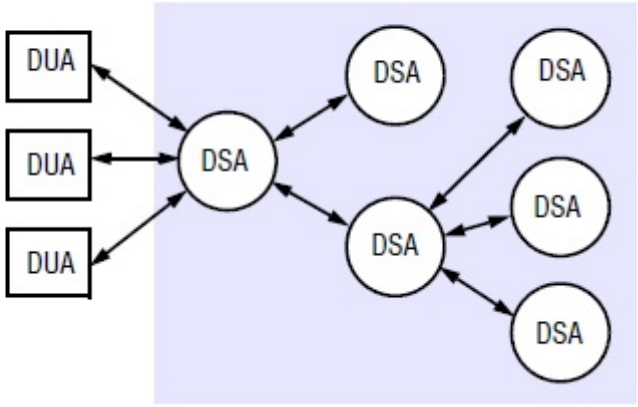
significant extent on the other OSI standards, and it can be viewed as a design for a general-purpose directory service. We outline the design of the X.500 directory service and its implementation here. Readers interested in a more detailed description of X.500 and methods for its implementation are advised to study Rose's book on the subject [Rose 1992]. X.500 is also the basis for LDAP, and it is used in the DCE directory service.

The data stored in X.500 servers is organized in a tree structure with named nodes, as in the case of the other name servers discussed in this chapter, but in X.500 a wide range of attributes are stored at each node in the tree, and access is possible not just by name but also by searching for entries with any required combination of attributes.

The X.500 name tree is called the *Directory Information Tree* (DIT), and the entire directory structure including the data associated with the nodes, is called the *Directory Information Base* (DIB). There is intended to be a single integrated DIB containing information provided by organizations throughout the world, with portions of the DIB located in individual X.500 servers. Typically, a medium-sized or large organization would provide at least one server. Clients access the directory by establishing a connection to a server and issuing access requests. Clients can contact any server with an enquiry. If the data required are not in the segment of the DIB held by the contacted server, it will either invoke other servers to resolve the query or redirect the client to another server.

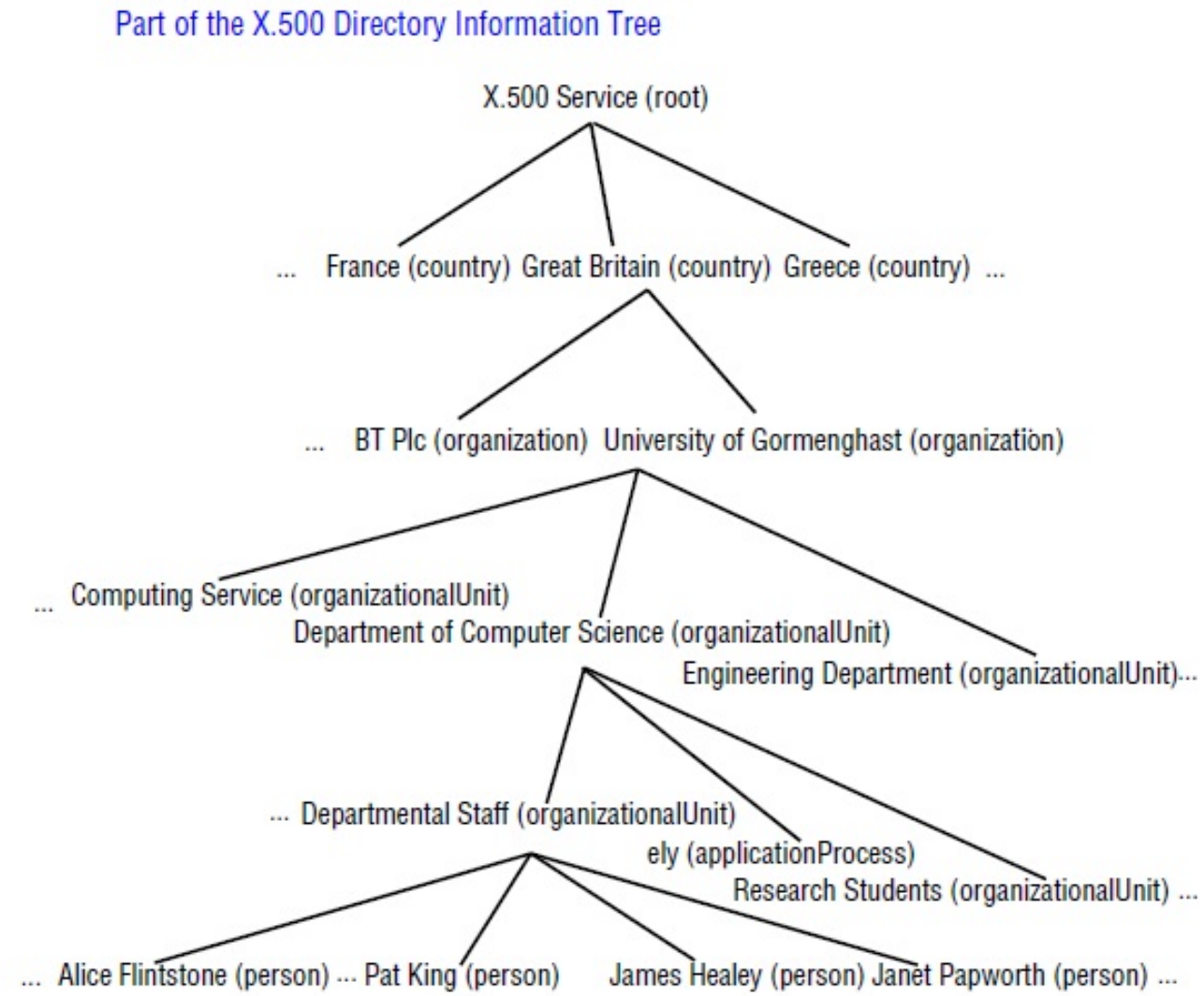
In the terminology of the X.500 standard, servers are *Directory Service Agents*(DSAs), and their clients are termed *Directory User Agents* (DUAs). Figure below shows the software architecture and one of the several possible navigation models, with each DUA client process interacting with a single DSA process, which accesses other DSAs as necessary to satisfy requests.

X.500 service architecture



Each entry in the DIB consists of a name and a set of attributes. As in other name servers, the full name of an entry corresponds to a path through the DIT from the root of the tree to the entry. In addition to full or *absolute* names, a DUA can establish a context, which includes a base node, and then use shorter relative names that give the path from the base node to the named entry.

Figure below shows the portion of the Directory Information Tree that includes the notional University of Gormenghast in Great Britain,



and Figure below is one of the associated DIB entries. The data structure for the entries in the DIB and the DIT is very flexible. A DIB entry consists of a set of attributes, where an attribute has a *type* and one or more *values*. The type of each attribute is denoted by a type name (for example, *countryName*, *organizationName*, *commonName*, *telephoneNumber*, *mailbox*, *objectClass*). New attribute types can be defined if they are required. For each distinct type name there is a corresponding type definition, which includes a type description and a syntax definition in the ASN.1 notation (a standard notation for syntax definitions) defining representations for all permissible values of the type.

An X.500 DIB Entry

info	
Alice Flintstone, Departmental Staff, Department of Computer Science, University of Gormenghast, GB	
commonName	*uid*
Alice.L.Flintstone	alf
Alice.Flintstone	*mail*
Alice Flintstone	alf@dcs.gormenghast.ac.uk
A. Flintstone	Alice.Flintstone@dcs.gormenghast.ac.uk
surname	*roomNumber*
Flintstone	Z42
telephoneNumber	*userClass*
+44 986 33 4604	Research Fellow

The name of a DIB entry (the name that determines its position in the DIT) is determined by selecting one or more of its attributes as *distinguished attributes*. The attributes selected for this purpose are referred to as the entry's *Distinguished Name*(DN).
Now we can consider the methods by which the directory is accessed. There are two main types of access request:

- *read*: An absolute or relative name (a *domain name* in X.500 terminology) for an entry is given, together with a list of attributes to be read (or an indication that all attributes are required). The DSA locates the named entry by navigating in the DIT, passing requests to other DSA servers where it does not hold relevant parts of the tree. It retrieves the required attributes and returns them to the client.
- *search*: This is an attribute-based access request. A base name and a filter expression are supplied as arguments. The base name specifies the node in the DIT from which the search is to commence; the filter expression is a boolean expression that is to be evaluated for every node below the base node. The filter specifies a search criterion: a logical combination of tests on the values of any of the attributes in an entry. The *search* command returns a list of names (domain names) for all of the entries below the base node for which the filter evaluates to *TRUE*.

Administration and updating of the DIB • The DSA interface includes operations for adding, deleting and modifying entries. Access control is provided for both queries and updating operations, so access to parts of the DIT may be restricted to certain users or classes of user.

The DIB is partitioned, with the expectation that each organization will provide at least one server holding the details of the entities in that organization. Portions of the DIB may be replicated in several servers.

Lightweight Directory Access Protocol • X.500's assumption that organizations would provide information about themselves in public directories within a common system has proved largely unfounded. Equally, its compexity has meant that its uptake has been relatively modest.
A group at the University of Michigan proposed a more lightweight approach called the Lightweight Directory Access Protocol (LDAP), in which a DUA accesses X.500 directory services directly over TCP/IP instead of the upper layers of the ISO protocol stack. This is described in RFC 2251 [Wahl et al. 1997]. LDAP also simplifies the interface to X.500 in other ways: for example, it provides a relatively simple API and it replaces ASN.1 encoding with textual encoding.

4. Discuss about the namespaces and naming resolution techniques.

A *name service* stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The collection is often subdivided into one or more naming *contexts*: individual subsets

of the bindings that are managed as a unit. The major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name.

Name spaces

A *name space* is the collection of all valid names recognized by a particular service. The service will attempt to look up a valid name, even though that name may prove not to correspond to any object – i.e., to be *unbound*. Name spaces require a syntactic definition to separate valid names from invalid names. For example, '...' is not acceptable as the DNS name of a computer, whereas *www.cdk99.net* is valid (even though it is unbound).

Names may have an internal structure that represents their position in a hierarchic name space such as pathnames in a file system, or in an organizational hierarchy such as Internet domain names; or they may be chosen from a flat set of numeric or symbolic identifiers. One important advantage of a hierarchy is that it makes large name spaces more manageable. Each part of a hierarchic name is resolved relative to a separate context of relatively small size, and the same name may be used with different meanings in different contexts, to suit different situations of use. In the case of file systems, each directory represents a context. Thus */etc/passwd* is a hierarchic name with two components. The first, 'etc', is resolved relative to the context '/', or root, and the second part, 'passwd', is relative to the context '/*etc*'. The name */oldetc/passwd* can have a different meaning because its second component is resolved in a different context. Similarly, the same name */etc/passwd* may resolve to different files in the contexts of two different computers.

The DNS name space has a hierarchic structure: a domain name consists of one or more strings called *name components* or *labels*, separated by the delimiter '.'. There is no delimiter at the beginning or end of a domain name, although the root of the DNS name space is sometimes referred to as '.' for administrative purposes. The name components are non-null printable strings that do not contain '.'. In general, a *prefix* of a name is an initial section of the name that contains only zero or more entire components. For example, in DNS *www* and *www.cdk5* are both prefixes of *www.cdk5.net*. DNS names are not case-sensitive, so *www.cdk5.net* and *WWW.CDK5.NET* have the same meaning.

Aliases • An *alias* is a name defined to denote the same information as another name, similar to a symbolic link between file path names. Aliases allow more convenient names to be substituted for relatively complicated ones, and allow alternative names to be used by different people for the same entity. An example is the common use of URL
shorteners, often used in Twitter posts and other situations where space is at a premium. For example, using web redirection, *http://bit.ly/ctqjvH* refers to *http://cdk5.net/additional/rmi/programCode/ShapeListClient.java*. As another example, the DNS allows aliases in which one domain name is defined to stand for another. Aliases are often used to specify the names of machines that run a web server or an FTP server. For example, the name *www.cdk5.net* is an alias for *cdk5.net*. This has the advantage that clients can use either name for the web server, and if the web server is moved to another computer, only the entry for *cdk5.net* needs to be updated in the DNS database.

Naming domains • A *naming domain* is a name space for which there exists a single overall administrative authority responsible for assigning names within it. This authority is in overall control of which names may be bound within the domain, but it is free to delegate this task.

Name resolution

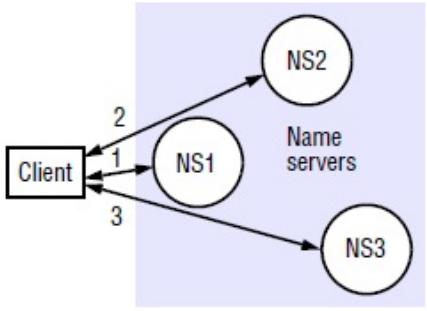
For the common case of hierarchic name spaces, name resolution is an iterative or recursive process whereby a name is repeatedly presented to naming contexts in order to look up the attributes to which it refers. A naming context either maps a given name onto a set of primitive attributes (such as those of a user) directly, or maps it onto a further naming context and a derived name to be presented to that context. To resolve a name, it is first presented to some initial naming context; resolution iterates as long as further contexts and derived names are output.

Name servers and navigation • Any name service, such as DNS, that stores a very large database and is used by a large population will not store all of its naming information on a single server computer. Such a server would be a bottleneck and a critical point of failure. Any heavily used name

services should use replication to achieve high availability. We shall see that DNS specifies that each subset of its database is replicated in at least two failure-independent servers.

One navigation model that DNS supports is known as *iterative navigation* (see Figure below). To resolve a name, a client presents the name to the local name server, which attempts to resolve it. If the local name server has the name, it returns the result immediately. If it does not, it will suggest another server that will be able to help. Resolution proceeds at the new server, with further navigation as necessary until the name is located or is discovered to be unbound.

Iterative navigation

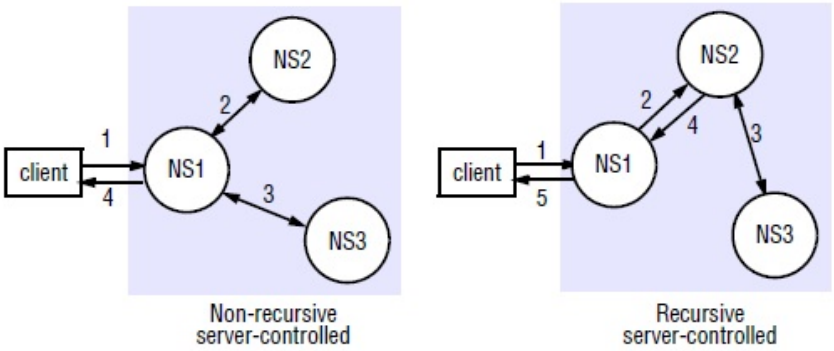


A client iteratively contacts name servers NS1–NS3 in order to resolve a name

In *multicast navigation*, a client multicasts the name to be resolved and the required object type to the group of name servers. Only the server that holds the named attributes responds to the request.

Another alternative to the iterative navigation model is one in which a name server coordinates the resolution of the name and passes the result back to the user agent. Ma [1992] distinguishes *non-recursive* and *recursive server-controlled navigation* (Figure 13.3). Under non-recursive server-controlled navigation, any name server may be chosen by the client. This server communicates by multicast or iteratively with its peers in the style described above, as though it were a client. Under recursive server-controlled navigation, the client once more contacts a single server. If this server does not store the name, the server contacts a peer storing a (larger) prefix of the name, which in turn attempts to resolve it. This procedure continues recursively until the name is resolved.

Non-recursive and recursive server-controlled navigation

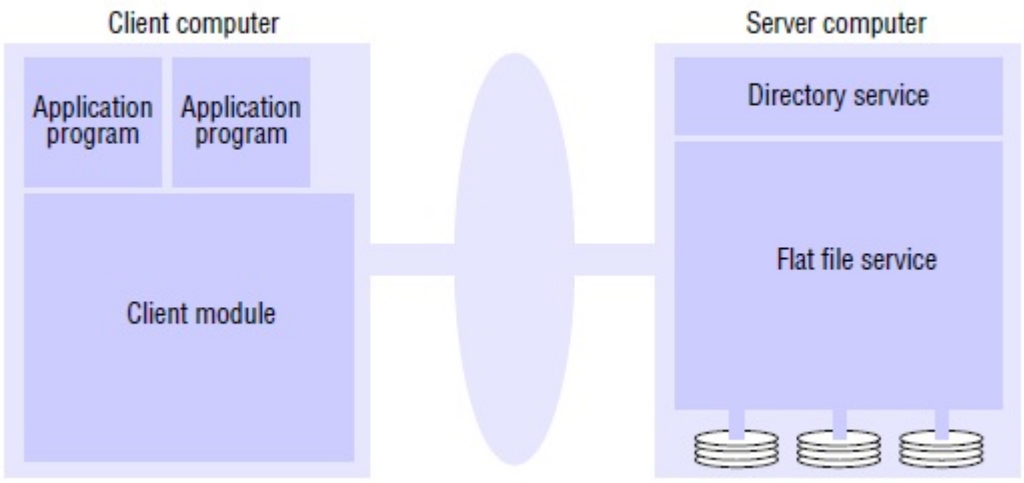


A name server NS1 communicates with other name servers on behalf of a client

5. Describe about the modules of in File service architecture.

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a *flat file service*, a *directory service* and a *client module*. The relevant modules and their relationships are shown in Figure below.

File service architecture



The division of responsibilities between the modules can be defined as follows:

Flat file service • The flat file service is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Directory service • The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

Client module • A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Flat file service interface • Figure below contains a definition of the interface to a flat file service. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId* argument contains an invalid UFID or the user doesn't have sufficient access rights. These exceptions are omitted from the definition for clarity.

Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of *n* data items beginning at item *i* from the specified file into *Data*, which is then returned to the client. The *Write* operation copies the sequence
of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position and extending the file if necessary.
Create creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.
GetAttributes and *SetAttributes* enable clients to access the attribute record. *GetAttributes* is normally available to any client that is allowed to read the file. Access to the *SetAttributes* operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the attribute record are not affected by *SetAttributes*; they are maintained separately by the
flat file service itself.

Directory service interface • Figure below contains a definition of the RPC interface to a directory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

Directory service operations

Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.

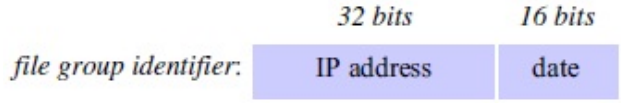
Hierarchic file system • A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a

strict hierarchy – files can have several names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a 'well-known' UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refer to a software component that provides access to files.) File groups were originally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:

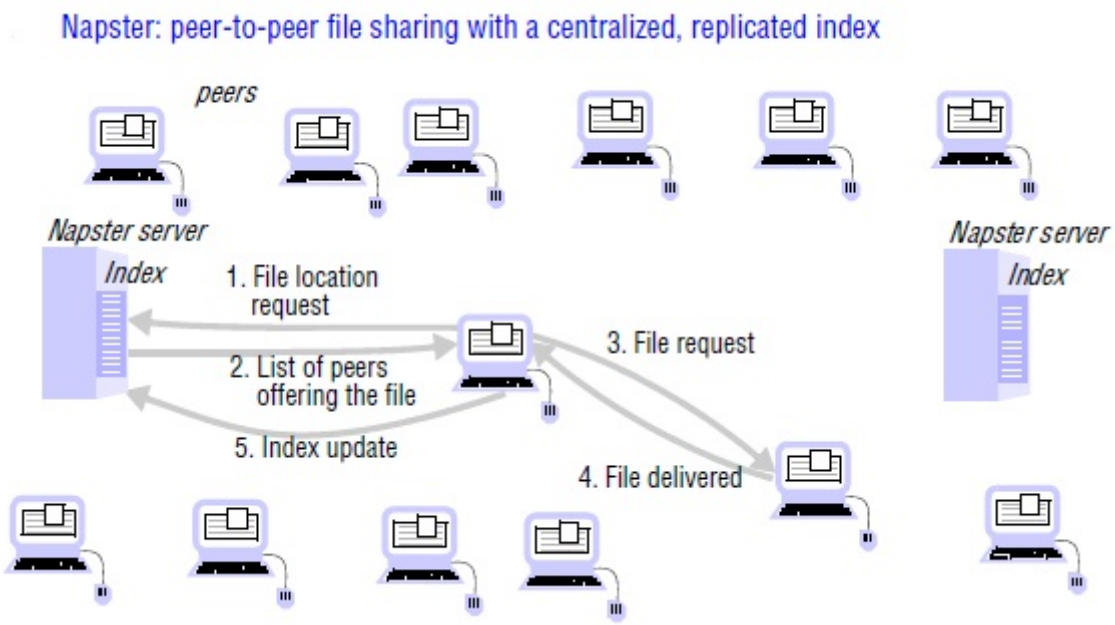


Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

6. Write in detail about Napster and its legacy.

The first application in which a demand for a globally scalable information storage and retrieval service emerged was the downloading of digital music files. Both the need for and the feasibility of a peer-to-peer solution were first demonstrated by the Napster filesharing system [OpenNap 2001] which provided a means for users to share files.

Napster became very popular for music exchange soon after its launch in 1999. At its peak, several million users were registered and thousands were swapping music files simultaneously. Napster's architecture included centralized indexes, but users supplied the files, which were stored and accessed on their personal computers. Napster's method of operation is illustrated by the sequence of steps shown in Figure 10.2. Note that in step 5 clients are expected to add their own music files to the pool of shared resources by transmitting a link to the Napster indexing service for each available file.



Thus the motivation for Napster and the key to its success was the making available of a large, widely distributed set of files to users throughout the Internet, fulfilling Shirky's dictum by providing access to 'shared resources at the edges of the Internet'.

Napster was shut down as a result of legal proceedings instituted against the operators of the Napster service by the owners of the copyright in some of the material (i.e., digitally encoded music) that was made available on it (see the box below).

Anonymity for the receivers and the providers of shared data and other resources is a concern for the designers of peer-to-peer systems. In systems with many nodes, the routing of requests and results can be made sufficiently tortuous to conceal their source and the contents of files can be distributed across multiple nodes, spreading the responsibility for making them available. Mechanisms for anonymous communication that are resistant to most forms of traffic analysis are available [Goldschlag *et al.* 1999]. If files are also encrypted before they are placed on servers, the owners of the servers can plausibly deny any knowledge of the contents. But these anonymity techniques add to the cost of resource sharing, and recent work has shown that the anonymity available is weak against some attacks [Wright *et al.* 2002].

The Freenet [Clarke *et al.* 2000] and FreeHaven [Dingledine *et al.* 2000] projects are focused on providing Internet-wide file services that offer anonymity for the providers and users of the shared files. Ross Anderson has proposed the Eternity Service [Anderson 1996], a storage service that provides long-term guarantees of data availability through resistance to all sorts of accidental data loss and denial of service attacks. He bases the need for such a service on the observation that whereas publication is a permanent state for printed information – it is virtually impossible to delete material once it has been published and distributed to a few thousand libraries in diverse organizations and jurisdictions around the world – electronic publications cannot easily achieve the same level of resistance to censorship or suppression. Anderson covers the technical and economic requirements to ensure the integrity of the store and also points out that anonymity is often an essential requirement for the persistence of information, since it provides the best defence against legal challenges, as well as illegal actions such as bribes or attacks on the originators, owners or keepers of the data.

Lessons learned from Napster •

Napster demonstrated the feasibility of building a useful large-scale service that depends almost wholly on data and computers owned by ordinary Internet users. To avoid swamping the computing resources of individual users (for example, the first user to offer a chart-topping song) and their network connections, Napster took account of network locality – the number of hops between the client and the server – when allocating a server to a client requesting a song. This simple load distribution mechanism enabled the service to scale to meet the needs of large numbers of users.

Limitations: Napster used a (replicated) unified index of all available music files. For the application in question, the requirement for consistency between the replicas was not strong, so this did not hamper performance, but for many applications it would constitute a limitation. Unless the access

path to the data objects is distributed, object discovery and addressing are likely to become a bottleneck.

Application dependencies: Napster took advantage of the special characteristics of the application for which it was designed in other ways:

- Music files are never updated, avoiding any need to make sure all the replicas of files remain consistent after updates.
- No guarantees are required concerning the availability of individual files – if a music file is temporarily unavailable, it can be downloaded later. This reduces the requirement for dependability of individual computers and their connections to the Internet.

7. Describe about the peer-to-peer systems and to role of middleware.

A key problem in the design of peer-to-peer applications is providing a mechanism to enable clients to access data resources quickly and dependably wherever they are located throughout the network. Napster maintained a unified index of available files for this purpose, giving the network addresses of their hosts. Second-generation peer-topeer file storage systems such as Gnutella and Freenet employ partitioned and distributed indexes, but the algorithms used are specific to each system.

This location problem existed in several services that predate the peer-to-peer paradigm as well. For example, Sun NFS addresses this need with the aid of a virtual file system abstraction layer at each client that accepts requests to access files stored on multiple servers in terms of virtual file references. This solution relies on a substantial amount of preconfiguration at each client and manual intervention when file distribution patterns or server provision changes. It is clearly not scalable beyond a service managed by a single organization. AFS has similar properties.

Peer-to-peer middleware systems are designed specifically to meet the need for the automatic placement and subsequent location of the distributed objects managed by peer-to-peer systems and applications.

Functional requirements •

The function of peer-to-peer middleware is to simplify the construction of services that are implemented across many hosts in a widely distributed network. To achieve this it must enable clients to locate and communicate with any individual resource made available to a service, even though the resources are widely distributed amongst the hosts. Other important requirements include the ability to add new resources and to remove them at will and to add hosts to the service and remove them. Like other middleware, peer-to-peer middleware should offer a simple programming interface to application programmers that is independent of the types of distributed resource that the application manipulates.

Non-functional requirements •

To perform effectively, peer-to-peer middleware must also address the following non-functional requirements: *Global scalability*: One of the aims of peer-to-peer applications is to exploit the hardware resources of very large numbers of hosts connected to the Internet. Peer-to peer middleware must therefore be designed to support applications that access millions of objects on tens of thousands or hundreds of thousands of hosts.

Load balancing: The performance of any system designed to exploit a large number of computers depends upon the balanced distribution of workload across them. For the systems we are considering, this will be achieved by a random placement of resources together with the use of replicas of heavily used resources.

Optimization for local interactions between neighbouring peers: The 'network distance' between nodes that interact has a substantial impact on the latency of individual interactions, such as client requests for access to resources. Network traffic loadings are also impacted by it. The middleware should aim to place resources close to the nodes that access them the most.

Accommodating to highly dynamic host availability: Most peer-to-peer systems are constructed from host computers that are free to join or leave the system at any time. The hosts and network segments used in peer-to-peer systems are not owned or managed by any single authority; neither their reliability nor their continuous participation in the provision of a service is guaranteed. A major challenge for peer to peer systems is to provide a dependable service despite these facts. As hosts join the system, they must be integrated into the system and the load must be redistributed to exploit

their resources. When they leave the system whether voluntarily or involuntarily, the system must detect their departure and redistribute their load and resources.

8. Describe about the routing overlays in peer-to-peer systems.

The development of middleware that meets the functional and non-functional requirements outlined in the previous section is an active area of research, and several significant middleware systems have already emerged. In this chapter we describe several of them in detail.

In peer-to-peer systems a distributed algorithm known as a *routing overlay* takes responsibility for locating nodes and objects. The name denotes the fact that the middleware takes the form of a layer that is responsible for routing requests from any client to a host that holds the object to which the request is addressed. The objects of interest may be placed at and subsequently relocated to any node in the network without client involvement. It is termed an overlay since it implements a routing mechanism in the application layer that is quite separate from any other routing mechanisms deployed at the network level such as IP routing. This approach to the management and location of replicated objects was first analyzed and shown to be effective for networks involving sufficiently many nodes in a groundbreaking paper by Plaxton *et al.* [1997].

The routing overlay ensures that any node can access any object by routing each request through a sequence of nodes, exploiting knowledge at each of them to locate the destination object. Peer-to-peer systems usually store multiple replicas of objects to ensure availability. In that case, the routing overlay maintains knowledge of the location of all the available replicas and delivers requests to the nearest 'live' node (i.e. one that has not failed) that has a copy of the relevant object.
The GUIDs used to identify nodes and objects are an example of the 'pure' names. These are also known as opaque identifiers, since they reveal nothing about the locations of the objects to which they refer.

The main task of a routing overlay is the following:
Routing of requests to objects: A client wishing to invoke an operation on an object submits a request including the object's GUID to the routing overlay, which routes the request to a node at which a replica of the object resides.
But the routing overlay must also perform some other tasks:
Insertion of objects: A node wishing to make a new object available to a peer-to-peer service computes a GUID for the object and announces it to the routing overlay, which then ensures that the object is reachable by all other clients.
Deletion of objects: When clients request the removal of objects from the service the routing overlay must make them unavailable.
Node addition and removal: Nodes (i.e., computers) may join and leave the service. When a node joins the service, the routing overlay arranges for it to assume some of the responsibilities of other nodes. When a node leaves (either voluntarily or as a result of a system or network fault), its responsibilities are distributed amongst the other nodes.

An object's GUID is computed from all or part of the state of the object using a function that delivers a value that is, with very high probability, unique. Uniqueness is verified by searching for another object with the same GUID. A hash function (such as SHA-1) is used to generate the GUID from the object's value. Because these randomly distributed identifiers are used to determine the placement of objects and to retrieve them, overlay routing systems are sometimes described as *distributed hash tables* (DHT). This is reflected by the simplest form of API used to access them, as shown in Figure below. With this API, the *put()* operation is used to submit a data item to be stored together with its GUID. The DHT layer takes responsibility for choosing a location for it, storing it with replicas to ensure availability) and providing access to it via the *get()* operation.

Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry

put(GUID, data)
Stores *data* in replicas at all nodes responsible for the object identified by *GUID*.

remove(GUID)
Deletes all references to *GUID* and the associated data.

value = get(GUID)
Retrieves the data associated with *GUID* from one of the nodes responsible for it.

A slightly more flexible form of API is provided by a *distributed object location and routing* (DOLR) layer, as shown in Figure below. With this interface objects can be stored anywhere and the DOLR layer is responsible for maintaining a mapping between object identifiers (GUIDs) and the addresses of the nodes at which replicas of the objects are located. Objects may be replicated and stored with the same GUID at different hosts, and the routing overlay takes responsibility for routing requests to the nearest available replica.

Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry

publish(GUID)
GUID can be computed from the object (or some part of it, e.g., its name). This function makes the node performing a *publish* operation the host for the object corresponding to *GUID*.

unpublish(GUID)
Makes the object corresponding to *GUID* inaccessible.

sendToObj(msg, GUID, [n])
Following the object-oriented paradigm, an invocation message is sent to an object in order to access it. This might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object's state. The final optional parameter *[n]*, if present, requests the delivery of the same message to *n* replicas of the object.

With the DHT model, a data item with GUID *X* is stored at the node whose GUID is numerically closest to *X* and at the *r* hosts whose GUIDs are next-closest to it numerically, where *r* is a replication factor chosen to ensure a very high probability of availability. With the DOLR model, locations for the replicas of data objects are decided outside the routing layer and the DOLR layer is notified of the host address of each replica using the *publish()* operation.

9. Explain in detail about the importance of DNS.

The Domain Name System is a name service design whose main naming database is used across the Internet. It was devised principally by Mockapetris and specified in RFC 1034 [Mockapetris 1987] and RFC 1035. DNS replaced the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them [Harrenstien *et al.* 1985].

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.

The objects named by the DNS are primarily computers – for which mainly IP addresses are stored as attributes – and what we have referred to in this chapter as naming domains are called simply *domains* in the DNS. In principle, however, any type of object can be named, and its architecture gives scope for a variety of implementations. Organizations and departments within them can manage their own naming data. Millions of names are bound by the Internet DNS, and lookups are made against it from around the world. Any name can be resolved by any client. This is achieved by hierarchical partitioning of the name database, by replication of the naming data, and by caching.

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:

com – Commercial organizations
edu – Universities and other educational institutions
gov – US governmental agencies
mil – US military organizations
net – Major network support centres
org – Organizations not mentioned above
int – International organizations

In addition, every country has its own domains:

us – United States
uk – United Kingdom
fr – France

DNS queries • The Internet DNS is primarily used for simple host name resolution and for looking up electronic mail hosts, as follows:

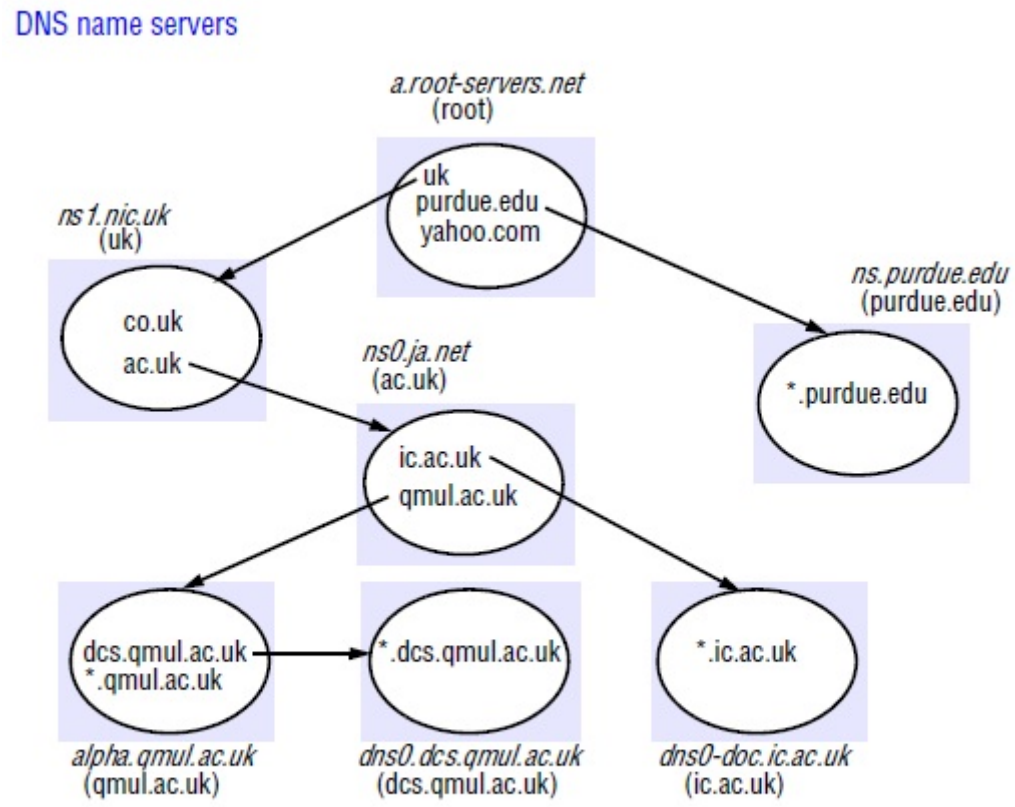
Host name resolution: In general, applications use the DNS to resolve host names into IP addresses. For example, when a web browser is given a URL containing the domain name *www.dcs.qmul.ac.uk*, it makes a DNS enquiry and obtains the corresponding IP address. As was pointed out in Chapter 4, browsers then use HTTP to communicate with the web server at the given IP address, using a reserved port number if none is specified in the URL. FTP and SMTP services work in a similar way; for example, an FTP program may be given the domain name *ftp.dcs.qmul.ac.uk*and can make a DNS enquiry to get its IP address and then use TCP to communicate with it at the reserved port number.

Mail host location: Electronic mail software uses the DNS to resolve domain names into the IP addresses of mail hosts – i.e., computers that will accept mail for those domains. For example, when the address *tom@dcs.rnx.ac.uk* is to be resolved, the DNS is queried with the address *dcs.rnx.ac.uk* and the type designation 'mail'. It returns a list of domain names of hosts that can accept mail for *dcs.rnx.ac.uk*, if such exist (and, optionally, the corresponding IP addresses). The DNS may return more than one domain name so that the mail software can try alternatives if the main mail host is unreachable for some reason. The DNS returns an integer preference value for each mail host, indicating the order in which the mail hosts should be tried.

DNS name servers • The problems of scale are treated by a combination of partitioning the naming database and replicating and caching parts of it close to the points of need. The DNS database is distributed across a logical network of servers. Each server holds part of the naming database – primarily data for the local domain. Queries concerning computers in the local domain are satisfied by servers within that domain. However, each server records the domain names and addresses of other name servers, so that queries pertaining to objects outside the domain can be satisfied.

Figure below shows the arrangement of some of the DNS database as it stood in the year 2001. This example is equally valid today even if some of the data has altered as systems have been reconfigured over time. Note that, in practice, root servers such as *a.root-servers.net* hold entries for several levels of domain, as well as entries for first level domain names. This is to reduce the number

of navigation steps required to resolve domain names. Root name servers hold authoritative entries for the name servers for the top-level domains. They are also authoritative name servers for the generic top-level domains, such as *com* and *edu*. However, the root name servers are not name servers for the country domains. For example, the *uk* domain has a collection of name servers, one of which is called *ns1.nic.net*. These name servers know the name servers for the second-level domains in the United Kingdom such as *ac.uk* and *co.uk*. The name servers for the domain *ac.uk* know the name servers for all of the university domains in the country, such as *qmul.ac.uk* or *ic.ac.uk*. In some cases, a university domain delegates some of its responsibilities to a subdomain, such as *dcs.qmul.ac.uk*.



Name server names are in italics, and the corresponding domains are in parentheses. Arrows denote name server entries

10. Give a detailed description of PASTRY and TAPESTRY approach.

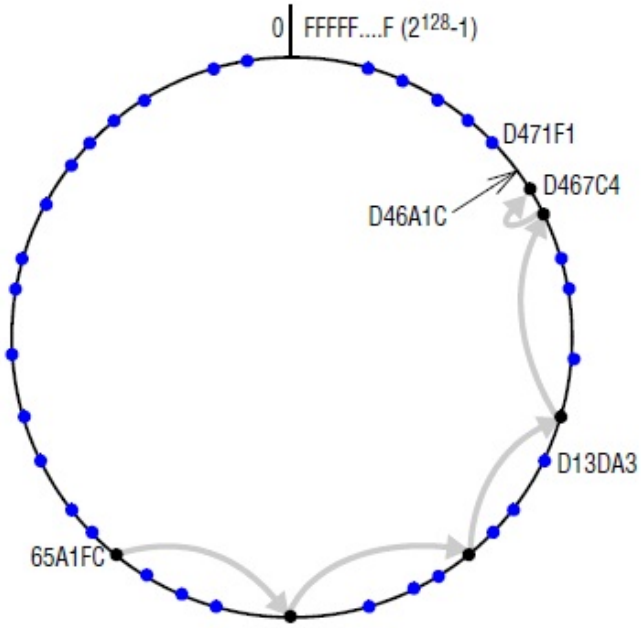
Pastry

Pastry is a routing overlay. All the nodes and objects that can be accessed through Pastry are assigned 128-bit GUIDs. For nodes, these are computed by applying a secure hash function (such as SHA-1) to the public key with which each node is provided. For objects such as files, the GUID is computed by applying a secure hash function to the object's name or to some part of the object's stored state. The resulting GUIDs have the usual properties of secure hash values – that is, they are randomly distributed in the range 0 to 2128–1. They provide no clues as to the value from which they were computed, and clashes between GUIDs for different nodes or objects are extremely unlikely. (If a clash occurs, Pastry detects it and takes remedial action.)

In a network with *N* participating nodes, the Pastry routing algorithm will correctly route a message addressed to any GUID in *O(log N)* steps. If the GUID identifies a node that is currently active, the message is delivered to that node; otherwise, the message is delivered to the active node whose GUID is numerically closest to it. Active nodes take responsibility for processing requests addressed to all objects in their numerical neighbourhood.

The GUID space is treated as circular: GUID 0's lower neighbour is 2^{128}–1. Figure below gives a view of active nodes distributed in this circular address space. Since every leaf set includes the GUIDs and IP addresses of the current node's immediate neighbours, a Pastry system with correct leaf sets of size at least 2 can route messages to any GUID trivially as follows: any node *A* that receives a message *M* with destination address *D* routes the message by comparing *D* with its own GUID *A* and with each of the GUIDs in its leaf set and forwarding *M* to the node amongst them that is numerically closest to *D*.

Circular routing alone is correct but inefficient *Based on Rowstron and Druschel [2001]*



The dots depict live nodes. The space is considered as circular: node 0 is adjacent to node (2^{128}-1). The diagram illustrates the routing of a message from node 65A1FC to D46A1C using leaf set information alone, assuming leaf sets of size 8 (/= 4). This is a degenerate type of routing that would scale very poorly; it is not used in practice.

The Figure below shows the structure of the routing table for a specific node,

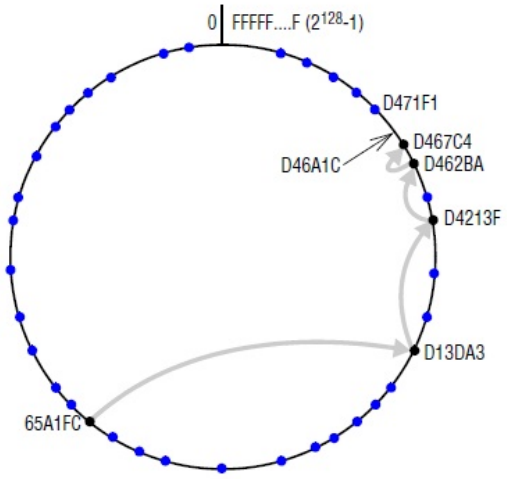
First four rows of a Pastry routing table

p =	GUID prefixes and corresponding node handles n															
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	n	n	n	n	n	n		n	n	n	n	n	n	n	n	n
1	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6F	6E	6F
	n	n	n	n	n		n	n	n	n	n	n	n	n	n	n
2	650	651	652	653	654	655	656	657	658	659	65A	65B	65C	65D	65E	65F
	n	n	n	n	n	n	n	n	n	n		n	n	n	n	n
3	65A0	65A1	65A2	65A3	65A4	65A5	65A6	65A7	65A8	65A9	65AA	65AB	65AC	65AD	65AE	65AF
	n		n	n	n	n	n	n	n	n	n	n	n	n	n	n

The routing table is located at a node whose GUID begins 65A1. Digits are in hexadecimal. The *n*s represent [GUID, IP address] pairs that act as node handles specifying the next hop to be taken by messages addressed to GUIDs that match each given prefix. Grey-shaded entries in the table body indicate that the prefix matches the current GUID up to the given value of *p*: the next row down or the leaf set should be examined to find a route. Although there are a maximum of 128 rows in the table, only $\log_{16} N$ rows will be populated on average in a network with *N* active nodes.

Figure below illustrates the actions of the routing algorithm. The routing table is structured as follows: GUIDs are viewed as hexadecimal values and the table classifies GUIDs based on their hexadecimal prefixes. The table has as many rows as there are hexadecimal digits in a GUID, so for the prototype Pastry system that we are describing, there are 128/4 = 32 rows. Any row *n* contains 15 entries – one for each possible value of the *n*th hexadecimal digit, excluding the value in the local node's GUID. Each entry in the table points to one of the potentially many nodes
whose GUIDs have the relevant prefix.

Pastry routing example *Based on Rowstron and Druschel [2001]*



Routing a message from node 65A1FC to D46A1C. With the aid of a well-populated routing table the message can be delivered in ~ $\log_{16}(N)$ hops.

The routing process at any node *A* uses the information in its routing table *R* and leaf set *L* to handle each request from an application and each incoming message from another node according to the algorithm shown in Figure below.

Pastry's routing algorithm

To handle a message M addressed to a node D (where $R[p,i]$ is the element at column i, row p of the routing table):

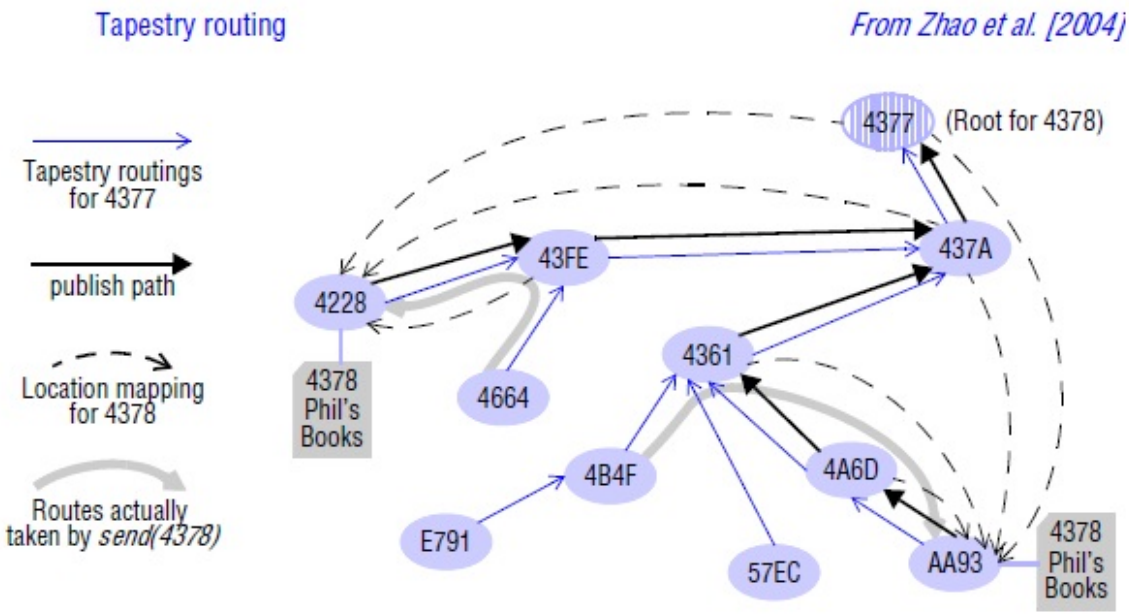
1. *If* ($L_{-l} < D < L_l$) { // the destination is within the leaf set or is the current node.
2. Forward M to the element L_i of the leaf set with GUID closest to D or the current node A.
3. } *else* { // use the routing table to despatch M to a node with a closer GUID
4. Find p, the length of the longest common prefix of D and A,. and i, the $(p+1)^{th}$ hexadecimal digit of D.
5. *If* ($R[p,i] \neq null$) forward M to $R[p,i]$ // route M to a node with a longer common prefix.
6. else { // there is no entry in the routing table.
7. Forward M to any node in L or R with a common prefix of length p but a GUID that is numerically closer.
   }
   }

Tapestry

Tapestry implements a distributed hash table and routes messages to nodes based on GUIDs associated with resources using prefix routing in a manner similar to Pastry. But Tapestry's API conceals the distributed hash table from applications behind a DOLR interface. Nodes that hold resources use the *publish(GUID)* primitive to make them known to Tapestry, and the holders of resources remain responsible for storing them. Replicated resources are published with the same GUID by each node that holds a replica, resulting in multiple entries in the Tapestry routing structure.

In Tapestry 160-bit identifiers are used to refer both to objects and to the nodes that perform routing actions. Identifiers are either *NodeIds*, which refer to computers that perform routing operations, or *GUIDs*, which refer to the objects. For any resource with GUID G there is a unique root node with GUID RG that is numerically closest to$G$. Hosts H holding replicas of G periodically invoke *publish*(G) to ensure that newly arrived hosts become aware of the existence of G.

On each invocation of *publish*(G) a publish message is routed from the invoker towards node RG. On receipt of a publish message RG enters (G, *IPH*), the mapping between G and the sending host's IP addressin its routing table, and each node along the publication path caches the same mapping. This process is illustrated in Figure 10.10. When nodes hold multiple (G, *IP*) mappings for the same GUID, they are sorted by the network distance (round-trip time) to the IP address. For replicated objects this results in the selection of the nearest available replica of the object as the destination for subsequent messages sent to the object.



Replicas of the file *Phil's Books* (G=4378) are hosted at nodes 4228 and AA93. Node 4377 is the root node for object 4378. The Tapestry routings shown are some of the entries in routing tables. The publish paths show routes followed by the publish messages laying down cached location mappings for object 4378. The location mappings are subsequently used to route messages sent to 4378.

UNIT IV
PART A

1. **What is clock synchronization?**
   Nodes in distributed system to keep track of current time for various purposes such as calculating the time spent by a process in CPU utilization ,disk I/O etc so that the corresponding user can be charged. Clock synchronization means the time difference between two nodes should be very small.
2. **What do you mean by clock skew and clock drift?**
   - **Clock skew** – Instantaneous difference between the readings of any two clocks is called clock skew. Skew occurs since computer clocks like any others tends not be perfect at all times.
   - **Clock drift** – Clock drift occurs in crystal based clocks which counts time at different rates and hence they diverge. The drift rate is the change in the offset between the clock and a nominal perfect reference clock per unit of time measured by the reference clock.
3. **What do you mean by Coordinated Universal Time?**
   Coordinated Universal Time generally abbreviated as UTC is an international standard for timekeeping. It is based on atomic time. UTC signals are synchronized and broadcast regularly from land based radio stations and satellites covering many parts of the world.
4. **Define External Synchronization.**
   Generally it is necessary to synchronize the processes' clocks C_i with an authoritative external source of time. It is called as External Synchronization. For a synchronization bound D>0, and for a source S of UTC time, $| S(t) - C_i(t)| < D$ for i=1,2..N for all real times t in I where I is the time interval.
5. **When an object is considered to be garbage?**
   An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system. The memory taken up by the object can be reclaimed once it is known to be garbage. The technique used here is distributed garbage collection.
6. **What do you meant by Distributed debugging?**

In general, distributed systems are complex to debug.A special care needs to be taken in establishing what occurred during the execution. Consider an application with a variable x_i(i=1,2..N) and the variable changes as the program executes but it is always required to be within a value $ of one other. In that case, relationship must be evaluated for values of the variables that occur at the same time.

7. Define marker receiving rule.

Snapshot algorithm designed by Chandy and Lamport is used for determining global states of distributed systems. This algorithm is defined through two rules namely marker sending rule and marker receiving rule. Marker receiving rule obligates a process that has not recorded its state to do so.

8. Define marker sending rule.

Snapshot algorithm designed by Chandy and Lamport is used for determining global states of distributed systems. This algorithm is defined through 2 rules namely marker sending rule and marker receiving rule. Marker sending rule obligates processes to send a marker after they have recorded their state ,but before they send any other messages.

9. Define total ordering?

Common ordering requirements are important in case of multicast approach. General ordering requirements are total ordering, FIFO ordering and causal ordering. Total ordering is the case where if a correct process delivers message m before it delivers m', then any other correct process that delivers m' will deliver m before m'.

10. Name any two election algorithms.

An algorithm for choosing a unique process to play a particular role is called an election algorithm.Generally used election algorithms are:

- Ring based election algorithm
- Bully algorithm

11. What do you mean by atomic transaction?

A transaction is said be an atomic transaction if it follows All or Nothing property according to which a transaction either completes successfully in which case the effects of all of its operations are recorded in the objects or has no effect at all. In short atomic transaction is a transaction that happens completely or not at all. It does not produce partial results.

For eg: ATM machine hands you cash and deducts amount from your account or does not have any effect at all.

12. What are the ACID properties of a transaction?

- **Atomicity** : A transaction must be all or nothing.
- **Consistency** : A transaction takes the system from one consistent state to another consistent state.
- **Isolation**: The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other.
- **Durability:**Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

13. **Define the characteristics of serial equivalent transactions**.

For any pair of transactions, it is possible to determine the order of pairs of conflicting operations on objects accessed by both of them. Read and write are the operations generally considered. For two transactions to be serially equivalent it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

14. What are the advantages of nested transactions?

The outermost transaction in a set of nested transactions is called top level transaction. Transactions other than the top level transaction are called subtransactions.

Advantages of nested transactions are:

- Subtransactions at one level may run concurrently with other subtransactions at the same level in the hierarchy. This can allow additional concurrency in a transaction.
- Subtransactions can commit or abort independently.

15. What are the rules of committing nested transactions?

Rules for committing of nested transactions are:

- A transaction may commit or abort only after its child transactions have completed.
- When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort.
- When a parent aborts, all of its transactions are aborted.
- When a subtransaction aborts, the parent can decide whether to abort or not.

16. Write short notes on strict two phase locking

A simple mechanism of a serializing mechanism is the use of exclusive locks. Under a strict execution regime, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted. To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. This is called *strict two-phase locking*. The presence of the locks prevents other transactions reading or writing the objects.

17. What are the drawbacks of locking?

Drawbacks of locking mechanism are:

- Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data.
- The use of locks can result in deadlock in some cases.
- To avoid cascading aborts, locks cannot be released until the end of the transaction. It significantly reduces the potential for accuracy.

18. Define the approach of two phase commit protocol.

Two phase commit protocol is designed to allow any participant to abort its part of a transaction. In the first phase of the protocol, each participant votes for the transaction to be committed or aborted. In the second phase of the protocol, every participant in the transaction carries out the joint decision.

19. How is distributed dead lock detected?

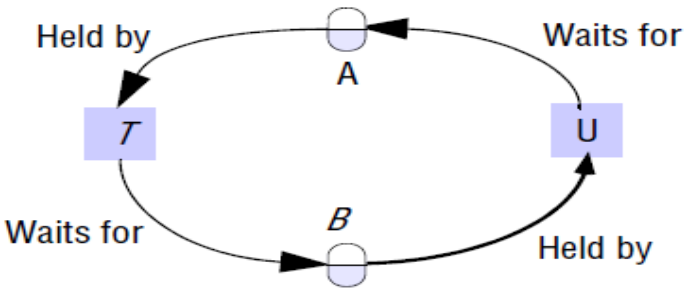
In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for-graph has to be constructed. If and only if there is a cycle in the wait-for-graph a distributed deadlock is said to be detected. Hence Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions.

20. What is a phantom deadlock?

A deadlock that is 'detected' but is not really a deadlock is called a phantom deadlock. In distributed system if there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. As this procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist. This is a sample case for phantom deadlocks.

21. What is wait-for-graph?

A wait-for graph can be used to represent the waiting relationships between current transactions. In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions. Following figure represents a wait-for-graph with transactions T and U.



22. Define Edge chasing

A distributed approach to deadlock detection uses a technique called edge chasing or path pushing. In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by

forwarding messages called probes, which follow the edges of the graph throughout the distributed system

23. What is the role of replication in distributed systems?

Replication is defined as the maintenance of copies of data at multiple computers. It is a key to the effectiveness of distributed systems in that it can provide enhanced performance, high availability and fault tolerance.

PART – B

1. Write about internal and external synchronization of physical clock.

In order to know at what time of day events occur at the processes in our distributed system for example, for accountancy purposes , it is necessary to synchronize theprocesses' clocks, *Ci* , with an authoritative, external source of time. This is ***externalsynchronization*.** And if the clocks *Ci* are synchronized with one another to a knowndegree of accuracy, then we can measure the interval between two events occurring atdifferent computers by appealing to their local clocks, even though they are notnecessarily synchronized to an external source of time. This is ***internal synchronization*.**

We define these two modes of synchronization more closely as follows, over an intervalof real time *I*:

***External synchronization*:** For a synchronization bound $D > 0$, and for a source *S* ofUTC time, $|S(t) - Ci(t)| < D$, for $i = 1,2\ldots N$ and for all real times *t* in *I*. Anotherway of saying this is that the clocks *Ci* are *accurate* to within the bound *D*.

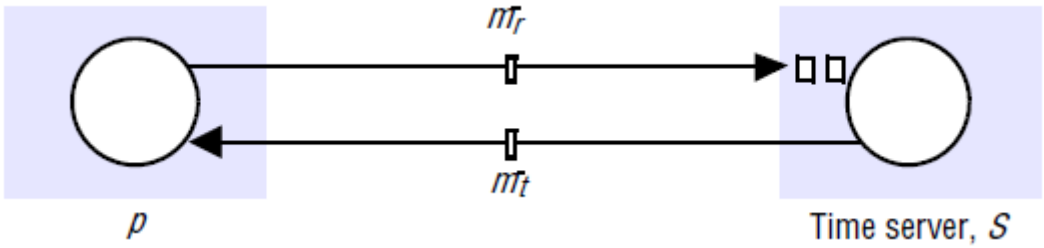
Internal synchronization: For a synchronization bound $D > 0$, $|Ci(t) - Cj(t)| < D$ for $i, j = 1,2\ldots N$, and for all real times *t* in *I*. Another way of saying this is thatthe clocks *Ci agree* within the bound *D*.

Clocks that are internally synchronized are not necessarily externally synchronized,since they may drift collectively from an external source of time even though they agreewith one another.A clock that does not keep to whatever correctness conditions apply is defined tobe *faulty*. A clock's *crash failure* is said to occur when the clock stops ticking altogether, any other clock failure is an *arbitrary failure*.

Cristian's method for synchronizing clocks

Cristian [1989] suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally.

Upon request, the server process *S* supplies the time according to its clock.



Cristian describes the algorithm as *probabilistic*: the method achieves synchronization only if the observed round-trip times between client and server are sufficiently short compared with the required accuracy. Cristian's method suffers from theproblem associated with all services implemented by a single server: that the single time server might fail and thus render synchronization temporarily impossible.

Berkeley algorithm

Gusella and Zatti [1989] describe an algorithm for internal synchronization that they developed for collections of computers running Berkeley UNIX. In it, a coordinator computer is chosen to act as the *master*. Unlike in Cristian's protocol, this computer periodically polls the other computers whose clocks are to be synchronized, called *slaves*. The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times and it averages the values obtained . The Berkeley algorithm eliminates readings from faulty

clocks. Such clocks could have a significant adverse effect if an ordinary average was taken so instead the master takes a *fault-tolerant average*.

2. Explain about Network Time Protocol.

The Network Time Protocol (NTP) [Mills 1995] defines architecture for a time service and a protocol to distribute time information over the Internet.
NTP's chief design aims and features are as follows:

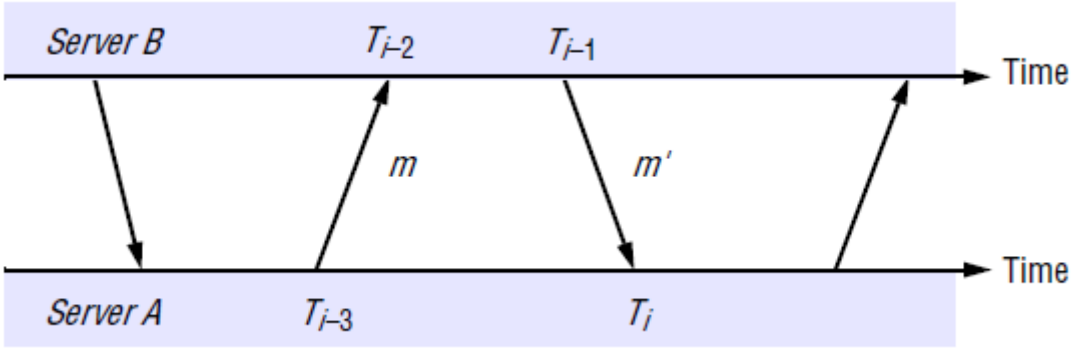
***To provide a service enabling clients across the Internet to be synchronizedaccurately to UTC*:** Although large and variable message delays are encountered inInternet communication, NTP employs statistical techniques for the filtering oftiming data and it discriminates between the quality of timing data from differentservers.

***To provide a reliable service that can survive lengthy losses of connectivity*:** There are redundant servers and redundant paths between the servers. The servers can reconfigure so as to continue to provide the service if one of them becomes unreachable.

***To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers*:** The service is designed to scale to large numbers of clientsand servers.

***To provide protection against interference with the time service, whether malicious or accidental*:** The time service uses authentication techniques to check that timingdata originate from the claimed trusted sources. It also validates the return addressesof messages sent to it.

The NTP service is provided by a network of servers located across the Internet. *Primary servers* are connected directly to a time source such as a radio clock receiving UTC;*secondary servers* are synchronized, ultimately, with primary servers. The servers areconnected in a logical hierarchy called a *synchronization subnet* whose levels are called *strata*.



NTP servers synchronize with one another in one of three modes: multicast, procedure-call and symmetric mode.
Multicast modeis intended for use on a high-speed LAN. One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay. This mode can achieve only relatively low accuracies, but ones that nonetheless are considered sufficient for many purposes.
Procedure-call modeis similar to the operation of Cristian's algorithm. In this mode, one server accepts requests from other computers, which it processes by replying with its timestamp . This mode is suitable where higher accuracies are required than can be achieved with multicast, or where multicast is not supported in hardware. For example, file servers on the same or a neighbouring LAN that need to keep accurate timing information for file accesses could contact a local server in procedure-call mode.
Finally, ***symmetric mode***is intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet, where the highest accuracies are to be achieved. A pair of servers operating in symmetric mode exchange messages bearing timing

information. Timing data are retained as part of an association between the servers that is maintained in order to improve the accuracy of their synchronization over time.

3. Discuss in detail about the Chandy and Lamports snapshot algorithm for determining the global states of distributed systems.

Chandy and Lamport [1985] describe a 'snapshot' algorithm for determining global states of distributed systems, which we now present. The goal of the algorithm is to record a set of process and channel states (a 'snapshot') for a set of processes *pi*($i = 1, 2 \ldots N$) such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.

The algorithm assumes that:

- Neither channels nor processes fail – communication is reliable so that everymessage sent is eventually received intact, exactly once.
- Channels are unidirectional and provide FIFO-ordered message delivery.
- The graph of processes and channels is strongly connected.
- Any process may initiate a global snapshot at any time.
- The processes may continue their execution and send and receive normal messages while the snapshot takes place.

Marker receiving rule for process p_i

On receipt of a *marker* message at p_i over channel c:

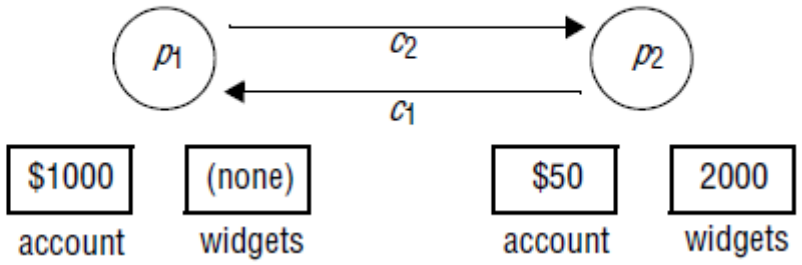
if (p_i has not yet recorded its state) it
records its process state now;
records the state of c as the empty set;
turns on recording of messages arriving over other incoming channels;
else
p_i records the state of c as the set of messages it has received over c since it saved its state.
end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c:
p_i sends one marker message over c
(before it sends any other message over c).

The marker sending rule obligates processes to send a marker after they have recorded their state, but before they send any other messages. The marker receiving rule obligates a process that has not recorded its state to do so.

Two processes and their initial states



In this snapshot algorithm, We assume that a process that has received amarker message records its state within a finite time and sends marker messages overeach outgoing channel within a finite time which is not to be correct at all time.

4. Explain briefly about election algorithms.

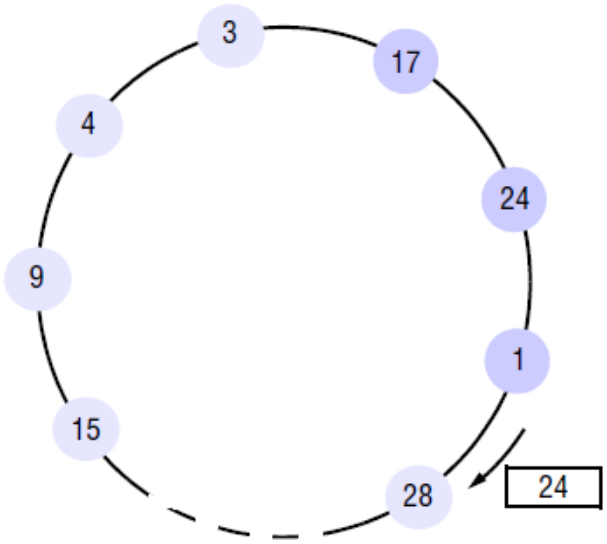
An algorithm for choosing a unique process to play a particular role is called an *electionalgorithm*. For example, in a variant of our central-server algorithm for mutualexclusion, the 'server' is chosen from among the processes *pi*(i = 1,2…N) thatneed to use the critical section. An election algorithm is needed for this choice. It isessential that all the processes agree on the choice. Afterwards, if the process that playsthe role of server wishes to retire then another election is required to choose areplacement.

A process *calls the election* if it takes an action that initiates particular run of the election algorithm. An individual process does not call more than one election at a time, but in principle the N processes could call N concurrent elections. At any point in time, a process *pi* is either a *participant* – meaning that it is engaged in some run of the election algorithm – or a *non-participant* – meaning that it is notcurrently engaged in any election. The performance of an election algorithm is measured by its total networkbandwidth utilization andby the *turnaround time* for the algorithm: the number of serialized message transmissiontimes between the initiation and termination of a single run.

Ring-based election algorithm

The algorithm of Chang and Roberts [1979] is suitable for a collection of processes arranged in a logical ring. Each process *pi* has acommunication channel to the next process in the ring, $p(i + 1) \bmod N$, and all messages are sent clockwise around the ring. We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect a single process called the *coordinator*, which is the process with the largest identifier.

A ring-based election in progress



Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a darker tint.

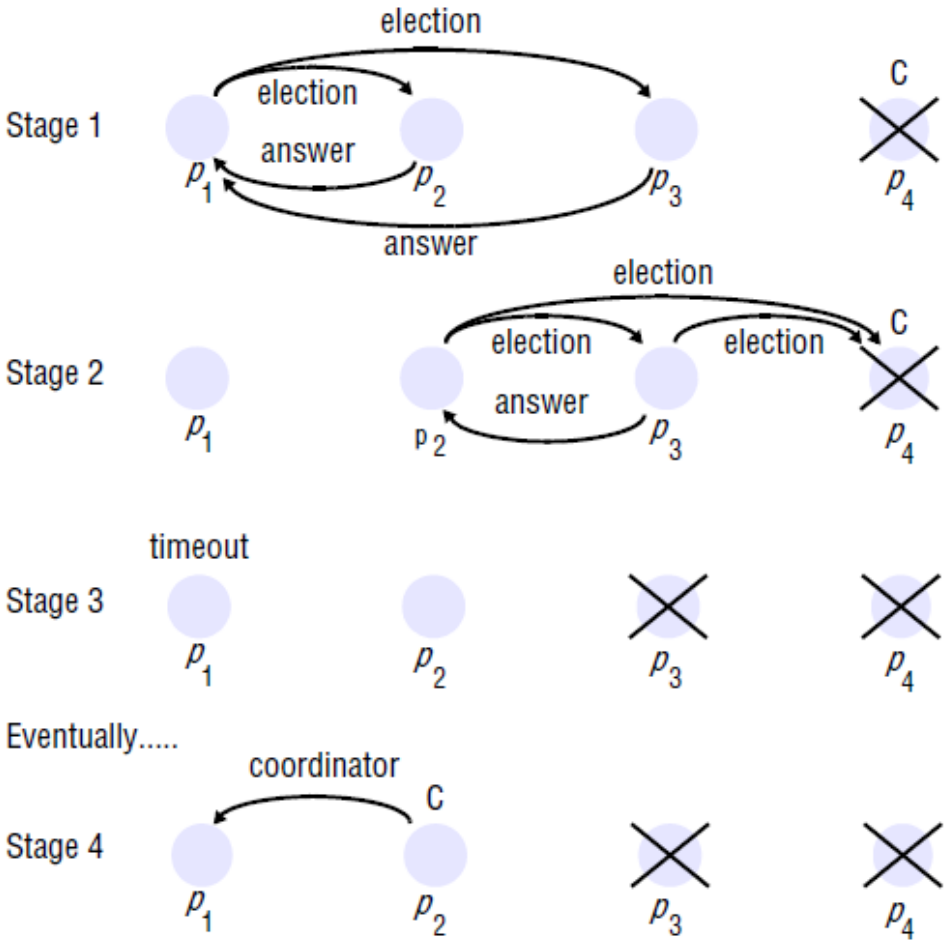
If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier. A total of $N - 1$ messages are then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages. The *elected* message is then sent N times, making $3N - 1$ messages in all. The turnaround time is also $3N - 1$, since these messages are sent sequentially.

Bully Algorithm

The bully algorithm [Garcia-Molina 1982] allows processes to crash during an election, although it assumes that message delivery between processes is reliable. Unlike the ring-based algorithm, this algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure. Another difference is that the ring-based algorithm assumed that processes have minimal *a priori* knowledge of one another: each knows only how to communicate with its neighbour, and none knows the identifiers of the other processes. The bully algorithm, on the other hand, assumes thateach process knows which processes have higher identifiers, and that it can communicate with all such processes.

There are three types of message in this algorithm: an *election* message is sent to announce an election; an *answer* message is sent in response to an election message and a *coordinator* message is sent to announce the identity of the elected process – the new 'coordinator'. A process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently.

The bully algorithm



The election of coordinator p_2, after the failure of p_4 and then p_3

5. Write about Distributed Mutual Exclusion.

Distributed processes often need to coordinate their activities. If a collection ofprocesses share a resource or collection of resources, then often mutual exclusion isrequired to prevent interference and ensure consistency when accessing the resources.This is the *critical section* problem, familiar in the domain of operating systems. In adistributed system, however, neither

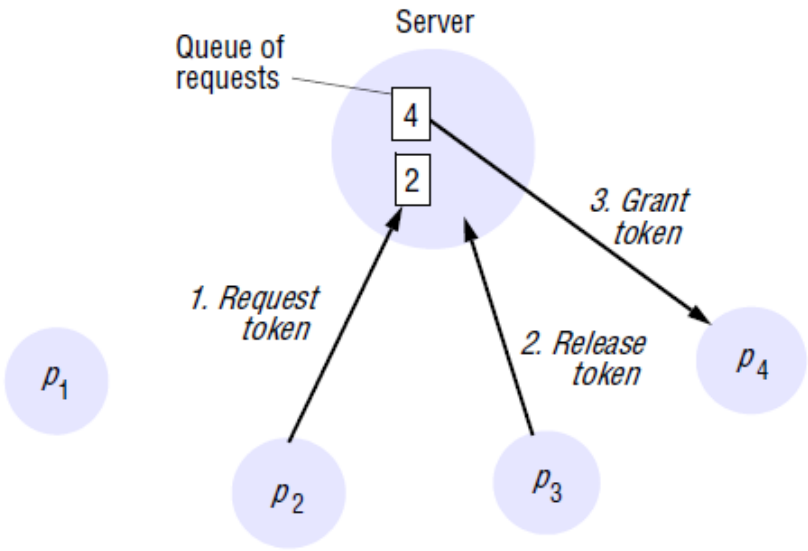
shared variables nor facilities supplied by a singlelocal kernel can be used to solve it, in general. Consider users who update a text file. A simple means of ensuring that theirupdates are consistent is to allow them to access it only one at a time, by requiring theeditor to lock the file before updates can be made.

Consider a system of N processes $pi, i = 1,2\ldots N$, that do not share variables.The processes access common resources, but they do so in a critical section. For the sakeof simplicity, we assume that there is only one critical section. It is straightforward toextend the algorithms we present to more than one critical section.We assume that the system is asynchronous, that processes do not fail and thatmessage delivery is reliable, so that any message sent is eventually delivered intact,exactly once.

The application-level protocol for executing a critical section is as follows:
enter() // enter critical section – block if necessary
resourceAccesses() // access shared resources in critical section
exit() // leave critical section – other processes may now enter

Our essential requirements for mutual exclusion are as follows:
ME1: (safety) At most one process may execute in the critical section(CS) at a time.
ME2: (liveness) Requests to enter and exit the critical section eventuallysucceed.
Condition ME2 implies freedom from both deadlock and starvation

Server managing a mutual exclusion token for a set of processes



ME3: ($\rightarrow$ ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

The performance of algorithms for mutual exclusion according to the following criteria:

- the *bandwidth* consumed, which is proportional to the number of messages sent in each *entry* and *exit* operation;
- the *client delay* incurred by a process at each *entry* and *exit* operation;
- the algorithm's effect upon the *throughput* of the system.

Central server algorithm

The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section.

Ring Based algorithm

One of the simplest ways to arrange mutual exclusion between the N processes without requiring an additional process is to arrange them in a logical ring. This requires only that each process pi has a communication channel to the next process in the ring, $p(i + 1) mod N$.

6. Write in detail about Optimistic Concurrency Control.

Kung and Robinson [1981] identified a number of inherent disadvantages of locking andproposed an alternative optimistic approach to the serialization of transactions thatavoids these drawbacks.

- Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data.
- The use of locks can result in deadlock.
- To avoid cascading aborts, locks cannot be released until the end of the transaction.

The alternative approach proposed by Kung and Robinson is 'optimistic' because it is based on the observation that, in most applications, the likelihood of two clients' transactions accessing the same object is low

Each transaction has the following phases:

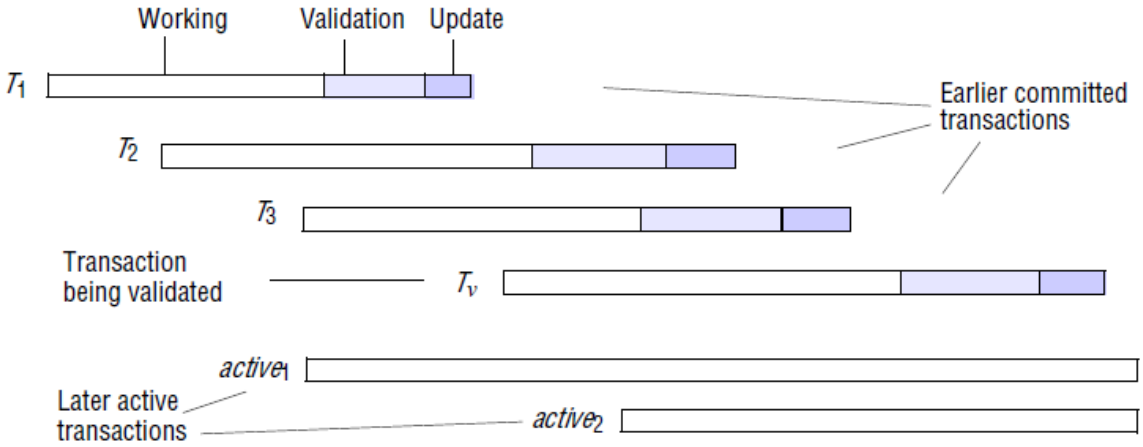
***Working phase*:** During the working phase, each transaction has a tentative version of each of the objects that it updates. This is a copy of the most recently committed version of the object.

Validation phase: When the *closeTransaction* request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects.

***Update phase*:** If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation.

The validation test on transaction *Tv* is based on conflicts between operations inpairs of transactions *Ti* and *Tv*. For a transaction *Tv* to be serializable with respect to anoverlapping transaction *Ti*,
Their operations must conform to the following rules:

T_v	T_i	*Rule*	
write	*read*	1.	T_i must not read objects written by T_v.
read	*write*	2.	T_v must not read objects written by T_i.
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i.



Backward validation checks the transaction undergoing validation with other preceding overlapping transactions , those that entered the validation phase before it.

Forward validation checks the transaction undergoing validation with other later transactions, which are still active.

7. Discuss about the concurrency control in distributed transactions.

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions.
Concurrency control protocols can be broadly divided into two categories –

Lock based protocols
Time stamp based protocols

Generally transactions are expected to follow the ACID properties whereACID stands for Atomicity, Consistency, Isolation and Durability. **ACID** (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably.

Atomicity requires that each transaction be "all or nothing".If one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

Consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.

Isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of concurrency control.

Durability property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently.

Process of managing simultaneous operations on the database without having them interfere with one another is to be accomplished by the mechanism of concurrency control.
Three examples of potential problems caused by concurrency:
• Lost update problem
• Uncommitted dependency problem
• Inconsistent analysis problem

LOST UPDATE PROBLEM

Successfully completed update is overridden by another user.
Example:
• T1 withdraws £10 from an account with bal x, initially £100.
• T2 deposits £100 into same account.
• Serially, final balance would be £190.

UNCOMMITTED DEPENDENCY PROBLEM
Occurs when one transaction can see intermediate results of another transaction before it has committed.
Example:
• T4 updates bal x to £200 but it aborts, so bal x should be back at original value of £100.
• T3 has read new value of bal x (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

INCONSISTENT ANALYSIS PROBLEM
Occurs when transaction reads several values but second transaction updates some of them during execution of first.
Example:
• T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
• Meantime, T5 has transferred £10 from bal x to bal z, so T6 now has wrong result (£10 too high).

8. Explain Time stamp ordering in detail.

In concurrency control schemes based on timestamp ordering, each operation in atransaction is validated when it is carried out. If the operation cannot be validated, thetransaction is aborted immediately and can then be restarted by the client. Eachtransaction is assigned a unique timestamp value when it starts. The timestamp definesits position in the time sequence of transactions. Requests from transactions can betotally ordered according to their timestamps.

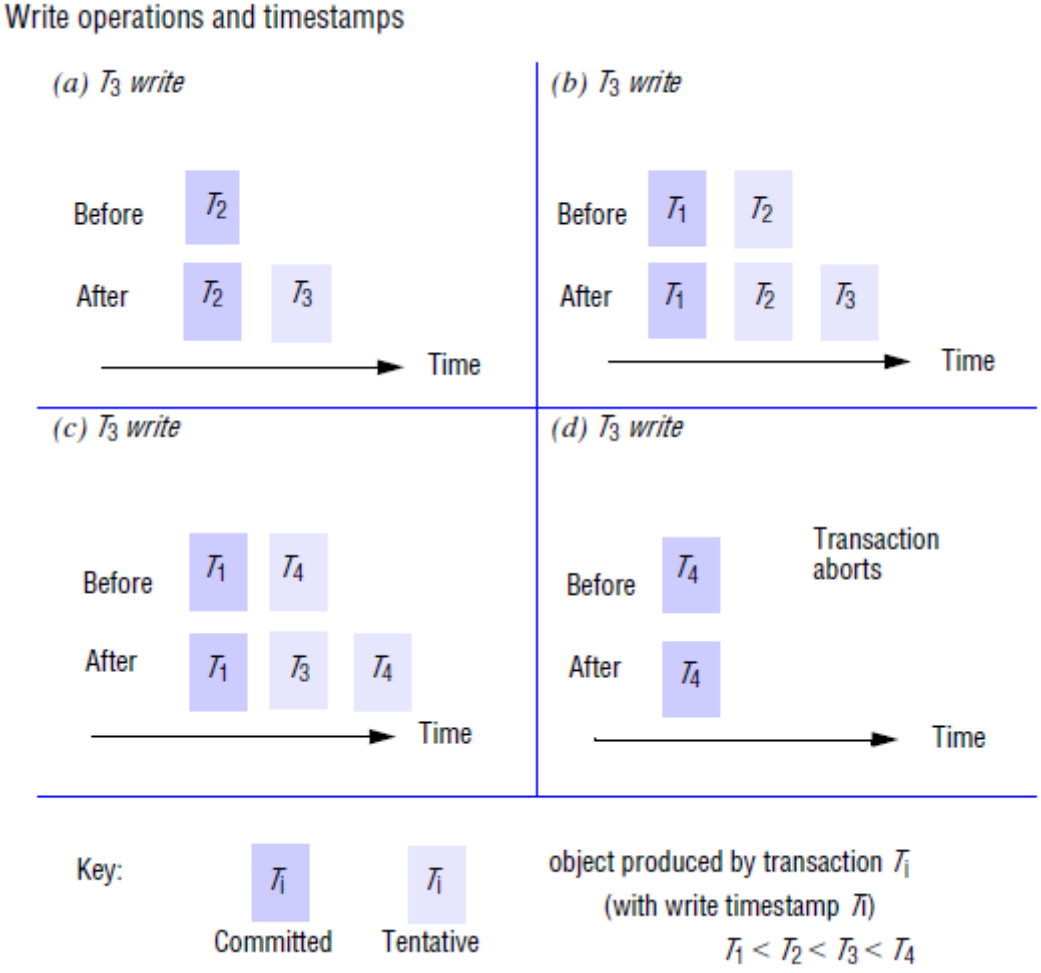
The basic timestamp ordering rule isbased on operation conflicts and is very simple:
A transaction's request to write an object is valid only if that object was last read andwritten by earlier transactions. A transaction's request to read an object is valid onlyif that object was last written by an earlier transaction.

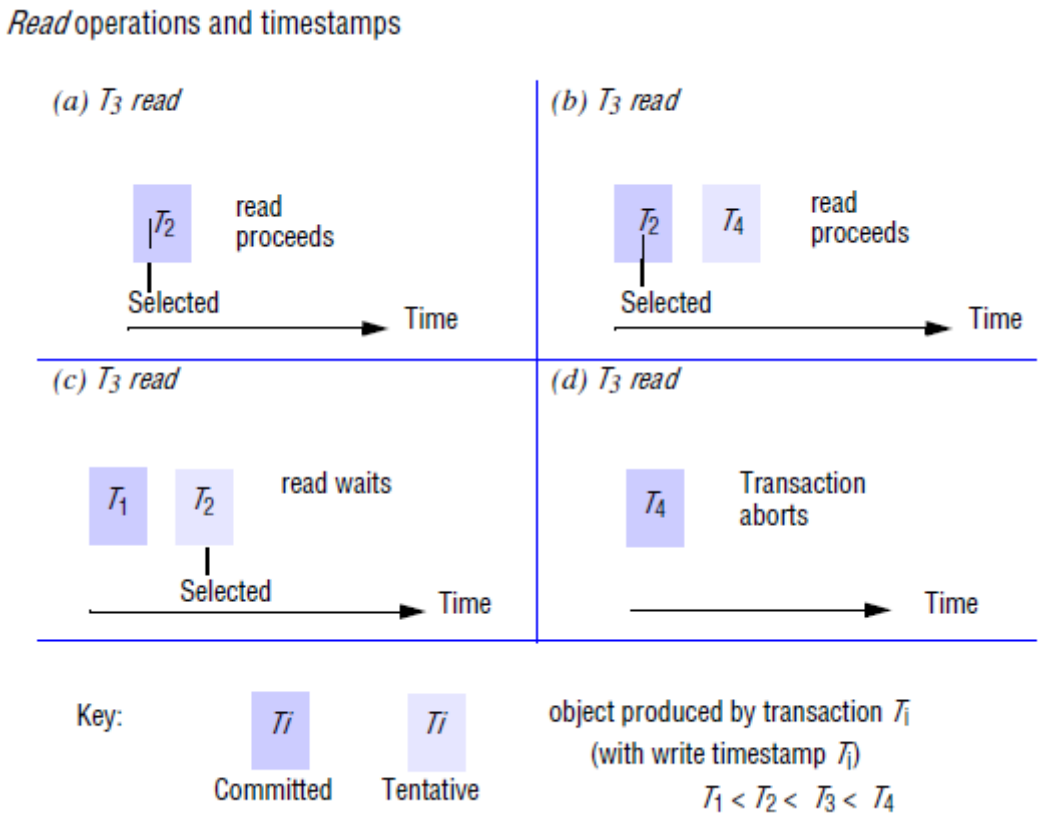
Timestamp based concurrency control is described as follows.
As usual, the *write* operations are recorded in tentative versions of objects and are invisible to other transactions until a *closeTransaction* request is issued and the transaction is committed. Every object has a write timestamp and a set of tentative versions, each of which has a write timestamp associated with it; each object also has a set of read timestamps. The write timestamp of the (committed) object is earlier than that of any of its tentative versions, and the set of read timestamps can be represented by its maximum member. Whenever a transaction's *write* operation on an object is accepted, the server creates a new tentative version of the object with its write timestamp set to the transaction timestamp. A transaction's *read* operation is directed to the version with the maximum write timestamp less than the transaction timestamp. Whenever a transaction's *read* operation on an object is accepted, the timestamp of the transaction is added to its set of read timestamps. When a transaction is committed, the values of the tentative versions become the values of the objects, and the timestamps of the tentative versions become the timestamps of the corresponding objects. In timestamp ordering, each request by a transaction for a *read* or *write* operation on an object is checked to see whether it conforms to the operation conflict rules. A request by the current transaction *Tc* can conflict with previous operations done by other transactions, *Ti*, whose timestamps indicate that they should be later than *Tc*. These rules are shown below.

Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$. This requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.

In which *Ti* >*Tc* means *Ti* is later than *Tc* and *Ti* <*Tc* means *Ti*, is earlier than *Tc*.





9. Describe in detail about two phase commit protocol.

Atomicity property of transactions requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them. The simple one-phase atomic commit protocol is inadequate, though, because itdoes not allow a server to make a unilateral decision to abort a transaction when the client requests a commit.

Two – Phase commit protocol

The *two-phase commit protocol* is designed to allow any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must be aborted**. In the first phase** of the protocol, each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a *prepared* state for a transaction if it will
eventually be able to commit it. To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its status – prepared. **In the second phase** of the protocol, every participant in the transaction carries out the joint decision. If any one participant votes to abort, then the decision must be to abort the transaction. If all the participants vote to commit, then the decision is to commit the transaction. The problem is to ensure that all of the participants vote and that they all reach the same decision.

During the progress of a transaction, there is no communication between the coordinator and the participants apart from the participants informing the coordinator when they jointhe transaction. A client's request to commit (or abort) a transaction is directed to thecoordinator. If the client requests *abortTransaction*, or if the transaction is aborted byone of the participants, the coordinator informs all participants immediately. It is whenthe client asks the coordinator to commit the transaction that the two-phase commitprotocol comes into use.

Operations for two-phase commit protocol

canCommit?(trans) *Yes / No*
Call from coordinator to participant to ask whether it can commit a transaction.Participant replies with its vote.
doCommit(trans)
Call from coordinator to participant to tell participant to commit its part of atransaction.
doAbort(trans)
Call from coordinator to participant to tell participant to abort its part of a transaction.
haveCommitted(trans, participant)
Call from participant to coordinator to confirm that it has committed the transaction.
getDecision(trans) *Yes / No*
Call from participant to coordinator to ask for the decision on a transaction when it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

The two-phase commit protocol

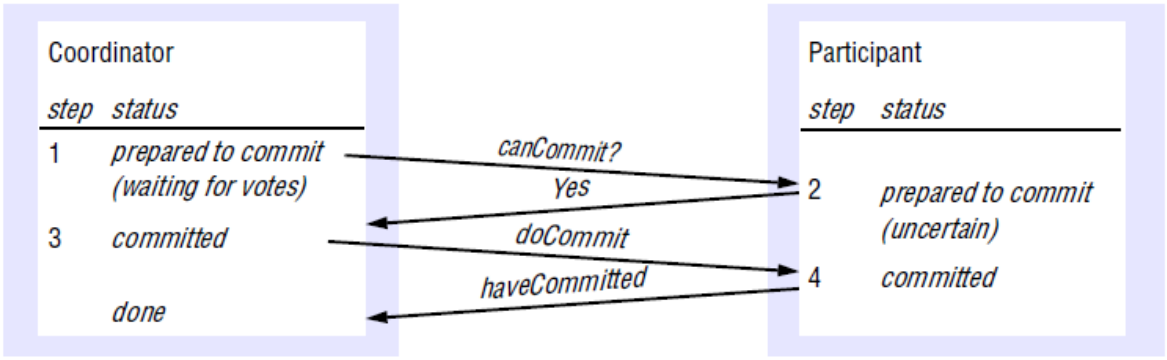
Phase 1 (voting phase):

1. The coordinator sends a *canCommit?* request to each of the participants in the transaction.
2. When a participant receives a *canCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No,* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes*, the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise, the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Communication in two- phase commit protocol



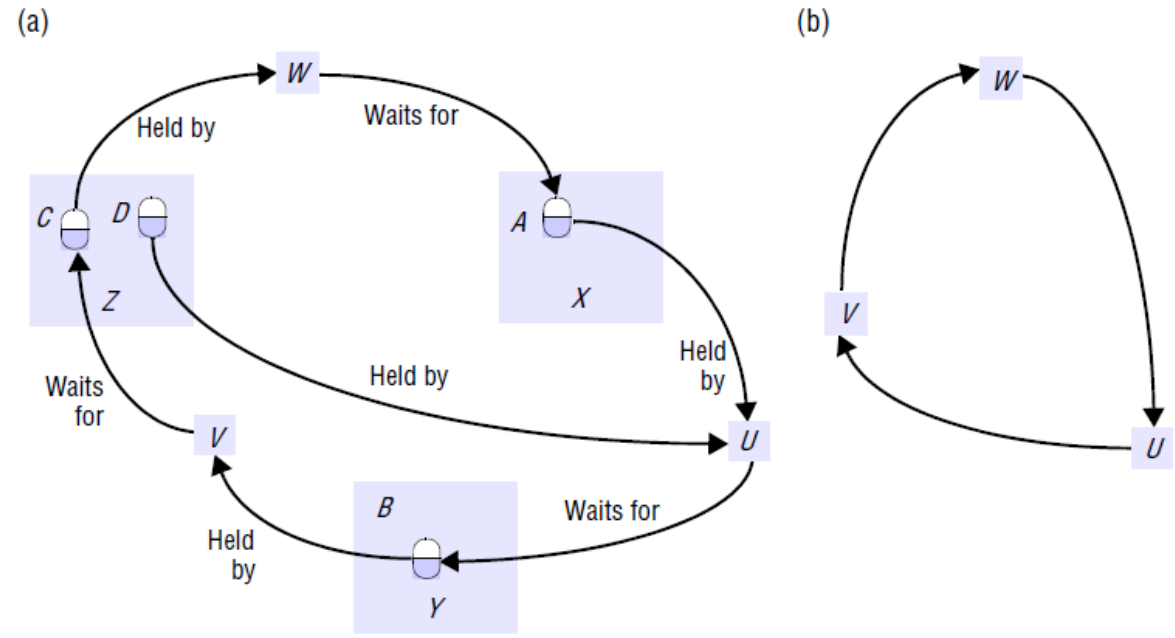
Performance of two-phase commit protocol

Provided that all goes well, that is that the coordinator, the participants and the communications between them do not fail.Tthe two-phase commit protocol involving *N* participants can be completed with *N canCommit?* messages and replies, followed by *N doCommit* messages. That is, the cost in messages is proportional to 3*N,* and the cost in time is three rounds of messages.

10. Explain about the distributed deadlocks.

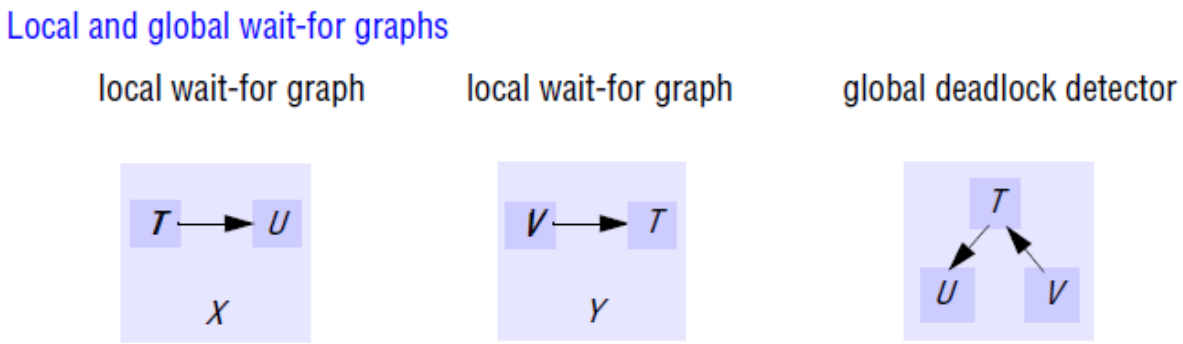
In a distributedsystem involving multiple servers being accessed by multiple transactions, a globalwait-for graph can in theory be constructed from the local ones. There can be a cycle inthe global wait-for graph that is not in any single local one – that is, there can be a*distributed deadlock*. Recall that the wait-for graph is a directed graph in which nodesrepresent transactions and objects, and edges represent either an object held by atransaction or a transaction waiting for an object. There is a deadlock if and only if thereis a cycle in the wait-for graph.

A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector. From time to time, each server sends thelatest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph. When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort. Centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. It suffers from the usual problems associated with centralized solutions in distributed systems – poor availability, lack of fault tolerance and no ability to scale.



Phantom deadlocks

A deadlock that is 'detected' but is not really a deadlock is called a *phantom deadlock*. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. As this procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.



Edge Chasing

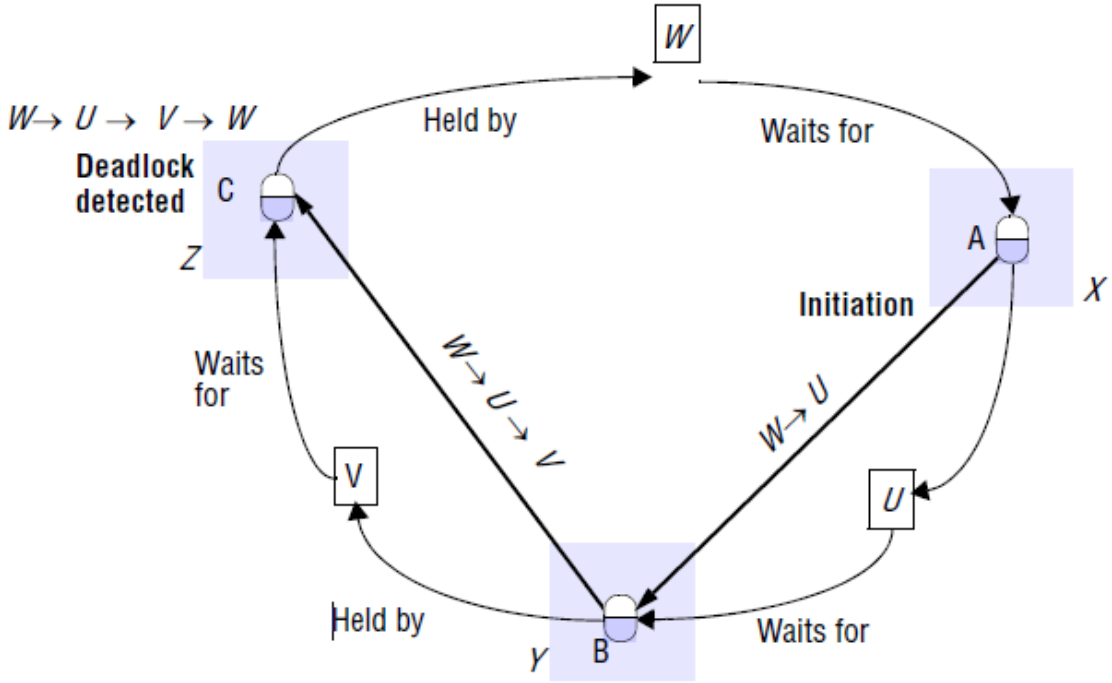
A distributed approach to deadlock detection uses a technique called *edge chasing* or *path pushing*. In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called *probes*, which follow the edges of the graph throughout the distributed system. A probe message consists oftransaction wait-for relationships representing a path in the global wait-for graph.
Edge-chasing algorithms have three steps:

Initiation: When a server notes that a transaction *T* starts waiting for another transaction *U*, where *U* is waiting to access an object at another server, it initiates detection by sending a probe containing the edge <*T* ->*U* > to the server of the object at which transaction *U* is blocked. If *U* is sharing a lock, probes are sent to all the holders of the lock. Sometimes further transactions may start sharing the lock later on, in which case probes can be sent to them too.

Detection: Detection consists of receiving probes and deciding whether a deadlock has occurred and whether to forward the probes. For example, when a server of an object receives a probe <*T* ->*U* >

(indicating that *T* is waiting for a transaction *U* that holds a local object), it checks to see whether *U* is also waiting. If it is, the transaction it waits for (for example, *V*) is added to the probe (making it <*T* ->*U*->*V* > and if the new transaction (*V*) is waiting for another object elsewhere, the probe is forwarded. In this way, paths through the global wait-for graph are built one edge at a time. Before forwarding a probe, the server checks to see whether the transaction (for example, *T*) it has just added has caused the probe to contain a cycle (for example, <*T* ->*U* ->*V* ->*T* >). If this is the case, it has found a cycle in the graph and a deadlock has been detected.

Resolution: When a cycle is detected, a transaction in the cycle is aborted to break the deadlock



11. Discuss the role of replication in the effectiveness of distributed systems.

Replication is a key to providing high availability and fault tolerance in distributedsystems. High availability is of increasing interest with the tendency towards mobilecomputing and consequently disconnected operation. Fault tolerance is an abidingconcern for services provided in safety-critical and other important systems.
The maintenance of copies of data at multiple computers is replication. Replication is a key to the effectiveness of distributed systems in that it can provide enhanced performance, high availability and fault tolerance. Replication is used widely. For example, the caching of resources from web servers in browsers and web proxy servers is a form of replication, since the data held in caches and at servers are replicas of one another.

Replication is a technique for enhancing services. The motivations for replication include:
Performance enhancement: The caching of data at clients and servers is by now familiar as a means of performance enhancement. Replication of immutable data is trivial: it increases performance with little cost to the system.

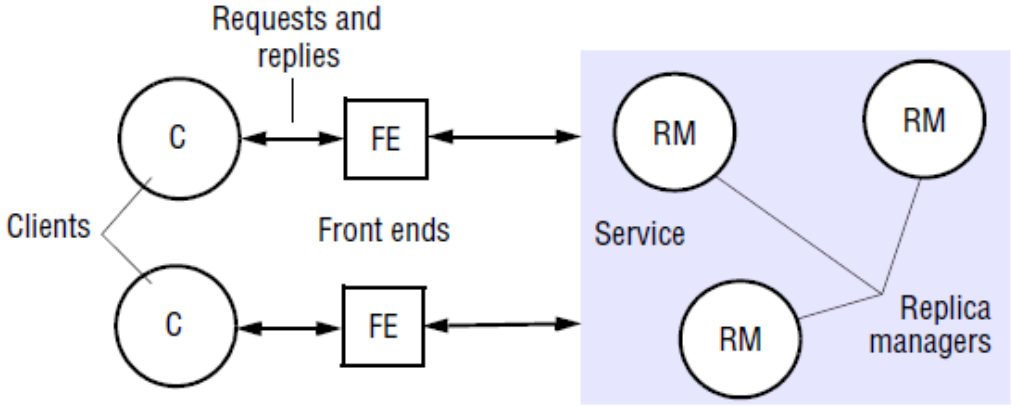
Increased availability: Users require services to be highly available. That is, the proportion of time for which a service is accessible with reasonable response times should be close to 100%. Apart from delays due to pessimistic concurrency control conflicts , the factors that are relevant to high availability are:
• server failures;
• network partitions and disconnected operation (communication disconnections that are often unplanned and are a side effect of user mobility).

Fault tolerance: Highly available data is not necessarily strictly correct data. It may be out of date, for example; or two users on opposite sides of a network partition may make updates that conflict and need to be resolved. A fault-tolerant service, by contrast, always guarantees strictly correct behaviour despite a certain number andtype of faults.

A common requirement when data are replicated is for *replication transparency*. That is, clients should not normally have to be aware that multiple *physical* copies of data exist. As far as clients are concerned, data are organized as individual *logical* objects and they identify only one item in each case when they request an operation to be performed. Furthermore, clients expect operations to return only one set of values. This is despite the fact that operations may be performed upon more than one physical copy in concert.

The other general requirement for replicated data – one that can vary in strength between applications – is that of consistency. This concerns whether the operations performed upon a collection of replicated objects produce results that meet the specification of correctness for those objects.

A basic architectural model for the management of replicated data



For the sake of generality, we describe architectural components by their roles and do not mean to imply that they are necessarily implemented by distinct processes (or hardware). The model involves replicas held by distinct *replica managers* ,which are components that contain the replicas on a given computer and perform operations upon them directly. This general model may be applied in a client-server environment, in which case a replica manager is a server. We shall sometimes simply call them servers instead. Equally, it may be applied to an application and application processes can in that case act as both clients and replica managers. For example, the user's laptop on a train may contain an application that acts as a replica manager for their diary.

A service that supports disconnected operation behaves differently from one that provides a fault-tolerant service. The phases are as follows:

Request: The front end issues the request to one or more replica managers: – *either* the front end communicates with a single replica manager, which in turn communicates with other replica managers; – *or* the front end multicasts the request to the replica managers.

Coordination: The replica managers coordinate in preparation for executing the request consistently. They agree, if necessary at this stage, on whether the request is to be applied (it might not be applied at all if failures occur at this stage). They also decide on the ordering of this request relative to others.

FIFO ordering: If a front end issues request *r* and then request *r*', any correct replica manager that handles *r*' handles *r* before it.

Causal ordering: If the issue of request *r* happened-before the issue of request *r*', then any correct replica manager that handles *r*' handles *r* before it.

Total ordering: If a correct replica manager handles *r* before request *r*', then any correct replica manager that handles *r*' handles *r* before it.

Execution: The replica managers execute the request – perhaps *tentatively*: that is, in such a way that they can undo its effects later.

Agreement: The replica managers reach consensus on the effect of the request – if any – that will be committed. For example, in a transactional system the replica managers may collectively agree to abort or commit the transaction at this stage.

Response: One or more replica managers responds to the front end. In some systems, one replica manager sends the response. In others, the front end receives responses from a collection of replica managers and selects or synthesizes a single response to pass back to the client.

UNIT – V
PART - A

1. **What is process migration?**

   Process migration is the relocation of a process from its current location (source node) to another node(destination node).The process can be either a non-preemptive or preemptive process. Selection of the process to be migrated, selection of the destination node and the actual transfer of the selected process are the three steps involved in process migration.

2. **What are the advantages of process migration?**

   Various advantages of process migration are:
   - Reduces average response time of processes.
   - Speeds up individual jobs.
   - Gains higher throughput.
   - Effective utilization of resources and reduces network traffic.

3. **What are the activities involved in process migration?**

   Migration of a process is a complex activity that involves many sub-activities. They are:
   1. Freezing the process on its source node and restarting it on its destination node.
   2. Transferring the process's address space from its source node to its destination node.
   3. Forwarding messages meant for the migrant process.
   4. Handling communication between cooperating processes that have been separated as a result of process migration.

4. **Mention the levels of transparency in process migration.**

   Transparency is an important requirement for a system that supports process migration.
   The two levels of transparency are:
   - **Object access level**: Minimum requirement for a system to support non-preemptive process migration.
   - **System call and interprocess communication level**: This facility is required to support preemptive process migration.

5. **What is Threads?**

   Threads are an efficient way to improve application performance through parallelism. Each thread of a process has its own program counter,its own register states, and its own stack.But all the threads shares the same address space.Threads are often referred to as lightweight process.

6. **What are the main advantages of using threads instead of multiple processes?**

   Threads has its own program counter,its own register states, and its own stack but shares the same address space
   Advantages of threads over multiple processes are:

- **Context Switching:** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For eg: they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc. In other words, it is relatively easier for a context switch using threads.
- **Sharing**: Treads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file etc

7. Mention the models used to organize the threads of a process.

- **Dispatcher-workers model** – In this model, process consists of a single dispatcher thread and multiple worker threads.
- **Team model** – In this model, all threads behave as equals and there is no dispatcher-worker relationship for processing client's requests.
- **Pipeline model** – In this model, the threads of a process are organized as a pipeline where the output data from one thread is used for processing by the other threads.

8. Define critical region.

A segment of code in which a thread may be accessing some shared variable is called critical region. Multiple threads should not access the same data simultaneously. Hence the execution of critical regions in which the same data is accessed by the threads must be mutually exclusive in time.

9. Define mutex variable.

Mutex variable is like a binary semaphore that is always in one of the two states locked or unlocked. Mutex variables are used to implement mutual exclusion technqiues. A thread that wants to execute in a critical region performs a lock operation over the mutex variable which has to be in unlocked state.

10. Mention some library procedures for managing the threads.

Some of the library procedures for managing threads are:

- pthread_create – Creates a new thread in the same address space as the calling thread.
- pthread_exit – Terminates the calling thread.
- pthread_join – It makes the calling thread to block itself and waits until thread specified in the routine's argument terminates.
- pthread_detach – Used by the parent thread to disown a child thread.
- pthread_cancel – Used by a thread to kill another thread.

11. Mention the types of mutex variables.

Generally the types of mutex variables supported are:

- Fast – Fast mutex variable causes a thread to block when the thread attempts to lock an already locked mutex variable.
- Recursive – It allows a thread to lock an already locked mutex variable.
- Nonrecursive – It neither allows a thread to lock an already locked mutex variable nor causes the thread to block.

12. Write short notes on resource management.

Distributed systems are characterized by resource multiplicity and system transparency. Efficient resource management is implemented by a resource manager. The resource manager schedules the processes to make use of the system resources in such a manner that resource usage,resource time,network congestion and scheduling overhead are optimized.

13. What are the features of global scheduling algorithm?

Desirable features of a good global scheduling algorithm are:

- No a priori knowledge about the processes to be executed by the scheduling algorithm.
- Dynamic in nature.
- Quick decision making capability.
- Balanced system performance and scheduling overhead.
- Stablility

- Scalability
- Fault tolerance
- Fairness of service

14. What is Task assignment approach?

Task assignment approach is the technique used for scheduling processes of a distributed system.In this approach each process submitted by a user for processing is viewed as a collection of related tasks and these tasks are scheduled to suitable nodes so as to improve performance

15. Define Load balancing approach.

Load balancing algorithm or load leveling algorihm aims for better resource utilization. Load balancing approach tries to balance the total system load by transparently transferring the workload from heavily loaded nodes to lightly loaded nodes in order to ensure overall good performance.

16. What are the issues in designing load balancing algorithm.

Designing a good load balancing algorithm is a difficult task because of the following issues:

- Load estimation policies – Estimates the workload
- Process transfer policy – Determines whether to execute a process locally or remotely.
- State information exchange policy – Deals about the load information exchange.
- Location policy – Selects the node to which the process to be sent.
- Priority assignment policy – Determines the priority of execution of local and remote process.
- Migration limiting policy – Decides the number of times a process can migrate.

17. What is load-sharing approach?

Load sharing approach also aims for efficient resource utilization. It attempts to ensure that no node is idle while processes wait for service at some other node. In this approach it is sufficient to know whether a node is busy or idle. Load sharing approach does not attempt to balance the average workload on all the nodes of the system.

18. State the differences between the static and dynamic load balancing alogrithms.

Static load balancing algorithms	Dynamic load balancing algorithms
It uses only information about the average behavior of the system ignoring the current state of the system.	It reacts to the system state that changes dynamically.
Potential of static algorithms is limited	It provides greater performance benefits.
These algorithms are simpler.	These algorithms are complex in nature

19. State the differences between the deterministic and probabilistic load balancing alogrithms.

Deterministic load balancing algorithms	Probabilistic load balancing algorithms
These algorithms use the information about the properties of the nodes and the characteristics of the processes to be scheduled.	These algorithms use the information about the static attributes of the system such as number of nodes, network topology etc.
This approach provides optimized performance.	It often suffers from having poor performance.
It costs more to implement.	It is easier to implement.

20. What is threshold?

Most of the load balancing algorithm use threshold policy to decide whether a node is lightly or heavily loaded. Threshold value of a node is the limiting value of its workload. A new process at a node is accepted locally for processing if the workload of the node is below its threshold value at that time. Threshold value is determined either by static or dynamic policy.

21. What is static policy and dynamic policy?

- **Static policy**: In this method, each node has a predefined threshold value on its processing capability. Main advantage of this method is that no exchange of state information among the nodes is required in deciding the threshold value.
- **Dynamic policy**: In this method, the threshold value of a node is calculated as a product of the average workload of all the nodes and a predefined constant. It gives a more realistic value for the threshold value based on the information exchange among the nodes.

22. Write the priority assignment policies.

Priority assignment policies need to be devised in order to implement process migration in a distributed operating system. It can be any one of the following:

- Selfish – Local processes given higher priority.
- Altruistic – Remote processes given higher priority.
- Intermediate: Priority of the processes depends on the number of local processes and number of remote processes at concerned node.

23. Write about sender-initiated location policy.

Sender-initiated location policy is a location policy where heavily loaded nodes search for lightly loaded nodes to which work may be transferred. In this method, the node either broadcasts the message or randomly probes the other nodes to find whether it is a lightly loaded node to accept one or more processes.

24. What is receiver-initiated policy?

In this location policy, lightly loaded nodes search for heavily loaded nodes from which work may be transferred . When the load's value of a node falls below the threshold value, it broadcasts a message indicating its willingness to receive processes for executing or randomly probes the other nodes to search to find heavily loaded nodes.

PART – B

1. What are the desirable features of a good process migration mechanism?

A good process migration mechanism must possess transparency, minimal interferences, minimal residue dependencies,efficiency, robustness and communication between coprocesses.

Transparency

Transparency is an important requirement for a system that supports process migration.

Following levels of transparency can be identified:

Object access level: Transparency at the object access level is the minimum requirement for a system to support non-preemptive process migration facility. If a system supports transparency at object access level, access to objects such as files and devices can be done at location independent manner.

System call and interprocess communication level: In this case, a migrated process does not continue to depend upon its originating node after being migrated. It is necessary that all system calls including interprocess communication are location independent.

Minimal Interference

Migration of a process should cause minimal interference to the progress of the process involved and to the system as a whole. One method to achieve this is to minimize the freezing time of the process being executed. Freezing time is defined as the time period for which the execution of the process is stopped for transferring its information to the destination node.

Minimal Residual Dependencies

A migrated process should not in any way continue to depend on its previous node once it has started executing on its new node. Else the following things occur:

The migrated process will continue to impose a load on its previous node.

A failure or reboot of the previous node will cause the process to fail.

Efficiency

The main sources of inefficiency are:

Time required for migrating a process.

Cost of location an object

Cost of supporting remote execution once the process is migrated.

Robustness

It is the case where the failure of one node other than the one on which a process is currently running should not in any way affect the accessibility or execution of a process.

Communication between coprocesses of a job

Communication between coprocesses of a job should be carried out a minimal cost and also should be able to directly communicate with each other irrespective of their locations.

2. **Explain benefits of process migration?**

Reducing average response time of processes: Process migration facility is used to reduce the average response time of the processes of a heavily loaded node by migrating some of the processes that is either idle or whose processing capacity is underutilized.

Speeding up individual jobs: There are two methods to speed up individual jobs. One method is to migrate the tasks of a job to different nodes to execute them concurrently. Another method is to migrate a job to a node having a faster CPU.

Gaining higher throughput: The throughput is increased by using a suitable load balancing approach. And it also executes a mixture of I/O and CPU bound processes on a global basis to increase the throughput.

Utilizing resources effectively: In a distributed system there are various resources such as CPU's, printers, storage devices etc. Depending upon the nature of the process the resources has to be allocated to the suitable node.

Reducing network traffic: Migrating a process closer to the resources it is using is a mechanism used to reduce the network traffic. It can also migrate and cluster two or more processes which frequently communicate with each other on the same node of the system.

Improving system reliability: System reliability can be improved in two ways. One way is to migrate a critical process to a node whose reliability is higher than other nodes in the system. Another way is to migrate a copy of the critical process to some node and execute both the original and copied processes concurrently on different nodes.

Improving system security: A sensitive process may be migrated and run on a secure node that is not directly accessible to general users thus improving the security of that process.

3. **What are the Address Transfer Mechanisms available for process migration?**

Generally process migration involves the transfer of the following information from the source node to the destination node:

Process's state that includes execution status, scheduling information, information about main memory, I/O states etc.

Process's address space that includes code,data and stack of the program.

Address space transfer mechanisms followed are:

Total freezing, pretransferring and transfer on reference.

Total Freezing

In this method, process's execution is stopped while its address space is transferred.

Disadvantage of this method is if a process is suspended for a long time during migration timeouts may occur.

Pretansferring

In this method, address space is transferred while the process is still running on the source node.Pretransferring or precopying is done as an initial transfer of the complete address space followed by repeated transfers of the pages that are modified.

In this pretransfer mechanism, first transfer moves the entire address space and takes longest time. The second transfer moves only those pages that were modified during the first transfer.

Transfer on Reference

This method is based on the assumption that processes tend to use only a relatively a small part of their address spaces while executing.In this demand-driven copy-on reference approach, a page of the migrant process's address space is transferred from source node to its destination node only when referenced.

4. **Discuss the Process Migration Message Forwarding Mechanisms.**

In migration of the process,it must be ensured that all pending, en-route and future messages arrive at the process's new location. The messages to be forwarded to the migrant process's new location can be classified into the following:

Type 1: Messages received at the source node after the process's execution has been stopped on its source node and the process's execution has not yet been started on its destination node.

Type 2: Messages received at the source node after the process's execution has started on its destination node.

Type 3: Messages that are to be sent to the migrant process from any other node after it has started executing on the destination node.

Generally followed Message Forwarding Mechanisms are:

Mechanism of Resending the message:

In this method, messages of type 1 and 2 are returned to the sender as not deliverable or are simply dropped with the assurance that the sender of the message is storing a copy of the data and is prepared to retransmit it.Type 3 messages are sent directly to the process's destination node. Drawback of this mechanism is it is non transparent to the processes interacting with the migrant process.

Origin Site Mechanism

The process identifier of these systems has the process's origin site embedded in it and each site is responsible for keeping information about the current locations of all processes created on it.In this method, the messages for a particular process are always sent to its origin site.Drawback of this mechanism is that the failure of the origin site will disrupt message forwarding mechanism.

Link Traversal Mechanism

In this mechanism, the messages of type 1 are placed in a message queue in the source node.Messages of type 2 and 3 are redirected using the forwarding address called link which is placed in the source node pointing to the destination node of the migrant process.Message process address is the main part of the link that contains 2 components.One is systemwide, unique, process identifier and another is the last known location of the process.

Link Update Mechanism

In this mechanism, processes communicate via location-independent links which are capable of duplex communication channels. During the transfer phase, source node sends link-update messages to the kernels controlling all of the migrant process's communication partners. Messages of type 1 and 2 are forwarded to the destination node by the source node. Type 3 messages are sent directly to the destination node.

5. **Discuss the issues related to thread programming, thread lifetime, thread synchronization, scheduling and implementation.**

Threads are used to improve application performance through parallelism. Each thread of a process has its own register states,program counter and its own stack. Threads are often referred to as lightweight processes.

A system that supports threads facility must provide a set of primitives to its users for thread related operations and those primitives forms the thread package.Some of the issues involved in it are:

Thread Creation

Threads can be created either statically or dynamically. In static approach, the number of threads of a process remains fixed for its entire lifetime. Fixed stack is allocated for each thread in this approach. On the other hand, in dynamic approach number of threads for a process keeps changing dynamically. New threads are created when needed. Here the stack size for each thread is passed as parameter in the system call.

Thread Termination

A thread may either destroy itself after it finishes its job or it can be killed from outside using kill command.In many cases threads are never terminated. In some cases, threads are never killed until the process terminates.

Thread Synchronization

Generally all threads share a common address space, some mechanism must be used to prevent multiple threads from trying to access the same data simultaneously. A segment of code in which a thread may be accessing some shared variable is called critical region. Execution of critical regions in which the same data is accessed by the threads must be mutually exclusive in time. A mutex variable is like a binary semaphore that is always in one of the two states locked or unlocked.

If the mutex variable is already locked depending on the implementation the lock operations is handled in one of the following ways.

1. The thread is blocked and entered in a queue of threads waiting on the mutex variable.
2. A status code indicating failure is returned to the thread.

When a thread finishes executing in its critical region it performs an unlock operation on the corresponding mutex variable.

A condition variable is associated with the mutex variable and reflects a Boolean state of that variable. Wait and signal are two operations normally provided for a condition variable.Condition variables are often used for cooperation between threads.

Thread Scheduling

Another important issue in the design of threads is how to schedule the threads.Some of the special features of threads scheduling are:

Priority assignment facility - In a simple scheduling algorithm, threads are scheduled on a first-in,first-out basis or round robin policy is used to timeshare the CPU cycles among the threads on a quantum-by-quantum basis.Priority based scheduling algorithm can be either preemptive or non-preemptive.Higher priority thread always preempts the lower priority thread.

Flexibility to vary quantum size dynamically - Generally in round robin scheduling algorithm, a fized quantum size is used to timeshare the CPU cycles.Fixed length quantum is not suitable in a multiprocessor system, hence quantum length can be kept as variable.In this case, it gives good response time to short requests.

Handoff Scheduling - A handoff scheduling scheme allows a thread to name its successor if it wants to. In this case, after sending a message to another thread, the sending thread can give up the CPU and request the receiving thread be allowed to run next.

Affinity Scheduling - In this scheduling, a thread is scheduled on the CPU it last ran on in hopes that part of its address space is still in that CPU's cache.It is used in multiprocessor system.

6. What are the desirable features of good global scheduling algorithms?

No a Priori knowledge about the processes – A good process scheduling algorithm should operate with absolutely no a prori knowledge about the processes to be executed.

Dynamic in nature – A good process scheduling algorithm should be able to take care of the dynamically changing load of the various nodes of the system. This feature also requires that the system support preemptive process migration facility.

Quick Decision-Making Capability – A process scheduling algorithm must make quick decisions about the assignment of processes to processors.

Balanced system performance and Scheduling overhead–Global scheduling algorithms collect global state information and use this information in making process assignment decisions.An overhead is also is involved in collecting more information about the global state of the systems.

Stablity–A scheduling algorithm is said to be unstable if it can enter a state in which all the nodes of the system are spending all of their time migrating processes without accomplishing any useful work.This form of fruitless migration of processes is called as processor thrashing.

Scalability–A scheduling algorithm should be capable of handling small as well as large networks.

Fault tolerance – A scheduling algorithm should not be disabled by the crash of one or more nodes of the system and hence it is said to be fault tolerant.

Fairness of Service – Fairness of service is the case where if two users simultaneously initiates equivalent processes expect to receive about the same quantity of service. Load sharing concept is introduced to facilitate this service.

7. Describe in detail about the task assignment approach?

In this approach, a process is considered to be composed of multiple tasks and the goal is to find an optimal assignment policy for the tasks of an individual process.

Assumptions made in this approach are:

- A process has already been split into pieces called tasks.
- Amount of computation required by each task and the speed of each processor are known.
- Cost of processing each task on every node of the system is known.
- Interprocess Communication(IPC) costs between every pair of tasks is known.

The main goals to be achieved in task assignment algorithms are:

- Minimization of IPC costs.
- Quick turnaround time for the complete process.
- A high degree of parallelism
- Efficient utilization of system resources is gained.

Considering an assignment problem, the intertask costs and the execution costs are given in a tabular form with tasks(t1,t2..etc) and the nodes(n1,n2,..).Infinite costs of a particular task against a particular node indicates that the task cannot be performed in that node.Serial assignment of the tasks along with the node is illustrated in a tabular form.

The problem of finding an assignment of tasks to nodes that minimizes the total execution and communication costs is analyzed using a network flow model.

Optimal assignment is found using a static assignment graph.A cutset is a graph defined with a set of edges such that when these edges are removed, the nodes of the graph is partitioned into two disjoint subsets such that the nodes in one subset are reachable from n1 and the nodes in the other are reachable from n2.

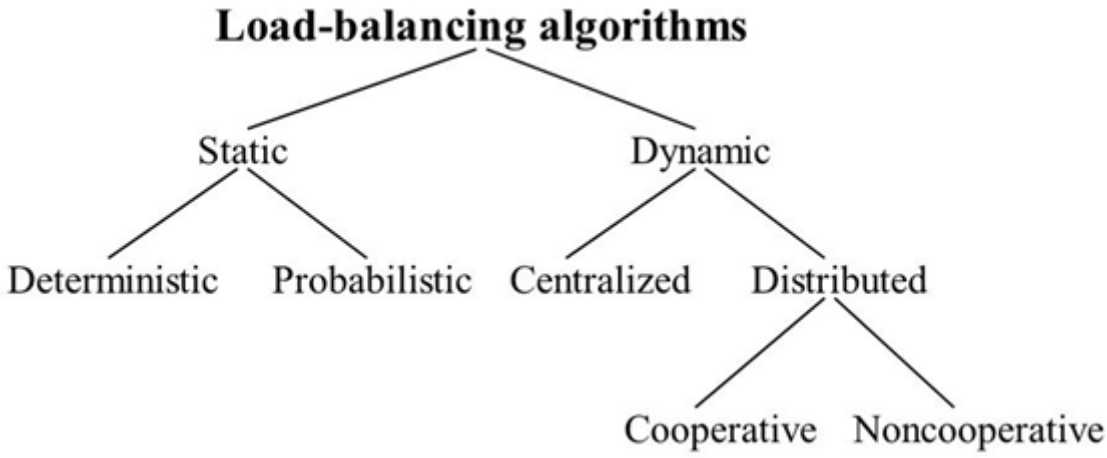
8. Discuss load balancing approach in distributed systems in detail.

Scheduling algorithms using load balancing approach are known as load-balancing algorithms or load-leveling algorithms. Load balancing algorithm tries to balance the total system load by transparently transferring the workload from heavily loaded nodes to lightly loaded nodes. It ensures overall good performance of the system. Basic goal of almost all load balancing algorithms is to maximize the system throughput.

Taxonomy of Load Balancing Algorithms

Static Vs Dynamic

Static algorithms use only information about the average behavior of the system ignoring the current state of the system. On the other hand, dynamic algorithms react to the system state that changes dynamically.Static algorithms are simpler in nature but give a less performance compared to dynamic algorthims.



Deterministic Vs Probabilistic

Static load-balancing algorithms may be either deterministic or probabilistic. Deterministic algorithms use the information about the properties of the nodes and the characteristics of the process to be scheduled to deterministically allocate processes to nodes. Probabilistic load balancing algorithms uses information regarding static attributes of the system such as number of nodes, processing capability of each node, network topology etc.

Centralized Vs Distributed

In Centralized dynamic scheduling algorithm the responsibility of scheduling physically resides on a single node. In this the system state information is maintained with a centralized server node. In Distributed dynamic scheduling algorithm, the information is distributed among the various nodes of the system.

Cooperative Vs Noncooperative

In noncooperative algorithms, individual entities act as autonomous entities and make scheduling decisions independent of the actions of other entities. In cooperative algorithms, distributed entities cooperate with each other to make scheduling decisions.

9. Explain load estimation approaches.

The main goal of load balancing algorithms is to balance the workload on all nodes of the system. A node's workload can be estimated based on some measurable parameters. The parameters include time-dependent and node-dependent factors such as the following :

Total number of processes on the node at the time of load estimation.

Resource demands of the processes

Instruction mixes of the processes

Architecture and speed of the node's processor

The main challenge of this method is to calculate the node's workload. One of the measures to calculate the node's workload is the sum of the remaining service time of all the processes on that node.

Following methods are used for the above purpose:

Memoryless method:This method assumes that all processes have the same expected remaining service time independent of the time used so far.

Pastrepeats: This method assumes that the remaining service of a process is equal to the time used so far by it.

Distribution method: If the distribution of service times is known, the associated processes remaining service time is the expected remaining time conditioned by the time already used.

Central processing unit(CPU) utilization is defined as the number of CPU cycles actually executed per unit of real time.

10. Discuss the Priority assignment policy for Load-balancing algorithms.

When process migration is supported by a distributed operating system, it becomes necessary to devise a priority assignment rule for scheduling both local and remote processes at a particular node.

One of the following priority assignment rules may be used for this purpose:

Selfish : Local processes are given higher priority than remote processes.

Altruistic : Remote processes are given higher priority than local processes

Intermediate: The priority of processes depends on the number of local processes and the number of remote processes at the concerned node.

A study on the effect of the above three priority assignment policies on the overall response time performance reveals the following results:

The selfish priority assignment rule yields the worst response time performance of the three policies.

The altruistic priority assignment rule achieves the best response time performance of the three policies.

The performance of the intermediate priority assignment rule falls between the other two policies.

Migration Limiting Policies

Another important policy to be used by a distributed operating system that supports process migration is to decide about the total number of times a process should be allowed to migrate.One of the following two policies may be used for this purpose:

Uncontrolled: In this case, a remote process arriving at a node is treated just as a process originating at the node.Under this policy, a process may be migrated any number of times.

Controlled: To overcome the instability problem of the uncontrolled policy most systems treat remote processes different from local processes and use a migration count parameter tp fix a limit on the number of times that a process may migrate.

11. Discuss about the load sharing approach.

Proper utilization of the resources of a distributed system serves to be challenge and this is met through some of the load sharing approaches. There are many important decisions to be made on load sharing approaches.

Load Estimation policies

Since load sharing algorithms simply attempt to ensure that no node is idle while processes wait for service at some other node, it is sufficient to know whether a node is busy or idle.

Process Transfer policies

Since load sharing algorithms are normally interested only in the busy or idle states of a node, most of them employ the all-or-nothing strategy. In the all-or-nothing strategy, a node that becomes idle is unable to immediately acquire new processes to execute even though processes wait for service at other nodes, resulting in a loss of available processing power in the system.

Location policies

Location policy decides the sender node or the receiver node of a process that is to be moved within the system for load sharing. Depending on the type of the node that takes the initiative there are two different location policies. They are:

Sender-Initiated policy

In this policy, the sender node of the process decides where to send the process. Heavily loaded nodes search for lightly loaded nodes to which work has to be transferred.When a node's load becomes more than the threshold value, it either broadcasts a message or randomly probes the other nodes one by one to find a lightly loaded node that can accept one or more process.

Receiver-Initiated policy

In this policy, the receiver node of the process decides where to get the process. Lightly loaded nodes search for heavily loaded nodes from which work can be transferred.When a node's load falls below the threshold value it either broadcasts a message indicating its willingness to receive process or randomly sends probes to heavily loaded nodes.